Extending Camille

Ingo Weigelt

September 21, 2010

Abstract

The Camille Text Editor is a successful extension for the Rodin Platform. It allows users to edit Event-B models in a plain text editor, as opposed to the default form-based editor.

Currently, problems arise when a plug-in extends the Event-B language. The form-based editor allows plug-ins to extend it, but there is no such support in Camille yet. In this document, we propose a generic mechanism that would allow plug-ins to extend the syntax of Camille.

The key idea is to use a block parser to process the textual representation of the model. Blocks containing information that belongs to the Rodin core is processed by the Camille parser. Blocks that belong to a certain plug-in are delegated to the plug-in that owns it as plain text.

We believe that this approach keeps the textual representation of the model clean and readable, while giving enough control to the plug-in authors to extend the syntax as they see fit. Amongst others, we propose: single and multi-line blocks; name space management; free choice of parser technology; and robustness, concerning syntax violations and detection of missing plug-ins.

1 Introduction

While the Camille Text Editor is popular for editing Event-B models, users currently cannot use it when they use certain plug-ins, as the editor does not support the extensions of those plug-ins. In this document, we offer a solution to this problem.

In Section 2, we will describe the new architecture in detail. In Section 3, we demonstrate our ideas on a number of existing Rodin plug-ins.

2 The Block-Parser

Most plug-ins only need the option to extend the base Event-B grammar, i.e. they need to add additional rules at certain points. Therefore it is sufficient to allow the insertion of plug-in-specific code at well defined points within an Event-B model.

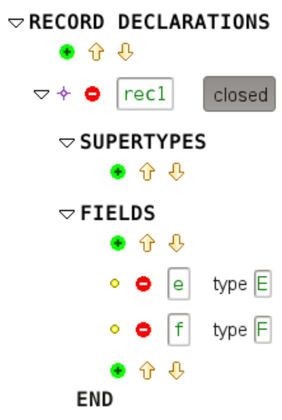
Our proposal is to implement another parser that filters out these additional blocks and dispatches them for processing. With this block-parser we can divide a textual specification into plain Event-B code and a set of plug-in specific code-blocks. The former can then be parsed by the existing Event-B parser without any modifications to the grammar. The latter is simply passed to the corresponding plug-in. Consequently the plug-in developers have to implement their own parser for their syntax extension(s)[1]. Since this parser is completely independent, developers are free to use the parser generator of their choice.

To implement this block-parser we need to overcome two major hurdles: First we need a way to reliably detect the additional blocks and second we need to determine which plug-in is responsible for each of them.

The easiest way to identify the blocks is to guard them by a unique character-sequence that can not appear elsewhere in an Event-B model. For instance we can open a block with `$$` and close it with `$$END`.

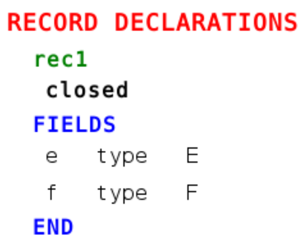
To identify the corresponding plug-in for each block we can use the first word after the opening `$$` as a registered keyword. This keyword is bound to a specific syntax-extension coming with a plug-in. Thereby it is possible for a plug-in to contribute more than one extension.

As an example, lets consider the records extension. This plug-in adds a new section *record declarations* to the Rodin editor and the pretty printer (Figure 1).



```
RECORD DECLARATIONS
  rec1  closed
    SUPERTYPES
    FIELDS
      e  type E
      f  type F
  END
```

(a) editor



```
RECORD DECLARATIONS
 rec1
  closed
 FIELDS
  e   type   E
  f   type   F
 END
```

(b) pretty-printer

Figure 1: *record declarations* section

The new section as implemented in the Rodin editor already gives us a possible syntax extension. To get a global syntax which is close to the one used by the pretty printer the records plug-in could register the keyword `record` and bring its own parser for the inner syntax.

[1]It should be possible to provide a generic default implementation

The record declaration depicted in Figure 1 can then for instance be typed as:

```
$$RECORD DECLARATIONS
   rec1 closed
   FIELDS
     e : E
     f : F
$$END
```

2.1 Keyword conflicts

Since the keywords are chosen by the plug-in developers independently, we have to find a way to deal with conflicting keyword registrations. Our proposal is to use *import statements* which define the concrete keyword used in a file for each imported syntax-extension. For example, let `org.Event-B.records.syntax` be the unique id of the syntax-extension contributed by the *records plug-in* used in the example above. Then at the beginning of the context definition we have to add the statement

```
uses org.eventb.records.syntax as RECORD
```

in order to bind the keyword `RECORD` to the *records-extension*. Additionally we could make this import statements mandatory. This has the advantage that it is always well documented which plug-ins are used and which syntax addition belongs to which plug-in.

2.1.1 Versioning

We will introduce a versioning mechanism. The details have not yet been worked out in detail. We will probably make the target version of the plug-in part of the uses statement. We will probably model the version number and ranges after the Eclipse versioning scheme.

For instance, the "uses" statement may look as follows:

```
uses org.eventb.records.syntax 1.2 as RECORD
```

In this example, the model would require the records plug-in version 1.2 or better, but not including 2.0 or higher.

Plug-in developers would be required to increase the second number when they make a backward compatible change to the syntax. When the change is not compatible, they must increase the first number and reset the second number to zero.

These version numbers must not be confused with the version of the plug-in.

2.2 Single-line extensions

Some plug-ins only extend the Event-B syntax by relatively short expressions. For instance the *modularisation* plug-in adds new attributes (*group* and *final*) to events. Here only one or two words have to be added to an event declaration. To reduce the syntactical clutter in such cases we plan to offer single-line blocks. Those are opened by a single `$` and are closed by the end of the line (See below for an example).

2.3 Joinpoints

Regarding their position within a model, we can distinguish between two kinds of syntax extensions. On one hand we have extensions adding new sections, e.g. top level elements, to a machine or a context. An example is the *record declarations* section described above. For such extensions, the exact location of the addition does not matter. However, to gain consistency when the source code has to be generated (by pretty-printing) this location should be limited to exactly one point. On the other hand some plug-ins, like the *modularisation* plug-in, need to extend the syntax for certain model elements such as events or invariants. This has two major implications for the parser design: First, a block can be inserted within or just before nearly every model element. Second, not only the block content but also its semantic position, i.e. the Event-B element it is 'attached' to, have to be passed to the responsible plug-in.

2.4 Pretty-printing

Since it is sometimes necessary to automatically (re)generate the source code from a given model, each plug-in has to implement a pretty-printer for its syntax extension(s).

2.5 Comments

We will require all extensions to comply with the comment conventions used by Camille. Camille accepts two kinds of comments: single line comments start with a double slash ("//") and continue until the end of the line. Multi line comments start with slash star ("/*") and end with star slash ("*/").

3 Case Studies: Existing Plug-ins

3.1 Records Extension

The *records extension* plug-in provides two new top level sections: *Record Declarations* and *Record Extensions.* See [2] and [3].

Syntax used in Documentation

The following syntax is used in the documentation in [3]:

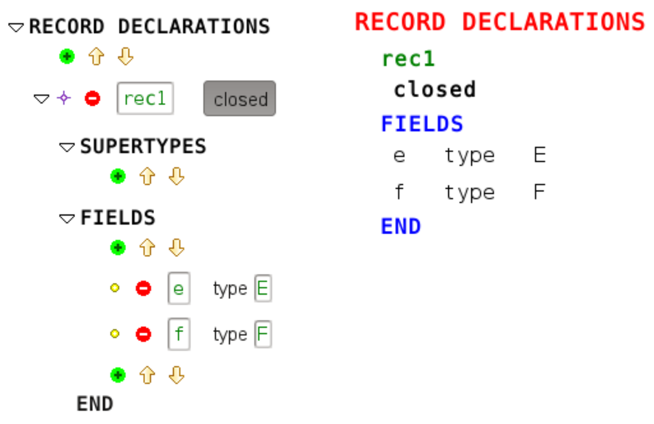
Example for a record structure R with fields e and f:

$$\boxed{\begin{array}{llll} \textbf{RECORD} & R & :: & e \in E \\ & & & f \in F \\ & & & \vdots \end{array}}$$

Extending Structured Types:

$$\boxed{\begin{array}{llll} \textbf{EXTEND RECORD} & R & \textbf{WITH} & g \in G \\ & & & h \in H \\ & & & \vdots \end{array}}$$

Syntax used in Rodin

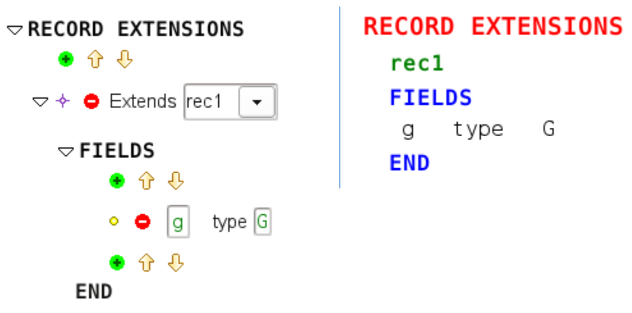


```
RECORD DECLARATIONS
    rec1    closed
    SUPERTYPES
    FIELDS
        e    type E
        f    type F
    END
```

```
RECORD DECLARATIONS
  rec1
   closed
  FIELDS
   e   type   E
   f   type   F
  END
```

(a) editor (b) pretty-printer

Figure 2: *record declarations* section



```
RECORD EXTENSIONS
    Extends rec1
    FIELDS
        g    type G
    END
```

```
RECORD EXTENSIONS
  rec1
  FIELDS
   g   type   G
  END
```

(a) editor (b) pretty-printer

Figure 3: *record extensions* section

Example syntax for Camille

The syntax extension for records is imported with a singe keyword `RECORD`:
(We could allow a white-space between `$$` and the keyword?)

Example 1:

```
uses org.eventb.records as RECORD

$$ RECORD DECLARATIONS
   rec2
   FIELDS
     e : E
     f : F
$$ END

$$RECORD EXTENSION
    rec2
    FIELDS
      g : G
$$END
```

Example 2:

```
uses org.eventb.records as RECORD

$$ RECORD DECLARATIONS
   rec2
   FIELDS
     e : E
     f : F

   RECORD EXTENSION
    rec2
    FIELDS
      g : G
$$END
```

Alternatively the plug-in could also register two different keywords `RECORD_DECLARATIONS` and `RECORD_EXTENSIONS`.

3.2 Modularisation

The *Modularisation Plug-in* [4] brings the following additions:

- **INTERFACE** - A new type of Event-B component. This is an independent type, hence no addition to the base grammar is needed. Can be implemented as an independent file type and parser. Nevertheless reusing existing parser rules is desirable.
- new machine constructs: **IMPLEMENTS** and **USES**

- new event attributes: **group** and **final**
- the ability to write operation calls in event actions. (Handled by the mathematical parser)

3.2.1 Syntax used in documentation

The following syntax is used in the documentation in [4]:

Implementing a Module

```
MACHINE m
IMPLEMENTS interface
```

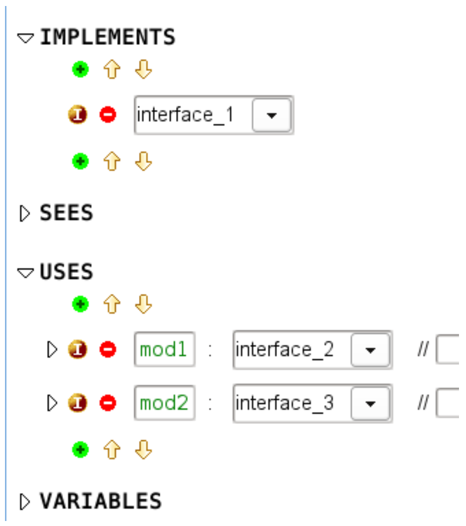
Importing a Module

```
USES   prefix1 : module1
       prefix2 : module2
       ...
```

Event attributes
Example taken from [5].

```
EVENTS
  button   =   FINAL GROUP Button
               WHEN
                  current = empty
               ...
```

3.2.2 Syntax used in Rodin



IMPLEMENTS

interface_1

SEES

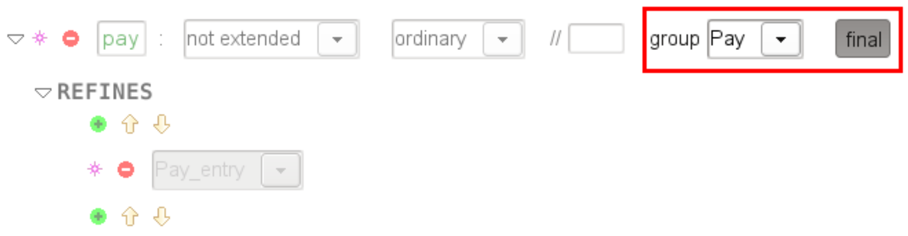
USES

mod1 : interface_2 //

mod2 : interface_3 //

VARIABLES

Figure 4: *implements* and *uses* sections



pay : not extended ordinary // group Pay final

REFINES

Pay_entry

Figure 5: New event attributes

Example Syntax for Camille

Syntax examples for IMPLEMENTS and USES:

Example 1: A syntax close to the one used in the Rodin editor

```
uses uk.ac.ncl.eventb.mod_feature.syntax.implements as IMPLEMENTS
uses uk.ac.ncl.eventb.mod_feature.syntax.uses as USES

MACHINE m

$$IMPLEMENTS
  interface_1
$$END

$$USES
  mod1 : interface_2
  mod2 : interface_3
$$END
...
```

Alternatively or additionally the plug-in could allow a single line version of the statements:

```
...
MACHINE m
$IMPLEMENTS interface
$USES mod1 : interface_2
$USES mod2 : interface_3
...
```

Example 2: All additional code is encapsulated in a block to clarify that it belongs to the modularisation plug-in.

```
uses uk.ac.ncl.eventb.mod_feature.syntax.implements as MODULARISATION

MACHINE m

$$MODULARISATION
  IMPLEMENTS interface_1

  USES
    mod1 : interface_2
    mod2 : interface_3
$$END
...
```

Syntax examples for event attributes:

Example 1:

```
uses uk.ac.ncl.eventb.mod_feature.syntax.event.final as final
uses uk.ac.ncl.eventb.mod_feature.syntax.event.group as group
...
events
  event pay
    $final
    $group Pay
    refines Pay_entry
    where ...
```

Example 2:

```
uses uk.ac.ncl.eventb.mod_feature.syntax as modularisation
...
events
  event pay refines Pay_entry
    $modularisation final, group: pay
    where ...
```

3.3 Model Decomposition

The *Model Decomposition* plug-in [6] requires syntax additions to

- declare variables as either *shared* or *private* and
- declare events as either *internal* or *external*.

3.3.1 Syntax Used in Rodin

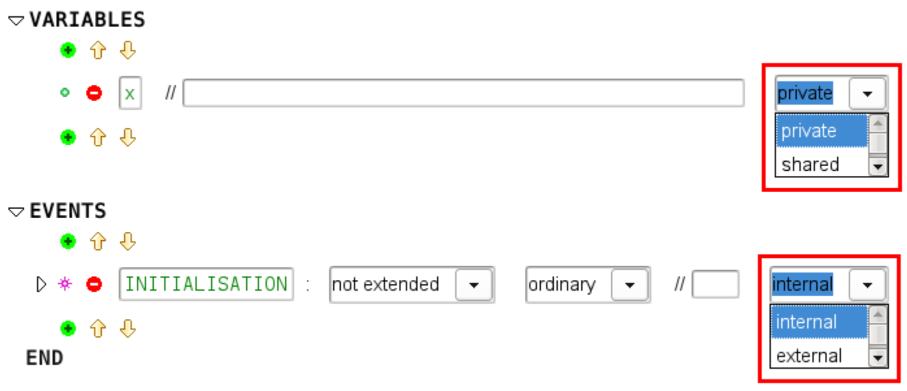


Figure 6: New variable and event attributes

3.3.2 Example Syntax for Camille

In this example only the attributes *shared* and *external* are added to the syntax. The default declaration is *private* for a variable and *internal* for an event.

```
uses ch.ethz.eventb.decomposition.syntax.shared   as shared
uses ch.ethz.eventb.decomposition.syntax.internal as external

machine m0
...
variables
  x
  y $shared
...
events
  $external
  event evt1
    then ...
```

3.4 Group refinement plug-in

The group refinement plug-in uses the four keywords “first”, “next”, “last” and “or”. Here is one possible representation of the plug-in’s syntax:

```
machine m1b
...
events
  event swap1
    $event_order first; next swap2
    then ...
  end

  event swap2
    then ...
  end
end
```

4 Conclusion and Next Steps

Please give us feedback on this proposal via the Deploy WP 9 mailing list[2]. Decisions regarding this project will be documented at [8].

The proposal described here represents a minimal extension for Camille to support extendibility. There is obviously room for further improvement (e.g. providing a default syntax for extensions).

We expect to have a beta version of Camille available by the end of 2010. We are happy to provide active plug-in developers with all information they need at any time. And of course, feedback is very much appreciated.

[2]WP9 mailing list: deploywp9-tooling@jiscmail.ac.uk

References

[1] F. Fritz. *A Semantics-Aware Text Editor for Event-B.* Masters Thesis, Heinrich-Heine-University, Düsseldorf, 2008.

[2] *Records Extension*
`http://wiki.event-b.org/index.php/Records_Extension`.

[3] *Structured Types*
`http://wiki.event-b.org/index.php/Structured_Types`

[4] *Modularisation Plug-in*
`http://wiki.event-b.org/index.php/Modularisation_Plug-in`

[5] A. Iliasov. *A Lecture on modularisation method and plug-in: Introduction and Parking Lot Case Study.* Teaching Resource, Newcastle University, 2010

[6] *Event Model Decomposition*
`http://wiki.event-b.org/index.php/Event_Model_Decomposition`

[7] *Group refinement plug-in*
`http://wiki.event-b.org/index.php/Group_refinement_plug-in`.

[8] *Wiki Page for Documenting Decisions Regarding Extending Camille*
`http://wiki.event-b.org/index.php/Extending_Camille`.