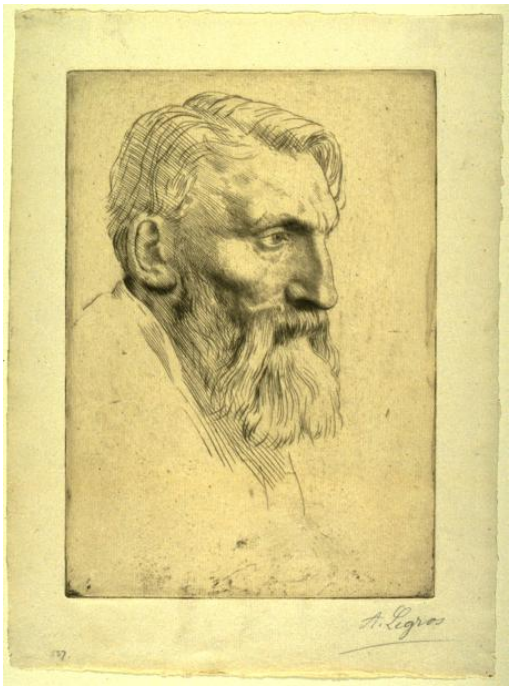
Information Society Technologies

Project IST-511599

RODIN

"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable 3.1

Final Decisions

C. Métayer (ClearSy) S. Hallerstede, F. Mehta, L. Voisin (ETH Zürich)

Public Document

28th February 2005

http://rodin.cs.ncl.ac.uk

Contents

1 Introduction

This document is the first deliverable of Work package 3: Open Tool Kernel. It complements the Description of Work [DoW] of the RODIN project by stating how the issues, that were left open in that document, are solved.

We first examine which Modelling Process we will support, then for each task that has open issues, we justify and state our decisions. Finally, a conclusion sums up all decisions taken.

2 Modelling Process

In this document, the term *modelling process* means the activity of modelling a system using a the open kernel tool. The output of this activity is a set of event-B models which are fully proved.

We first present the *Sequential Modelling Process* which was envisioned in the document Description of Work [DoW]. Then, we introduce the *Reactive Modelling Process*. Finally, we state which Modelling Process we want the RODIN Platform to support.

2.1 Sequential Modelling Process

The modelling process used with Atelier B (the classical tool supporting the genuine B Method, now termed classical B) can be sketched as follows:

1. The user first writes its model in a source text file.
2. Then, the user runs several tools, one after the other, to produce proof obligations.
3. Finally, the user attempts to discharge these proof obligations using automated and interactive provers.

The main advantage of this process is that it is very simple, and therefore straightforward to implement and use. Tools can mostly be developed independently. All that is needed to support this process is

- a *Connection Language* allowing tools to share common information;
- a *Project Manager* that links up the tools, ensuring they are run one after the other and in the correct order;
- a *Platform* that gathers all parts behind a user interface.

However, the main drawback of this process is that it is not adapted to the way people actually work. In the process described above, if all proof obligations can be discharged, then the model is correct and the work is finished. But most of the time, while trying to discharge proof obligations, the user will find out that the model is incorrect in some respect. The user subsequently fixes the model and starts the whole process from the beginning. This is very disruptive and, even for a slight change of the model, the user will need to wait for the tools to process repeatedly the whole model. Typically, when designing an average model, the initial writing of the model will represent 20 % of user time, while fixing it and discharging all proof obligations will represent 80 %. Moreover, that latter phase is disrupted quite often by several launches of tools to generate new proof obligations when the model has been changed.

And the situation is even worse due to the use of refinement. In fact, the user seldom works on a single model, but rather on a refinement chain. For instance, Figure 1 shows a development comprising four models that refine each other. Model `M1` is the top-level model. It is refined by model `M2`, which is itself refined by model `M3`, and so on.

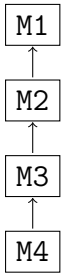


Figure 1: Example of a refinement chain.

In this example, it is quite possible that, when trying to discharge the proof obligations for model `M4`, the user finds out that there is an error in model `M2` and fixes it. The consequence of the changes in model `M2` is that all proof obligations of models `M2`, `M3` and `M4` have to be generated and discharged again. In the traditional modelling process, this entails that these three models need to be processed entirely again, which takes a lot of time.

2.2 Reactive Modelling Process

Fortunately, there exists another approach for designing the modelling process which is more user-centered and therefore more efficient.

The key idea is that the user's goal is well-known: it is to discharge all proof obligations generated for models. Moreover, proof obligations can usually be generated independently of each other. Hence, there is no reason why the different tools should wait for the user to have finished editing. The tools could start generating proof obligations while the user is still editing a model. In the same vein, as soon as a proof obligation has been generated, an automatic prover could try to discharge it. The main difference with the previous sequential modelling process is that, now, the tools are not launched on the user's demand anymore, but run permanently in the background, reacting to changes entered by the user.

The drawback of this approach is that it is more difficult to implement, as one needs to coordinate carefully the tools and make them react to user actions. The control flow is turned inside out in comparison with the sequential process.

Advantages are numerous:

- The tools now work in a differential way: they compute only the impact of changes entered by the user, rather than processing again and again the whole model. Hence, the tools work much faster and the user spends less time waiting for them to finish their work. These is even more relevant when working on a refinement chain as exemplified in previous section.

- By having tools working greedily in the background, we can use all the CPU time which is otherwise wasted waiting for the user to do something.

2.3 Decision

In view of the previous discussion, we have decided to implement the reactive modelling process for RODIN. As a consequence, some of the tasks of the document Description of Work [DoW] must be adapted to the new process:

- Task 3.5: The Connection Language is replaced by a Database Manager.

- Task 3.6: The Project Manager remains but its internals are very different: instead of launching tools on user demand, the Project Manager will organize the reaction of tools to the changes entered by the user.

- Task 3.7: The Open Platform also remains, but it shall satisfy an additional requirement: the Open Platform must provide support for implementing the reactive modelling process.

In the sequel, we shall come back to each task and state decisions that have been taken.

3 Task 3.5: Database Manager

In this section, we first describe the two approaches that have been studied for solving that task. For each approach, we describe the pros and cons. Finally, we state which approach will be followed and some additional implementation decisions.

3.1 Connection Language

As stated in the Description of Work [DoW]:

> The *Connection Language* is the "universal" means by which the syntactic structure of Event-B texts can be exported to the plug-in tools. It will be based on XML and the generation of this syntactic structure will be performed by the low level basic tools (after type checking).

This approach fitted very well with the Sequential Modelling Process, where tools were working independently of each other and were taking entire models as input. However in the Reactive Modelling Process advocated above, this approach doesn't work anymore, as the input of tools is not the syntactic structure of models, but rather the changes on the model that the user has entered.

3.2 Database Manager

As a consequence, we need a different means for feeding tools with their input. Moreover, we also want that this input remains easy to compute, as it will be computed quite often. In this respect, having the user entering a model in a flat file is not very efficient. It would be better if the way data are stored would allow us to compute efficiently the changes entered by the user.

To this end, the use of a database approach seems to be what we need. Then, a database manager will know precisely what the changes to the model are and tools will be fed directly with these changes. There is no need for a special computation trying to figure out what changes the user has made to his model by comparing two entire models.

Another advantage of this database approach is that, when new plug-ins need additional input from the user, these new inputs can also be stored in the database. One just needs to extend the database schema (and the user interface). This is much easier to do compared to with extending the syntax of models for allowing these additional inputs. Thus, the database approach also brings us more versatility and reinforces the openness of the platform.

Moreover, in the Connection Language approach, one would have needed two languages: one for describing the model and another one for describing the proof obligations generated by the tools. In the database approach, we can store both the model and the proof obligations in the same database, thus providing a unified interface between the data being manipulated and the tools.

3.3 Decisions

In view of the previous discussion, we have decided to implement the database approach for RODIN. This choice creates another issue, which is to define how the database will be implemented. Basically, there are two ways to implement it:

- Use of a classical database management system (DBMS) off the shelf, then define our schema on top of it. Tools would then use standard means (such as JDBC) to interact with the database.

- Use of a more pragmatic approach, retaining the database concept, but using an ad-hoc implementation with a small memory footprint and tailored to our needs.

The advantage of the first approach is that it allows for reuse of software and minimizes a priori development time. Its main drawback is that it is quite heavy compared to our needs. For instance, DBMS provide a lot of features such as concurrent access, transactions, etc. that we don't need in our special case. Another cause of heaviness, is that DBMS are built for handling efficiently vast amounts of data (typically millions of records), while, in our case, we have relatively small amounts (in the order of tens of thousands, at most). So, a DBMS-based approach seems to be a big hammer to pound a small nail.

On the contrary, in an ad-hoc approach, we have to develop only the functionality that we need. Moreover, as our needs are quite small, this implementation will be lightweight. Basically, the database contents can be realized using a set of Java objects in memory. We can then use the

1 edit model
2 save document
3 run static checker (in case of error stop with message)
4 run proof obligation generator
5 run automatic prover
6 run interactive prover

Figure 2: Typical sequence of batch commands

serialization mechanism provided by Java for making the data persistent in files on the disk.

Furthermore, using an ad-hoc approach allows us to map parts of the database to files, so that we can have a direct correspondence between a model (or a context) in the database and a file on disk. This scheme allows us to reuse the basic platform mechanisms for configuration management and team collaboration support. It also allows us to implement easily project archiving, where an archive groups a set of models and contexts that together form a formal development. Archives can then be used for exchanging developments. These two aspects are very important for allowing formal development in teams.

Thus, we have chosen to take the ad-hoc approach for implementing the Database Manager.

4 Task 3.6: Project Manager

The architecture of the tool has been changed to achieve improved responsiveness by operating on parts of the database rather than on full source texts of modelling documents. The Project Manager will not exist anymore as an independent component but will be tightly integrated into the tool in the form of builders that process changes differentially. The Project Manager would have worked in batch mode where the user would issue a sequence of commands (perhaps in a make file) in order to analyze a modelling document as shown in Figure 2. We refer to this user activity as *edit-prove cycle*.

Terminology: We use the term *modelling document* to refer to entities the user perceives as a document. These can be *models* or *contexts* which are part of the EventB notation, or corresponding concepts contributed by others. We use the term *modelling element* to refer to entities that are contained in a modelling document.

In this scenario the user spends a lot of time waiting for the tool to either

produce proof obligations to be discharged interactively or error messages. Especially when creating new models, errors in a model are often found during interactive proof sessions. As the model grows in size the user spends more and more time waiting during each edit-prove cycle. If the tool would work more differentially only checking those parts of the model that have changed, the main modelling activity consisting of edit-prove cycles could be carried out much faster. The sequence of commands of Figure 2 would only be executed for the parts that have changed.

We have changed the architecture of the tool accordingly. The tool follows user changes and only performs necessary work on those parts of the database that may be affected by a user change. Furthermore, the user will not have to start any of the components like static checker or proof obligation generator or automatic prover by issuing some extra command. This will happen in the background when changes to the database are committed by the user. The different components are arranged in layers that are matched by corresponding layers of the elements stored in the database. Different parts of the database may belong to different layers, some may contain typing errors while others that do not depend on erroneous element may have been proved correct.

In the list of actions mentioned in Figure 2 two are of real interest to the user: edit model and run interactive prover. In the new architecture the user will only see these.

We can associate modelling elements with different layers according to whether they have passed the check of static properties by the static checker, whether their proof obligations have been generated by the proof obligation generator, or whether corresponding proof obligations have been discharged by the automatic or interactive prover.

In the original Project Manager approach all modelling elements of a document that has been edited would traverse all layers during each edit-prove cycle. In the builder approach, modelling elements that are not affected by some change are left untouched and do not need to be reanalyzed during each cycle. In this approach, the analysis will happen more often but usually only on a small part of the database of modelling elements. Different modelling elements may belong to different layers. This can be used to give much better feedback to the user already while editing a modelling document, and it will improve general performance.

5 Task 3.7: Open Platform

We first describe the target platform and justify the choice of Eclipse. Then, we show that this choice doesn't restrict the way plug-ins can be written. Finally, we present the result of a small study for choosing our implementation programming language.

5.1 Eclipse Platform

Eclipse is an open platform consisting of a consortium of information technology companies, including IBM, Red Hat, SuSE, Borland, HP, Telelogic, Oracle, SAP, among its about 50 members. A non-profit organization called Eclipse Foundation manages work around the Eclipse Platform, and hosts Open Source projects around the platform.

Eclipse hosts at the moment 4 major Open Source projects in which plug-ins for graphical editors, UML2, Java/C/C++, among others are being developed. Plug-ins for Java development and C/C++ development exist that can be used for the development of Eclipse plug-ins.

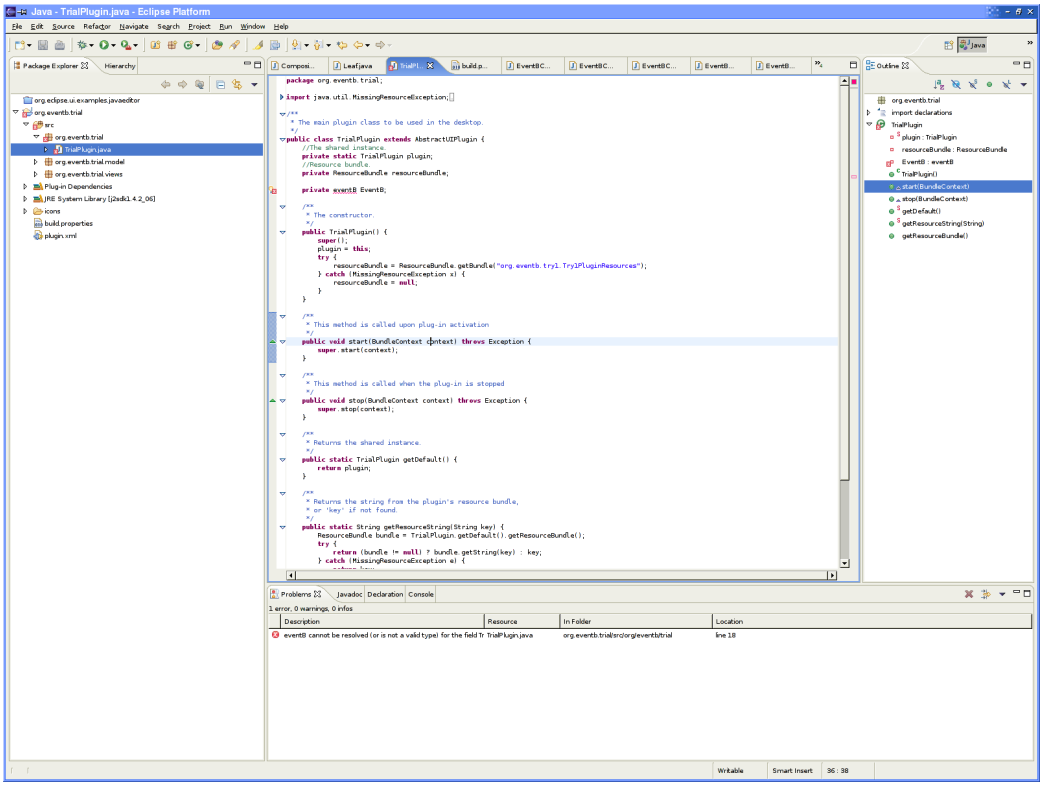


Figure 3: Eclipse platform

The Eclipse Platform (see Figure 3) is a production quality environment and plug-in architecture. Figure 3 shows the typical appearance of the platform that will also be used by the RODIN Platform. On the left hand side is the Resource View that shows documents and other resources used in a development. The big window in the middle shows an Editor for a document. On the right hand side is the Outline View that is used for quick reference and navigation in the document. On the bottom is the Task View that usually shows error messages created by builders that run in the background when a document is saved. All these views are connected to each other. For instance, a mouse click on an error message jumps to the position in the Editor where the error was found, and a mouse click in the Outline View jumps to the corresponding position in the Editor. The infrastructure to implement these features is readily available in Eclipse and will facilitate the implementation of corresponding features in the RODIN Platform.

The Eclipse framework provides guidelines for plug-in development, user interface design, and some general architectural guidelines that allow better interoperability between plug-ins developed in different places. This frees resources for development of the kernel tools of the RODIN platform, because these things would have to be provided in order to achieve an open platform.

The Eclipse Platform is implemented mostly in Java. The RODIN Platform will inherit this property profiting from the same benefits: It is not dependent on a particular operating system, hence, easily portable, and the RODIN Platform and all plug-ins can make use of Java extensive class libraries.

The large industrial support offers a good basis for dissemination of the RODIN Platform. This is also true for the publication of results concerning the tool because the audience already exists.

We expect that the Eclipse platform will exist for many years because of the large existing commercial and non-commercial community that supports it. This argument is enforced by a recent move of the Eclipse Foundation to turn Eclipse into an application platform not restricted any more just on programming related tasks.

Other alternative platforms that were considered are:

- *NetBeans* is a platform for developing Integrated Development Environments (IDE). Unfortunately, this platform is quite specialized for Web and Java IDE development and seems much more difficult to reuse than Eclipse. In particular, it provides much less support for developing new plugins.

- *Coral* is a metamodel-independent software platform to create, edit and transform new models and metamodels. It is being developed in

the context of a research project at CREST, the Centre for Reliable Software Technology at Åbo Akademi University. The main drawbacks of this platform are twofold: firstly, it lacks the notion of refinement and proofs; secondly, it lacks portability and stability. Currently, this platform is available natively only on Linux, not on Windows or Mac OS X.

5.2 Plugins

There are two major ways of providing plug-ins for the Eclipse platform. Firstly, newly written or existing Java code can be equipped with plug-in interfaces as required in the Eclipse architecture. Secondly, legacy code for which the first way is not available, either because their is no source or API available, or because it is written in another programming language, requires Java glue. Java offers various ways of linking foreign code like, e.g., the native language interface to use C-APIs, or invocation of command-line tools. The Java glue will access the database and exchange data with the tool that is to be plugged into the platform.

5.3 Programming Language

A small study has been carried out to examine the programming language that should be used for developing the kernel tools. This study appears in Appendix A on the next page.

As a result of that study, we have decided to choose Java as the programming language for implementing the RODIN platform, including the kernel tools.

6 Conclusion

In summary, the decisions stated in this document are:

- The RODIN Platform supports the *Reactive Modelling Process*.
- Models and proof obligations are stored in a *database*. That database is implemented in an *ad-hoc* manner, tailored to the specific needs of the RODIN Platform.
- The *Project Manager* ensures that kernel tools process changes entered by the user as soon as possible (i.e., in a greedy way).

- The RODIN Platform is based on the Eclipse Platform. The implementation language of choice is Java.

References

[Caml] Xavier Leroy and Pierre Weis. *Manuel de référence du langage caml*. InterEditions, 1993. ISBN 2-7296-0492-8.

[DoW] *Description of Work*. First Annex of the RODIN Contract. April 27, 2004.

[ML] Laurence Paulson. *ML for the working programmer*. Cambridge, 1993. ISBN 0-521-39022-2.

[mP] Jean-Raymond Abrial. A simple prover specification. Internal report, July 2004.

[PP] Jean-Raymond Abrial. Le prouveur de prédicat. Internal report, August 1997.

[TOM] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *12th Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76. Springer, May 2003.

A Programming language study

This appendix describes the results of some tests done in order to make an informed decision on the implementation language for the new provers that go into the new B environment developed as part of the RODIN project. The case study involved implementing a 'mini-prover' for propositional logic that closely resembles the one envisaged for the new predicate prover. The results of this case study are presented and a conclusion is drawn from these observations.

A.1 The Case Study

The case study was to implement a mini-prover[mP] for propositional logic in Java and Caml[Caml].

The mini-prover uses pattern matching and term rewriting to automatically prove the truth or invalidity of a given formula in propositional logic.

Its specification[mP] is precise enough to be formally proven correct. It is a simplified version of the predicate prover[PP] used currently. The new predicate prover is planned to be an extension of this. The mini-prover is therefore an ideal candidate for this case study.

The implementation languages chosen for comparison are Java and Caml.

Java is a simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, dynamic language (as described on Sun's website). Moreover, it is the basic implementation language of the Eclipse Platform.

Caml is a strongly typed functional programing language. It is a dialect of ML[ML], a language initially designed with implementing theorem provers in mind. ML and its variants have successfully been used to implement many existing theorem proving systems such as Isabelle, Coq, HOL, and LCF. Caml has gone over the Standard ML specification to provide greater flexibility to the programmer. It comes with advanced compilers that can generate machine independent byte-code or even highly optimized binaries.

A.2 Observations

The two implementations were compared with each other with regards to:

- time performance,
- code size,
- ease of programming,
- ease of integration and deployment.

A.2.1 Performance

A set of 31 tautologies (30 taken from the end of [mP], and one made from their conjunction) was given to both provers 100 times and the time taken on the same machine[1] was measured.

Implementation	Time taken
Java	3ms
Caml (byte-code)	17.4ms
Caml (executable)	4.8ms

[1]Tests were run on a Thinkpad T41p running Debian GNU/Linux (sarge) on an Intel Pentium M 1.6 Mhz CPU.

At the byte-code level, the Java implementation severely outperforms its Caml equivalent. When compiled to a machine optimized binary, the Caml implementation nears the Java bite-code implementation but still does not outperform it.

Java is the clear winner.

A.2.2 The resulting code

ML was thought of with implementing theorem provers in mind. Data is represented as recursive data types and the standard way of doing computation is by pattern matching over these recursively defined data types. This makes symbolic computation, which is what is mainly done here, much easier in Caml than it is in Java. The resulting Caml code is compacter, more understandable, and much closer to the specification than its Java counterpart.

In order to make pattern matching easier in Java, a *Pattern Matching Programming Language* preprocessor, TOM[TOM] was used, making the Java code easier to write.

Implementation	Code readability	Lines of Code
Java(+TOM)	less readable	approx. 800
Caml	more readable	approx. 120

A.2.3 Ease of programing

Similarly, the time and energy needed to implement the Caml version of the mini-prover which was less than that needed for the Java version.

Implementation	Ease of programing	Programming effort
Java	harder	approx. 2 days
Caml	easier	approx. 5 hours

A.2.4 Integration with the complete tool

At the end of the day the provers implemented have to communicate with the rest of the B environment. In this respect the Java implementation has a clear advantage since the rest of the tool will be implemented in Java. Keeping all code in the same language has clear advantages with respect to ease of use and maintainability of the entire system. Another important concern is ease of deployment. Choosing Java allows seamless deployment, whereas choosing Caml adds a burden on the deployment (one needs to install either the Caml byte-code interpreter or the provers have to be deployed in binary form which have to be produced separately for every target platform).

Implementation	Integration
Java	not an issue
Caml	non-standard

A.3 Conclusion

Putting all together, we get the following simplistic overview:

Implementation	Performance	Shorter code	Ease of coding	Ease of Integration
Java	$\surd$	$\times$	$\times$	$\surd$
Caml	$\times$	$\surd$	$\surd$	$\times$

Although Caml would be more appropriate from the implementation point of view per se, since the goal of the RODIN project is to develop an industrial tool, the need for better performance and smoother integration carry a much larger weight. The language of choice should therefore be Java.