Tech Report: Architectures for an Extensible Text Editor for Rodin

Ingo Weigelt

April 2012

Contents

1 Introduction **3**

2 Problem Analysis **3**

2.1 Extensibility of Rodin . . . 3

2.1.1 Extension Points . . . 4

2.1.2 The Rodin API . . . 5

2.2 Eclipse Modeling Framework (EMF) . . . 5

2.3 EMF-Compare . . . 6

2.4 Plug-in Identification and Versioning . . . 6

2.5 Comments everywhere . . . 7

2.6 Plug-In Interdependencies . . . 7

3 Requirements **7**

3.1 Text Syntax . . . 7

3.2 Comments . . . 8

3.3 Editor . . . 8

3.4 Pretty-Print . . . 9

3.5 Conflicts . . . 9

4 Possible Solutions **9**

4.1 Dynamically generate a Parser at Runtime . . . 9

4.1.1 Architecture . . . 9

4.1.2 Discussion . . . 10

4.2 Blockparser . . . 10

4.2.1 Architecture and Syntax . . . 10

4.2.2 Default Handling . . . 11

4.2.3 Conflicts . . . 11

4.2.4 Conclusion . . . 12

4.3 YAML-Parser . . . 12

4.3.1 Syntax 12
4.3.2 Architecture 14
4.3.3 Conclusion 15
4.4 A Generic Event-B Language 15
4.4.1 Syntax 16
4.4.2 Conflicts 17
4.4.3 Example 17
4.4.4 Discussion 17

5 Conclusion **18**

1 Introduction

Rodin is a platform for Event-B modelling. Its native editor is a structural editor that allows the modification of a tree-like model.

Especially for large models, some users found the default editor inadequate and preferred a text-based editor. Such an editor, called Camille, was created by Fabian Fritz in 2009 [Fri09] and was a huge success.

Nevertheless, Camille does not directly support extensions for Rodin. As more and more extensions are being created, this became a larger issue over the years.

This report analyses how Camille extensibility can be achieved. It analyses a number of different architectures and compares their mutual advantages and disadvantages.

This paper discusses four different approaches. Out of these, we find two particularly promising. The first option would be the implementation of a blockparser (Section 4.2) that provides a useful default implementation for plug-in developers and a pleasing syntax to editor users. The second option is the use of an existing syntax like YAML (Section 4.3). This solution could be extended all the way to the persistence layer of Rodin, thereby simplifying the back-end significantly. But such an effort would require additional resources. Nevertheless, the result would simplify maintenance of the platform significantly in the long run.

2 Problem Analysis

In this section, we look at the current architecture and attempt to describe the problem of an extensible text editor as precisely as possible, taking advantage of what was learnt with the current implementation.

2.1 Extensibility of Rodin

Rodin has been designed to be open and extensible. This requirement was one of the drivers to use Eclipse as the foundation for Rodin.

Eclipse has a plug-in mechanism that allows software to provide extension points. Rodin uses this mechanism to allow plug-in developers to add new elements to the Event-B model at arbitrary positions. An element is a node in the model tree. Elements have attributes, and attributes can be added by plug-ins to existing or new elements. Attributes are typed and can be of one of the following types: boolean, handle (reference to elements), integer, long or string.

Modelling elements can have certain properties that are defined through the API, i.e. by implementing the appropriate interfaces. These include “commented”, “labelled”, “identified”, “have a convergent” and “carry an expression”, “predicate” or “assignment”.

The minimal design goal of an extensible text editor is to offer the same possibilities regarding extensibility as Rodin’s structural editor does. To summarise, the most interesting ones are:

- Defining new modelling elements and attributes, i.e. adding them to the Rodin database
- Defining parent-child relationships between modelling elements
- Defining element-attribute relationships
- Extending the structural editor by new sections and new form elements for attributes

An interesting problem is the issue of pretty-printing, which was a feature of the original structural editor, but is missing from the new one. Pretty-printing is the process of converting modelling elements into a textual form. As the original structural editor is still shipped with Rodin, the pretty-printer extension point still exists (see below).

2.1.1 Extension Points

Using the Eclipse infrastructure, Rodin provides a number of extension points. The following may be relevant for the problem at hand:

Rodin Database — The database and its persistence layer make use of the resource model of Eclipse. To allow plug-ins to store all kind of information in the repository, it has been designed to be generic and independent from Event-B. The use of such a repository was chosen to make extending Event-B easier [ABH+10]. Though any plug-in can store arbitrary data in the repository, two special extension points are defined to extend the Event-B modelling language. These allow developers to add new elements and attributes.

Structural Editor — The structural editor can be extended by new items, i.e. form fields and sections, to edit new attributes and elements added to the database as described above. At the same time these extensions are used to determine which attributes an element consist of and to determine the parent-child relationship between elements.

These are the most relevant extension means regarding this work. Our design goal is to offer as a minimum the same possibilities regarding extensibility for the textual editor.

A number of additional extension points exist that may affect this work and are therefore worth mentioning:

Pretty-Printer — Elements contributed to the database and the editor should probably also be displayed in the pretty-print. Therefore, in the extension point for adding new elements to the editor, developers can also declare a class to render the corresponding parts of the final pretty-print. Note that currently extensions only affect the deprecated structural editor.

Static Checker — Every value entered by a user has to be validated before proof obligations can be generated. "The role of the static checker is to filter all elements from the repository that would cause errors in the proof obligation generator" [ABH+10]. Accordingly, the static checker offers developers extension points to implement the appropriate validation rules for their modelling extensions. Regarding Camille, the errors and warnings generated by the static checker are of special interest: within the textual representation we want to mark the corresponding elements and attach the error or warning message to them. For a detailed explanation of error and warning markers, refer to [Fri09].

PO Generator and Reasoners — The proof obligation generator generates the proof obligations. These depend on the semantics of the contributed modelling elements. Hence, an extension point allows plug-ins to implement their own specific PO generator. Analogously, plug-ins must implement reasoners to contribute proof rules for the new modelling elements. However, this is out of scope for this work because extensions to these subsystems are completely independent from the editor or a textual representation.

Because the Rodin platform is based on Eclipse, plug-ins can also make use of all extension points offered by Eclipse itself. For example, the Event-B explorer, as a part of the core Rodin platform, is based on Eclipse's common navigator framework. Consequently, plug-ins contributing new elements should also use the Eclipse extension points to extend the explorer.

2.1.2 The Rodin API

In addition to extension points, the Rodin platform provides an API to access Event-B models and the persistence layer. This API tries to abstract the internal data representation used by Rodin. The API is not only used by plug-ins to extend the Event-B language, but also to add other functionality to Rodin.

On the one hand some plug-ins access models read only for further processing. For example, the *ProB plug-in* [LB08] reads a model to animate or model check it. On the other hand, some plug-ins use the API to generate Event-B models from other sources. An example is the *statemachine plug-in* [SB08], which brings a graphical diagram editor to model statemachines completely independent from the Rodin editors. From these statemachines, the plug-in can finally generate a corresponding Event-B machine that can then be further processed as any other Event-B model. Beside this, the plug-in also makes use of the database-API to persist the statemachine definitions in the same file as the corresponding machine is saved.

2.2 Eclipse Modeling Framework (EMF)

Camille relies on the Event-B EMF Framework developed by Fabian Fritz, Colin Snook and Alexei Iliasov [SFI10].

> “This framework provides an abstraction of the Rodin database with its API for Event-B models. It strives to offer an API which is comparable to the Rodin API but makes the handling of the models easier.
>
> In addition to a full-featured EMF meta model, the EMF framework comes with support to persist its models into the Rodin database.” [Fri09]

The Event-B EMF framework also reflects Rodin’s extensibility features. That means on the one hand, each EMF-element carries a set of arbitrary attributes — no matter whether they are core attributes or defined by plug-ins. On the other hand, each of those elements contains a list of extension-objects which reflect plug-in-specific extensions. Those extension-objects are again EMF-elements, that means they can again have attributes, child elements and extension-objects.

This design provides the same flexibility, regarding extensibility, for the EMF model as the Rodin Database does.

Camille makes use of the Event-B EMF framework to merge the changes made through the text editor back to the original version connected to the database. Each time the text editor’s input changes and parsing succeeds, a new EMF model instance is generated from the parse result. The model instances generated by Camille differ from the original model generated from the Rodin database. In particular, they can not contain any information that is not reflected in the textual representation.

On the one hand, since the Camille language only supports core Event-B modelling elements, no modelling extensions coming from plug-ins can be generated. On the other hand, the Rodin database stores some data that is usually invisible to the user. An example are the internal identifiers Rodin automatically generates for each modelling element. At the same time, Camille generates and stores some internal information in the EMF model that are not present in the database. These include the source position information of the single elements, for instance.

To synchronise the two different model versions, Camille uses the EMF-Compare Framework [Fou]. This means merging all changes made through the text editor into the database.

2.3 EMF-Compare

The *EMF-Compare* project [Fou] is a part of EMF and adds comparison and merge support to EMF.

> “This tool provides generic support for any kind of metamodel in order to compare and merge models. The objectives of this component are to provide a stable and efficient generic implementation of model comparison and to provide an extensible framework for specific needs.”[1]

Merging models with EMF-Compare is divided into three steps:

Step 1: Matching — A match-engine is applied to two versions of an EMF model. This engine inspects both models to determine which nodes in each version match. While the default match engine already produces acceptable results, it can be further tailored to take more information into account. For instance, the ID of an element can be used to ambiguously decide that one element is the modified version of another. The result of this step is a *match model.*

Step 2: Differencing — A diff-builder uses the match model to build a *difference model* that represents the actual differences between the two models.

Step 3: Merging / Analysis — The difference model from step 2 can be used to visualise the differences or to apply the changes to the models, that means to merge differences from one model into the other and vice versa. In the latter case a merge-engine is applied to merge the relevant changes from one model into the other.

Camille uses this technique to merge all changes made through the text editor into the EMF model that is connected to the Rodin database. This may trigger other open editors to be updated with the new content.

As already described above, the EMF model connected to the database and the model used by Camille contain different information. By merging only the changes back to the original model, all additional data can be retained.

The use of EMF-Compare also allows the user to use different editors concurrently. The same way changes made in the text-editor are merged to the model connected to the database, changes made in the database by other tools are merged into the model connected to Camille. In such cases the model gets pretty-printed to update the textual representation. This solved in part the problem that Camille is not extensible. While plug-in-specific parts of a model can only be edited through the Rodin editors, users were still able to edit the core Event-B parts in Camille.

2.4 Plug-in Identification and Versioning

It is not clear to us if and how much information regarding used plug-ins and their versions is stored or not. As the minimum, we would like to see better documentation, and mechanisms that ensure that the versioning mechanism is used.

In the current implementation, some information is persisted in each component as an attribute of the root element in an attribute called `org.eventb.core.configuration`. This element seems to be only related to the involved builders.

No warning is generated when an Event-B component is opened while the required plug-ins are missing. A warning is generated after the project is cleaned, at least for some plug-ins (the ones present in the configuration attribute: “`Could not get Root Module for configuration ...`”).

[1]Quote from the project’s wiki (http://wiki.eclipse.org/index.php/EMF_Compare)

We think that the information about the required plug-ins is a project-wide property and hence should be handled as such. That means instead of persisting this data along with the individual components, it should be stored in the project settings or at another appropriate place.

More important, we propose to also store the version number of each plug-in. By this, Rodin would be able to generate a warning or error message when the required plug-ins are present, but in an incompatible version.

For the version numbers and ranges we propose to use a scheme like the one used by Eclipse. In this scheme, versions consist of major and minor version-numbers, which allows to infer compatibility between different versions (modelled on the Eclipse version scheme, for instance). If we use such an approach, Rodin will be able to compute compatibility solely by the version numbers.

However, these are both problems that should be solved in the Rodin platform, rather than in a new version of Camille, and ideally before the new Camille is being implemented.

2.5 Comments everywhere

Even when using the structural editor, users felt limited by the inability to put comments in arbitrary places. Once the first version of Camille was released, this problem was amplified, as users attempted to comment in various places. When commenting in the wrong places, they were confronted with error messages, indicating that the comment's position was not allowed.

This problem has been recognised, and an initiative called "comments everywhere" launched. It would be highly desirable that comments everywhere were realised, before a new editor implementation is started. This feature will significantly affect the new implementation of Camille.

2.6 Plug-In Interdependencies

In theory it is possible that plug-ins not only reference elements from the core Event-B model, but also from other plug-ins. However, we are not aware of any plug-ins doing this right now.

We propose to exclude this from the requirements.

3 Requirements

Now that the problem has been analysed in the previous section, we present the requirements that a new Camille implementation shall fulfil.

3.1 Text Syntax

A concrete syntax of Event-B is not officially defined [Abr10, ABH$^+$10]. In addition, Rodin does not need a textual syntax at all, because it is model-based. Of course it needs a syntax for the mathematical expressions. In Rodin, this is handled by its own parser.

There is a textual representation of Event-B in Rodin in the form of the pretty-printer, which was the foundation for Camille's grammar [Fri09].

While Rodin was designed to be open and extensible [ABH$^+$10], neither Camille nor its grammar were designed to be extensible. Therefore, problems arise when plug-ins extend the Event-B language. While they can easily extend the form-based editor, support in Camille is limited. In particular, such extensions are simply not rendered or editable in Camille (although they are maintained in the database). Users have to use the structural editor for editing such elements, while they can use Camille to edit core Event-B (see Section 2.3).

Event-B components can be seen as a simple tree of attributed *modelling elements*. Plug-ins can add any kind of new elements to this tree and they can define arbitrary new attributes for any element (see Section 2.1). That means on the one hand, that we need an Event-B syntax that allows users to insert plug-in-specific additions where needed, and on the other hand, we need a parser that can parse all plug-in-specific input.

During the initial development of Camille, Fabian Fritz [Fri09] defined, amongst others, the following requirements for a text syntax for Event-B models:

- *Usability*: The new text syntax needs to be easy to understand and learn. Readability of the textual representation is an important aspect.
- *Similarity*: Since different text representations have already been used for Event-B models in presentations, documentations and lectures, the text syntax developed strives to be as similar as possible to the most common ways of expressing Event-B models. The current syntax is based on Rodin's pretty-print. In addition, the existing mathematical syntax for formulae was used, since this was already a part of the official description of the Event-B method [Abr10], and users have been creating models with it for years.
- *Completeness*: All information of a core Event-B model which is editable in the graphical editor should be available in text editor, too.

Of course, all these requirements still hold for extensible syntaxes. In addition, we should elaborate on the aspect of similarity with regard to extensions:

We might want to make the additional constraint of similarity between different extensions. Especially the main concepts of element relationships and attribute assignments should be implemented consistently (from the end users point of view) for all plug-ins. This will finally lead to a consistent syntax within and throughout models using different plug-ins, just like all extensions to the structural editor look the same for each plug-in.

However, a too strong similarity to the form based editor might not always be desirable. For instance, in its current implementation, the form elements for new attributes get simply appended to the exiting ones. This sometimes leads to an unnatural visual representation, and accordingly, many plug-in developers prefer a more natural arrangement of elements and attributes in their documentation. Ultimately, the goal should be to provide a pragmatic solution, from a usability point of view.

3.2 Comments

We will require all extensions to comply with the comment conventions used by the core Event-B language. Ideally, we can take "comments everywhere" already in account, as described in Section 2.5. For instance, the current Camille editor accepts two kinds of comments: single line comments starting with a double slash ("//") and multi line comments between "/*" and "*/". Consequently, any corresponding string within a plug-in-specific syntax extension must have the meaning of a comment.

3.3 Editor

Camille supports the usual features of a modern text-editor to increase usability such as syntactic and semantic highlighting, code completion, templates, error markers and so on. For a complete description see Chapter 3.4 in [Fri09].

Of course, all these features should also be available for plug-in-specific extensions. As much as possible should be inferred automatically to take the work out of the plug-in developer's hands. Standard features should be generated automatically. This again increases consistency between different plug-ins.

In addition to the features mentioned above, we may want to clarify which parts of the model text belong to the core Event-B language, and which parts are plug-in-specific. To achieve this, we can make use of common visual hints within the text editor, such as a different background or text colour, markers in the vertical ruler, or the code folding functionality of Eclipse editors.

3.4 Pretty-Print

When a model is opened in Camille for the first time, or after changes in other editors, the textual representation has to be generated from the existing model. Accordingly, we need a mechanism to generate the representation for the plug-in-specific parts.

Depending on the final solution, this can be done completely automatically or has to be implemented by the plug-in developer. In the latter case, each plug-in developer has to be aware that he is responsible to generate a pretty-print that is parsable again.

3.5 Conflicts

Whenever multiple plug-ins are installed at the same time, especially when they have been implemented by different developers, we might get conflicts. For instance, two different plug-ins could declare the same keyword or element name with different meanings.

The ideal solution should eliminate the risk of conflicts altogether. This might not always be possible. At least this risk should be minimised and, more important, each conflict should be resolvable by the end-user (as easily as possible).

Independent from the solution, the most important requirement is that conflicts **must** be detected. An unparsable input is acceptable, but an ambiguous model is **not**.

4 Possible Solutions

In this chapter, we present a number of possible solutions for a new implementation of Camille, which realises the requirements discussed in Section 3.

4.1 Dynamically generate a Parser at Runtime

In this approach, Rodin generates a specialised parser for an extended Event-B language, based on the installed (or used) plug-ins. Every time a new plug-in is added or removed, the parser has to be regenerated. While this approach is extremely flexible and provides opportunities for reuse with respect to the parser rules, the drawbacks dominate. Those include fragility and risk of interference between plug-ins. Also, the implementation and maintenance will be complex. In addition, this solution requires more effort for plug-in developers, compared to others.

4.1.1 Architecture

The foundation is provided by a grammar for the core Event-B language that also functions as a skeleton-grammar, allowing grammar-extensions to be inserted at specific points.

Accordingly, Camille provides extension points for plug-ins to register their language contributions in form of grammar-artefacts. In addition, a plug-in developer would have to write the code to connect their grammar to the data model.

Each time a plug-in is installed, Camille collects all artefacts and inserts them into the skeleton-grammar. From the resulting grammar file, a parser-generator can finally build a specialised parser that supports exactly the extended Event-B language supported by the combination of installed plug-ins.

4.1.2 Discussion

The solution presented will be difficult to implement right, and once implemented, it will create a significant burden for the plug-in developers.

Plug-in developers have to write grammar artefacts, and they have to write them for the parser-generator we choose (which they are unlikely to be familiar with). They will also have to implement more functionality, like an AST visitor to process the grammar.

An implementation must generate a single grammar that is assembled by contributions from individual developers. Since each of them develops an independent plug-in, coordination is unlikely. This means that conflicts between plug-ins are probable. This distributed approach is very error-prone. The grammar that will finally be assembled could be faulty, that means not accepted by the parser generator or, even worse, it will be accepted, but the generated parser does not work as expected. All of these conflicts will appear at run-time. Users may not even know which plug-in is the source of a problem, and plug-in developers may not even be aware of users having problems.

The only promising solution to this problem is that every possible combination of plug-ins is heavily tested. This would have to be done either by each plug-in designer before he publishes a new extension, or by a single responsible institution. These are both solutions that contradict the idea of an open extensible platform.

4.2 Blockparser

In this approach, Camille consists of an outer parser that breaks the model code into blocks (therefore a *blockparser* approach). The blocks are then processed by multiple inner parsers. One special inner parser processes the core Event-B language. The other blocks are dispatched to parsers provided by their respective plug-ins.

The special inner parser is part of the Camille core and can most probably reuse the existing one. At least he can be written such that the grammar does not differ from the one employed in the current Camille implementation. Thus, an Event-B model using no extensions would not differ from a model in the current Camille implementation.

4.2.1 Architecture and Syntax

To implement this blockparser, 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.

An easy way to detect the blocks is to guard them by unique character-sequences that can not appear elsewhere in an Event-B model. For example, we can open a block with `$$` and close it with `$$END`, as illustrated in Figure 1. Then the blockparser can filter out all extensions and, after the core part has been processed, dispatch them to the corresponding plug-ins along with the needed context information such as the parent element, etc.

To identify the corresponding plug-in for each block, we can use the first token after the opening guard as a registered keyword. This keyword is bound to a specific syntax-extension

```
context ctx0

sets  E, F

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

...
```

Figure 1: An Example of a Block Parser Element

that is part of a plug-in. The raw input text of the block is finally passed to the responsible plug-in, which can handle it as it sees fit.

Some plug-ins only extend the Event-B syntax by relatively short expressions, such as new simple attributes added to existing elements. For instance, the *Qualitative Probability* plug-in [HH07] adds one new boolean attribute (*probabilistic*) to events. In this example, only one single word has to be added to an event declaration.

To reduce the syntactical clutter in such cases, we could offer an additional short version of blocks. Those could, for example, be opened by a single `$` and consist of only one token or a parenthesised expression. See Figure 2 for an example.

Short and long blocks are handled by the blockparser, transparently to the plug-ins.

```
convergent $probabilistic event evt1 refines ...
```
or
```
event evt1 $(group Pay) refines...
```

Figure 2: Examples for short blocks.

4.2.2 Default Handling

Many plug-ins contribute quite simple extensions to the structural editor, and accordingly to the Event-B language. For instance, as described in Section 4.2.1, the Qualitative Probability plug-in adds one attribute to events. In the structural editor the value can be chosen from a dropdown list. The possible value strings presented in this list are defined by the plug-in implementation. Camille can use this implementation to provide a default implementation for the text editor. That means we can automatically generate a default parser that accepts exactly the allowed values and sets the attribute's value in the model. At the same time, this can also be used to provide editor-features like code completion, quick-fix, and so on.

All in all this means the Block Parser solution will "work out of the box" for many plug-ins without plug-in developers having to implement anything special.

4.2.3 Conflicts

Since language extensions are guarded by a special character-sequence, the outer parser can reliably distinguish each block from the others and the core language.

The only conflict that can occur in this approach is that two plug-ins register the same keyword. A possible solution is to use *import statements* that define the concrete keyword used in a file (or project) for each imported language-extension. For example, if `org.EventB.records.syntax` is the unique id of the syntax-extension contributed by the *Records plug-in* [EB06], then we can add the statement

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

at the beginning of the model in order to bind the keyword `RECORD` to the *records-extension*.

In case of conflicting keywords, the end-user is able to resolve the error by simply choosing other keywords and replacing the appropriate occurrences in the model. Instead of manually replacing the keywords, one could also simply pretty-print the model again. Since the pretty-printer should use the new, renamed, keywords it will produce a conflict free textual representation automatically.

4.2.4 Conclusion

The blockparser approach promises a very robust solution: since a language extension is isolated by the guards, it cannot interfere with the core language or other extensions. The potential for conflicts to occur is minimised to the risk of conflicting keywords. However, such conflicts can be resolved automatically.

While the core Event-B language remains the same, plug-in developers are free to design their extensions as they like. For this, they may have to implement their own parsers, but are free to use the parser-generator or technique of their choice. However, at least for simple or short extensions, a set of default implementations should be adequate. In such cases the extra workload for plug-in developers is implementing only a few simple interfaces.

4.3 YAML-Parser

As described is Section 3.1, Event-B models consist of a tree of attributed modelling elements. The tree is persisted as XML in the database. Rodin's current XML format is not suitable as a human readable text-format, and especially not for editing (even though users have been known to edit the XML directly).

One solution to the extensibility problem is to use the very same language for persistence and as an editable text representation, as this is the case for most programming languages.

Since XML was not designed for human use in the first place, we discuss this approach with YAML as an example for an alternative persistence format and a new Event-B language. The most important design goal for YAML was to be "easily readable by humans" [BKEdN].

> YAML^{TM}[...] is a human-friendly, cross language, Unicode based data serialization language designed around the common native data types of agile programming languages. It is broadly useful for programming needs ranging from configuration files to Internet messaging to object persistence to data auditing. [BKEdN]

4.3.1 Syntax

YAML uses indentation to represent structure hierarchy and to denote scope. The most important entities are sequences and mappings. For sequences, each entry is denoted by a dash and space ("- "). Mappings use a colon and a space (": ") to mark "key: value" pairs. The value can be of any type, i.e. it can be a scalar such as a string or a number, or again a sequence or mapping.

With this features we can already express an Event-B model in YAML:

```
machine: m1           # Comments begin with a ‘#’ in YAML documents
refines: m0

sees: [ctx0, ctx1]    # Alternative representation of lists

variables:
    - x
    - y

invariants:
    - inv1: x ∈ ℕ
    - inv2: y ∈ ℕ
      theorem: Yes    # Boolean values can be written as ‘Yes’ and ‘No’

events:
    - event: INITIALISATION
      then :
        - act1: x:∈ℕ

    - event: evt1
      ...
```

Plug-in-specific extensions can now be inserted at nearly every point within this definition. Analogous to the blockparser approach, we can mark plug-in-specific YAML-elements. This can help to keep the parser simple and to avoid conflicts between new keys and user-generated labels. For instance, we can define that all keys or elements (in the YAML sense) coming from a plug-in have to begin with a $-sign. Accordingly, the use of a $ as the first character would be forbidden anywhere else.

With this, an extended Event-B machine with, for example, the Probabilistic plug-in, could be expressed as following:

```
machine: m1

sees: c0  # we may also want to accept a single element
          # instead of a list with one element, i.e.: [c0]

variables: [x, y, z]

$bound: const0

invariants:
    ...

events:
    - event: INITIALISATION
      ...

    - event: EVT1
        convergence: convergent
        $probabilistic: Yes
      ...
```

4.3.2 Architecture

One option would be a parser that maps to the existing database back-end. A better solution, however, would be to completely re-implement Rodin's persistence layer, based on a human readable language like YAML. Doing so has a number of advantages: The EMF-Compare merge layer becomes obsolete. An extensible editor comes "for free", and versioning with standard tools like subversion or git works as expected (which is not the case with the current implementation).

Since YAML is a widely used general purpose language, there already exist many YAML-implementations we can build upon. For instance, SnakeYAML [sna] is a YAML parser and emitter for Java. A first evaluation confirmed that SnakeYAML would be suitable as a parser for the proposed Event-B language. By default, SnakeYAML generates an AST that mainly consists of Java maps and lists. From this AST, we can easily generate the internal Rodin representation by using its database API.

Analysing the current XML-serialisation reveals two new problems, if we use the persistence format also as the textual representation. In the current implementation, some information gets persisted that is usually invisible or irrelevant for the end-user. Accordingly, this information would have no meaning to the user in the textual representation. For instance, Rodin assigns an internal name that is used by other builders and tools to each element. To avoid cluttering the model representation, we can make use of YAMLs "repeated nodes" feature (see Figure 3). Each modelling element can be identified by an *anchor* in YAML. On the one hand, we can directly use this anchor as the internal name to uniquely identify the element. On the other hand, we can thereby refer to a certain element in other parts of the document. That means, for instance, that we can attach additional information in a dedicated section of a text document. This section could even be hidden or made read-only by the text editor.

An even better solution is to move all data that is not meant to be directly visible or editable in the text editor, to a separate file. In the current implementation, when plug-ins use Rodin's persistence API, all data get stored in the same XML file, together with the model.

```
- event: &INITIALISATION
        # define an anchor labelled ‘INITIALISATION’ for the event by
        # using the prefix ‘&’. At the same time, this is the event name.
  then: x:∈ℕ
    ...

- event: *INITIALISATION  # refer to the event labelled as ‘INITIALISATION’
```

Figure 3: Repeated Nodes in YAML

The other problem is that Rodin uses full qualified names for elements and attributes (e.g. `org.eventb.core.label`) in the current XML serialisation. Since these qualified names are derived from the unique plug-in ids, their use guarantees conflict freeness. In general, if we use the short element names, this is the same problem of conflicting keywords as already discussed in Section 4.2.3, and the same solution can be applied to the YAML-syntax: import statements can link the human readable keywords to the internally used fully qualified names.

YAML already provides such a feature called *tags*. With tags, one can denote the type of a YAML node within the document. For instance, we can prefix an event with `!!org.eventb.Event` to directly instantiate the Java `class org.eventb.Event`, instead of a generic YAML node. The parser can then be configured to use a shortcut, e.g. `event`, for this tag. From then on, `!event` can simply be used to denote a new event node in the document.

4.3.3 Conclusion

All in all, YAML has the potential to dramatically simplify Rodin's architecture — but only if it is used all the way down to the persistence layer of the database. This is a task that goes far beyond the scope of this project, and additional resources must be freed, should this avenue be followed.

While it is possible to use YAML just for the editor (and continue to use the existing back-end), then this solution offers few advantages over a blockparser solution, as described in Section 4.2, or a generic Event-B language, as described in Section 4.4.

4.4 A Generic Event-B Language

Basically, this approach is a combination of the concepts described in the blockparser (Section 4.2) and YAML-Parser (Section 4.3) solutions: Instead of using special symbols for guarding blocks, the core Event-B grammar is extended to allow plug-in-specific extensions at defined positions. These extensions are limited to a fixed built-in grammar, which enables the user to define attributes and elements in a generic syntax. An example for such a generic syntax is the YAML syntax described in Section 4.3.

In contrast to the YAML-solution, the generic language extensions provides less flexibility. The idea behind a generic language extension is the fact that all existing plug-ins only use the extension points described in Section 2.1.1 to extend Event-B components. That means, they only add new attributes and elements to the database and appropriate input fields to the structural editor. For such well defined extensions, it is possible to provide generic language constructs, that allow to express the needed additions. At the same time, we can keep a specialised language for the core Event-B parts, promising less syntactical clutter, and hence a better user experience.

In summary, a generic Event-B language needs to provide the following features:

- Add new plug-in-specific elements to the model.
- Add new plug-in-specific attributes to elements (both existing elements and plug-in-specific elements).
- Attributes are typed, i.e. their values can be one of the types boolean, number (integer or long), string or handle (see Section 2.1).
- Elements can have special attributes that act as labels, identifiers or a convergent, and string attributes can be marked as an expression, predicate or assignment. Therefore, any behaviour that built-in elements and attributes express, can be replicated in plug-in-specific elements and attributes.

In principle, we will have three options to implement a generic Event-B language: We can (1) either create a completely new language; we (2) keep Camille's existing language as a core language and extend it to support the features we just listed; or (3) we adapt and extend Camille's existing language to support the new features.

One example for a complete redesign has already been given with the YAML parser in Section 4.3 and therefore we omit (1) from this discussion. Obviously, we can use a language like YAML without replacing Rodin's persistence layer but keep using the EMF and EMF-Compare solution.

The second option would allow users to keep their formatted textual representations. As we already argued before that this is not a high-priority requirement. Option (3) is more attractive, as it is less constraining to the developer and to the resulting syntax, while still leaving the user with a familiar syntax. We will discuss option (2) in the following, but are aware that (3) is acceptable as well, should some additional features require changes to the existing syntax.

In the rest of this section, we will discuss the requirements for a generic extension to Camille's current Event-B language and outline one possible solution. That means, we can keep the current syntax as the core Event-B language while providing a generic solution for plug-in extensions.

The possible solutions to (2) are restricted by the current grammar definition. All extensions must seamlessly integrate into the existing syntax, without any conflicts or ambiguities. Therefore it is sensible to introduce some constraints on the syntax that can ease the final implementation. One such constraint, for instance, is to restrict names and identifiers. Within the structural editor one can choose an arbitrary string as a name. This string can even contain whitespace, special symbols like colons, at-signs, etc. While this is no problem within a form-based editor, where each value is entered in its own input field, it is very hard to implement such a behaviour for a parsable language. Accordingly, some restrictions already apply for the current Camille implementation. For instance, using a space in an invariant's label is accepted by the structural editor, but yields a syntax error in Camille. For more details refer to [Fri09]. One major additional restriction may be to accept at most one attribute definition per line, as described below.

4.4.1 Syntax

Taking the above into account, we propose the following conceptual features that are visualised in a mock-up in Section 4.4.3.

As mentioned above, the language must support defining and adding new attributes in a generic way, without any knowledge about the used plug-ins:

- Attributes can only be written as key-value pairs, for example as *name*:=*value*.
- An exception can be defined for some built-in attributes like labels. Since each Element can have at most one label we can reuse Camille's convention to begin labels with an @-sign. In other word *@value* is an alternative expression for *label:=value*.
- Attribute names and labels should be restricted as described above. Basically this means to restrict the allowed characters as identifiers are restricted in nearly every programming language. Especially all kind of whitespace characters are forbidden.
- Attribute values can consist of a mathematical formula. As already argued in [Fri09] we want to reuse the existing mathematical parser and we do not want to replicate it's functionality within Camille's parser. We can significantly reduce the complexity if we require mathematical formulae to be terminated by a line-end[2]. By this, the parser does not need to be aware of the inner structure of formulae but can simply scan for the next line-break that terminates the formula. Note that a mathematical formula should be allowed to span multiple lines. Many users already use line breaks and indentation to write formulae in a structured and readable way, even within the form fields of the structural editor. Accordingly we see this as a highly desired feature.

  Equally, the parser can be further simplified by a restriction that only one attribute definition per line is allowed. However, such a rule — or otherwise a more complex grammar — may only be required for more or less rare combinations of attribute definitions where arbitrary strings are allowed as an attribute value.

Further, the language must support defining and adding new elements:

- Child-elements can only be inserted at defined positions, since the appropriate rules have to be inserted into the grammar definition.

[2]In this sense comments are treated as a line break, thus allowed to be inserted before the actual line break.

- Parent-child relationships can, for example, be defined by using braces or indentation.
- Braces, or the different indentation levels, also define the scope of a block. This is conceptually another type of delimiter, as used in the blockparser approach in Section 4.2.
- Element names can simply be placed in front of the corresponding block, i.e. written as: **keyword :** { *body* ...}.

4.4.2 Conflicts

As in the other solutions, we have to face the problem of conflicting element or attribute names. However, the same solution as described in Section 4.2.3 can be applied. In this case, by using import statements, the user is able to rename attribute and element names instead of registered keywords.

4.4.3 Example

Defining a plug-in specific attribute: In this example, the attribute *probability* from the *Qualitative Probability* plug-in [HH07] is defined for the event *EVT1*.

```
events
  convergent event EVT1
  probability := probabilistic
  where
    @guard1 ...
  then
    @act1 ...
  ...
```

Defining a plug-in specific element: This example depicts a context using the *Records plug-in* [EB06]. In this context the complete *record_declaration* section is plug-in specific.

```
context ctx0

sets E F

record_declaration : {
  record : {
    @rec1
    closed := closed
    fields : {
      field : { @e type := E }
      field : @f type := F  /* no braces means the element definition
                               reaches to the end of the line (or a
                               comment) */
    }
  }
}
...
```

4.4.4 Discussion

The main drawback of this approach is the fact that it is essentially a simplified implementation of the blockparser described in Section 4.2. It attempts to omit special guard-characters, at the expense of less expressiveness. Therefore, a plug-in developer has no means to influence the language used by end-users to work with his plug-in through Camille.

On the other hand, this means that all plug-in specific extensions will be handled equally by the text editor. For Rodin users, this results in a consistent user experience. Concepts used by the structural editor are equally transferred to Camille. The main advantage of this approach is that all required information can be extracted from the already existing Rodin extension points. Plug-in developers do not need to provide any additional implementations to make their plug-in usable with Camille. Also, only one single parser would be employed, promising a robust implementation.

However an equivalent solution can be achieved by using the blockparser together with a default implementation that follows the proposals from Section 4.4.1. This would give us all the advantages mentioned above, while still providing the opportunity to adapt to future, yet unknown, requirements.

5 Conclusion

The main criteria for selecting a solution are:

- A manageable workload for the Camille developer
- Maintainable for future Camille and/or Rodin developers
- Ease of use for the plug-in developer
- Reuse of existing technology, as much as possible
- Flexibility in supporting the current and future needs of plug-ins
- Robustness with respect to interoperability of plug-ins

These criteria already rule out the dynamically generated parser (Section 4.1), as it is a lot of work, reuses little technology, and lacks in terms of maintainability and robustness (but it offers the most flexibility).

Along similar lines, we discourage the generic Event-B language (Section 4.4). It offers less flexibility than the blockparser approach, while being similar in all other respects to it.

We consider the two remaining solutions, blockparser (Section 4.2) and YAML (Section 4.3), as promising.

The YAML solution (Section 4.3) has the potential to drastically improve Rodin's architecture, but only if this solution is extended to Rodin's back-end. This vastly exceeds the scope of this work and would require additional resources. However, the author of this paper is willing to implement this solution, should the project leaders decide to support this approach.

If the YAML solution is not extended to Rodin's back-end, then it provides few advantages over the blockparser solution (Section 4.2), which would be our recommendation in that case.

References

[ABH^{+}10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin, *Rodin: An Open Toolset for Modelling and Reasoning in Event-B*, International Journal on Software Tools for Technology Transfer (STTT) **12** (2010), no. 6, 447–466.

[Abr10] Jean-Raymond Abrial, *Modeling in Event-B*, Cambridge University Press, 2010.

[BKEdN] Oren Ben-Kiki, Clark Evans, and Ingy döt Net, *Official YAML Specification: "YAML Ain't Markup Language (YAML^{TM}) Version 1.2", 3rd Edition, Patched at 2009-10-01*, Website: http://www.yaml.org/spec/1.2/spec.html.

[EB06] Neil Evans and Michael Butler, *A Proposal for Records in Event-B*, Lecture Notes in Computer Science **4085/2006** (2006), 221–235.

[Fou] Eclipse Foundation, *EMF Compare Framework*, Website: http://www.eclipse.org/emf/compare/.

[Fri09] Fabian Fritz, *A semantics-aware text editor for event-b*, Master's thesis, Heinrich Heine Universität Düsseldorf, 2009.

[HH07] S. Hallerstede and T. Hoang, *Qualitative probabilistic modelling in Event-B*, Integrated Formal Methods, Springer, 2007, pp. 293–312.

[LB08] M. Leuschel and M. Butler, *ProB: an automated analysis toolset for the B method*, International Journal on Software Tools for Technology Transfer (STTT) **10** (2008), no. 2, 185–203.

[SB08] C. Snook and M. Butler, *UML-B and Event-B: an integration of languages and tools*, The IASTED International Conference on Software Engineering - SE2008, innsbruck, Austria, 14 - 12 Feb 2008 (2008).

[SFI10] C. Snook, F. Fritz, and A. Illisaov, *An EMF framework for Event-B*, Workshop on Tool Building in Formal Methods - ABZ Conference, Orford, Quebec, Canada (2010).

[sna] *snakeyaml, YAML parser and emitter for Java*, Website: http://snakeyaml.org.