A Language Development Environment for Eclipse

M.G.J. van den Brand *† H.A. de Jong * P. Klint *‡
www.cwi.nl/~markvdb www.cwi.nl/~jong www.cwi.nl/~paulk

A.T. Kooiker *
e-mail: Taeke.Kooiker@cwi.nl

Abstract

The ASF+SDF Meta-Environment provides a collection of tools for the generation of programming environments. We show how Eclipse can be extended with these generic language tools. By integrating the GUI and text editor of the Meta-Environment with Eclipse using TOOLBUS technology, we demonstrate the integration of third party, non-Java, software in Eclipse. By doing so, we create an experimentation framework for further programming language research. We describe our experiences and sketch future work.

1 Introduction

Eclipse [7] is an open source framework for creating programming environments. Currently versions exist for C / C++[1], Java and Cobol[2]. New tools and languages can be added by writing Java applications that perform parsing, type checking and the like for a new language. Eclipse provides a rich set of tools oriented toward user-interface construction and Java compilation. The level of automation for building environments for new languages is, however, low.

The ASF+SDF Meta-Environment [3, 6] is a programming environments generator: given a language definition consisting of a syntax definition (grammar) and tool descriptions (using rewrite rules) a language specific environment is generated. A language definition typically includes such features as pretty printing, type checking, analysis, transformation and execution of programs in the target language. The ASF+SDF Meta-Environment is used to create tools for domain-specific languages and for the analysis and transformation of software systems.

As the Eclipse and Meta-Environment technologies are to a large extent complementary, it is worthwhile to investigate how they can be integrated.

1.1 Eclipse Plugin Technology

The Eclipse Platform is designed for building integrated development environments (IDEs) [7]. An IDE can be built by providing the Eclipse Platform with a plugin contributing to an extension point of some other plugin. In fact the Eclipse Platform is a number of plugins itself. It consists of a small kernel which starts all necessary plugins to run a basic instance of the Eclipse Platform. All other functionality is located in plugins which extend these basic plugins. In this way Eclipse provides tool providers with a mechanism that leads to seamlessly integrated tools.

Eclipse plugins are written in Java and consist of a manifest file and Java classes in a JAR archive. The manifest file declares the extension points of the plugin and which other plugins it extends. On start-up the Eclipse Platform discovers which plugins are available and it generates a plugin registry. The plugin itself is loaded when it actually needs to be run.

1.2 Meta-Environment Technology

The ASF+SDF formalism [1, 4] is used for the definition of syntactic as well as semantic aspects of a language. It can be used for the definition of a range of languages (for programming, writing specifications, querying databases, text processing,

*CWI, Dept. of Software Engineering, Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands

†Hogeschool van Amsterdam, Instituut Informatica en Electotechniek, Weesperzijde 190, NL-1097 DZ Amsterdam, The Netherlands

‡University of Amsterdam, Programming Research Group, Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

[1] Available at www.eclipse.org/cdt

[2] Available at www.eclipse.org/cobol

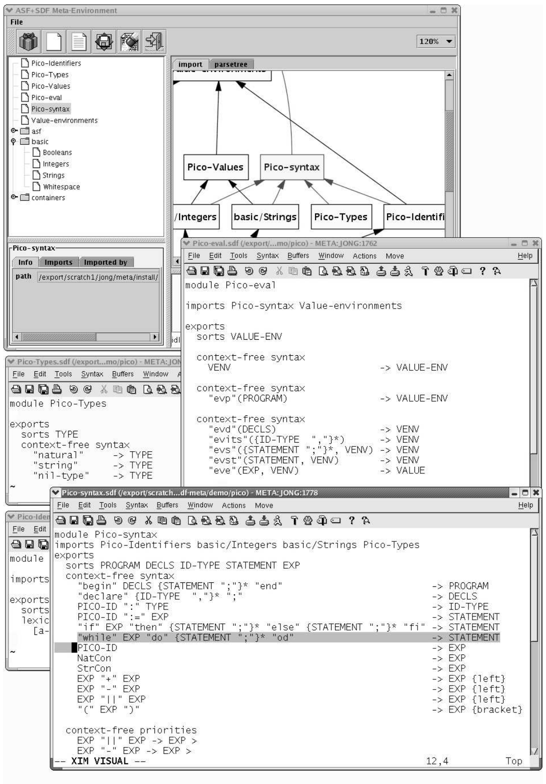

Figure 1: The Meta-Environment GUI.

or other applications). In addition it can be used for the formal specification of a wide variety of problems. ASF+SDF can be characterized as a modular, rewriting-based, specification formalism in which syntax and semantics are completely integrated.

The ASF+SDF Meta-Environment is both a programming environment for ASF+SDF specifications and a programming environment generator which uses an ASF+SDF specification for some (programming) language L to generate a stand-alone environment for L. The design of the Meta-Environment is based on openness, reuse, and extensibility. The Meta-Environment offers syntax-directed editing of ASF+SDF specifications as well as compilation of ASF+SDF specifications into dedicated interactive stand-alone environments containing various tools such as a parser, unparser, syntax-directed editor, debugger, and interpreter or compiler.

Figure 1 shows the user interface developed using JFC/Swing. This figure shows the modular structure of the specification. Each node in the graph can be clicked and allows the invocation of a syntax, equation, or term editor.

The various types of editors are decorated with

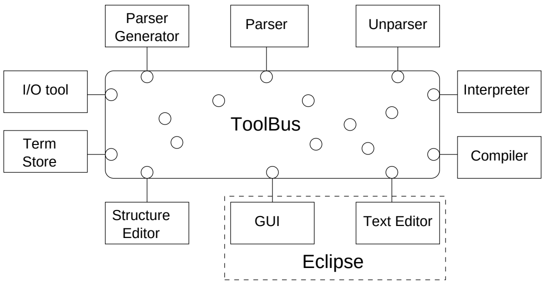

Figure 2: Architecture of the Meta-Environment using the ToolBus.

different pull-down menus. All editors have functionality to invoke the parser, view the parse tree of the focus as graph, and to move the focus. Term editors may have language specific pull-down menus.

In order to achieve a strict separation between coordination and computation we use the TOOLBUS *coordination architecture* [2], a programmable software bus based on process algebra. Coordination is expressed by a formal description of the cooperation protocol between components while computation is expressed in components that may be written in any language. We thus obtain interoperability of heterogeneous components in a (possibly) distributed system. The components are not allowed to communicate directly with each other, but only via the TOOLBUS. This leads to a rigorous separation of concerns.

2 Architectural considerations

The Meta-Environment consists of about 20 cooperating components, including a parsetable generator, a parser and unparser, a term store (for caching results), and an interpreter and compiler. Also, a graphical user interface and a number of text editors (such as GNU Emacs[3] and Vim[4]) as well as a structure editor are connected to the Meta-Environment. These allow user interaction with the system, and in particular allow users to edit syntax, equations and terms. Figure 2 is a (simplified) view showing these components connected to the TOOLBUS.

Current architecture: using JFC/Swing and external editors. Figure 2 shows the current implementation with separate components for the GUI and the various text editors. Currently, the GUI is

[3]Available at `www.gnu.org/software/emacs`
[4]Available at `www.vim.org`

implemented in JFC/Swing. Each time a text editing session is requested by the user, a new instance of one of the supported system editors is executed to take care of the editing session. These text editors need only implement a minimal interface to be usable by the Meta-Environment. Some form of operating system level communication channel is needed (e.g. socket, pipe). The editor then needs to be able to receive and execute commands to add a menu to the menu-bar, set the cursor at a specific location, and highlight or select a region of text.

Target architecture: using Eclipse for both GUI and editors. Eclipse exports many GUI features that can be used to write plugins and also has a built-in editor which implements the required Meta-Environment text editor interface. From an Eclipse point of view, it is interesting to be able to reuse the generic language technology offered by the Meta-Environment. From the Meta-Environment point of view, it would be interesting to see if Eclipse could be used to implement the GUI and the text editors (the dotted rectangle in Figure 2). From a TOOLBUS point of view, it is interesting to see how a *single* tool (Eclipse in this case) can serve as the implementation of *multiple* components (both GUI and text editor).

3 Implementation

In Section 3.1 we describe some of the implementation details of the current Meta-Environment GUI.

In the target architecture we replace both the JFC/Swing GUI and the external text editors by Eclipse as described in Section 3.2.

3.1 JFC/Swing-based implementation

The TOOLBUS principle to separate functionality leads to a generic implementation of the user interface. To meet the Meta-Environment requirements the user interface only has to implement some basic functionality. The JFC/Swing implementation extends the Meta-Environment with a GUI that supports several components: a tree panel, graph panel, and some informational panels. The tree and graph panels provide the user with a representation of opened and imported modules in a textual and graphical way, respectively. Status messages and information about selected modules are displayed in dedicated informational panels. Each of these GUI elements is *dumb*: it is capable of presenting a graphical representation of its data and it communicates events (e.g. a mouse click) to the TOOLBUS, but it abstracts from the details of these events. The actual implementation of an event (e.g. performing a refactoring operation on a selected module) is handled elsewhere in the Meta-Environment.

The provided basic framework can be extended dynamically with user interface elements by means of TOOLBUS messages sent to the user interface. These messages contain the type of user interface element to be added (e.g. menu, tool-bar button), the caption to be displayed, and the action that has to be performed when selecting this user interface element. This setup ensures that the user interface does not know about any functionality implemented in the Meta-Environment.

Text editing functionality is provided by means of external text editors as described before. In general the choice of text editor is free as long as it is capable of adding menus and methods for displaying a focus. After connection with the TOOLBUS is established it will receive its specific menus, menu items, and corresponding actions.

3.2 Eclipse-based implementation

In order to use Eclipse for the implementation of the Meta-Environment GUI and text editor, we adapt the Meta-Environment architecture as shown in Figure 3. In a TOOLBUS setting external tools (such as a GUI and text editor) are rigorously separated components which never directly communicate with each other, but always do so via the TOOLBUS. In order to connect Eclipse (a single operating system level component) to the Meta-Environment, we use a second TOOLBUS which acts as a proxy between the Meta-Environment on one side, and the actual implementations of the GUI and text editor in Eclipse on the other. This second TOOLBUS, together with two instances of a transparent stub (one for the GUI and one for the text editor) takes care of any (de-)marshalling and forwarding from the Meta-Environment to Eclipse and back.

The Eclipse Meta plugin is implemented as an Eclipse *perspective*, containing extensions of an *explorer* (to display the modules), several *views* (e.g. to display status messages) and instances of an extension of the built-in *editor*. The perspective itself takes care of setting up a connection to the TOOLBUS before instantiating the other Eclipse *view parts* which receive their operational details from the TOOLBUS. Figure 4 shows the Eclipse

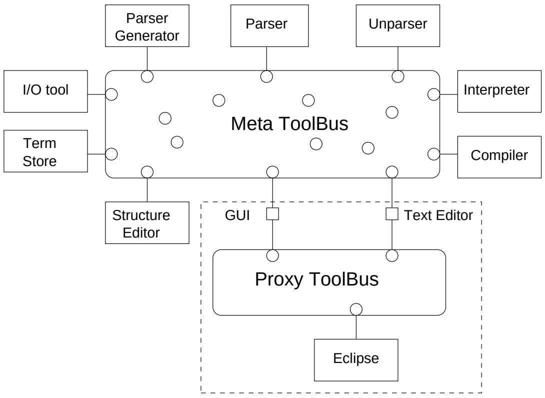

Figure 3: Eclipse as implementation of the GUI and text editors.

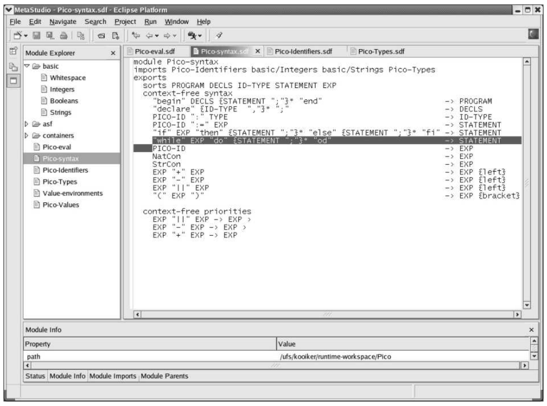

Figure 4: The Meta-Environment in Eclipse.

user interface of the Meta-Environment.

4 Lessons learned

We have identified several opportunities for improvement in both the JFC/Swing Meta-Environment (Section 4.1) as well as in Eclipse (Section 4.2).

4.1 Meta-Environment issues

Complex editor management. Before we started integrating the Meta-Environment and Eclipse, all text editor management was handled in several TOOLBUS processes. For each editing session, the TOOLBUS invoked a new instance of the system editor. This conflicted with Eclipse, because Eclipse already handles multiple editor instances itself. Since the original setup was quite complex, we decided to encapsulate this complexity in a separate tool. The JFC/Swing implementation now uses this tool, the Eclipse setting handles the editor management inside Eclipse itself.

4.2 Eclipse issues

Most of the Meta-Environment functionality present in the JFC/Swing version was implemented equally well in the Eclipse version, but we did encounter some difficulties which we would hope to see eliminated in a future version of Eclipse.

No support for File Open dialog. An important difference between the current Meta-Environment and Eclipse is exposed when we consider how to open new modules. The current JFC/Swing implementation delegates *open module* events to the TOOLBUS, where other processes subsequently ask for the instantiation in the GUI of a “File Open” dialog to ask the user for the name of the module to be opened. Because Eclipse does not have such a dialog, we had to implement the opening of modules quite differently. The user first selects a file in the module explorer, and then hits the open module button. This causes the order of user interaction in Eclipse (select file, hit button) to be the opposite of the original order in JFC/Swing (hit button, select file).

No access to files outside workspace. Eclipse only allows access to files residing in the workspace. Files outside the workspace first need to be imported into the workspace, before they can be used. However, when the Meta-Environment uses a module, it also needs the transitive closure of its imported modules which are not necessarily located in the workspace (they could be anywhere on the file system). As a consequence, a user cannot edit any module that is not part of the workspace.

Plugin configurability too rigid. The plugin manifest file is not usually edited by plugin users. One of the things that is *hard coded* in this manifest, is the link between file extension and corresponding editor to be used in Eclipse when such a file needs to be edited. Because the Meta-Environment has no fixed language, and file extensions are often associated with a particular, new, language, an explicit link between each developed language and the Meta-Environment plugin editor has to be inserted in the manifest manually.

Workbench state management too Eclipse centric Eclipse keeps track of the state of the workbench. There is no flexibility when an external tool also needs to maintain a portion of this state. This interferes with the way the Meta-Environment operates. Upon Eclipse startup, *views* from a previous session are still present in the workbench, but they do not have the state from the previous session. Most notably, any connection to the TOOLBUS is lost, and in fact, the rest of the Meta-Environment components may not even have been started yet. A `Perspective.close()` method (not yet available, as other plugin writers have noted in the Eclipse newsgroups) would already have been useful, as it would have allowed us to simply close any *view* that is managed by the Meta-Environment.

5 Conclusions

The main contributions of this work are as follows:
i) A proof-of-concept connection between Eclipse and the Meta-Environment: this extends Eclipse with language definition tools and extends the Meta-Environment with richer user-interface functionality.
ii) The TOOLBUS provides a general mechanism for connecting non-Java tools to Eclipse.
iii) We have pinpointed several issues of possible improvement in both systems.

The presented Eclipse Meta Plugin consists of a user interface and text editing capabilities as already provided by the JFC/Swing Meta-Environment. Through the Eclipse user interface, all generators of the Meta-Environment are available.

We plan to work on extending each system by integrating functionality from the other one. On the one hand, Eclipse provides functionality for on-line help, documentation and error reporting. All these can be borrowed by the ASF+SDF Meta-Environment.

On the other hand, we are currently integrating the Meta-Environment's graph viewer into Eclipse. Other useful functionality is APIGEN [5] which generates application program interfaces in C and Java from a grammar definition. This might make Eclipse further open for non-Java tools.

The integration experiment we described in this paper shows that the combination Eclipse/ASF+SDF Meta-Environment creates a versatile experimentation platform for programming language research.

6 About the Authors

Mark van den Brand is senior researcher at Centrum voor Wiskunde en Informatica (CWI, the Dutch national research center for computer science and mathematics) and lecturer at the Hogeschool voor Amsterdam. Hayco de Jong is a PhD researcher at CWI. Paul Klint is head of the software engineering department at CWI and professor in computer science at the University of Amsterdam. Taeke Kooiker is a graduate student at the University of Amsterdam.

References

[1] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[2] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, 1998.

[3] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[4] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[5] H.A. de Jong and P.A Olivier. Generation of abstract programming interfaces from syntax definitions. Technical Report SEN-R0212, St. Centrum voor Wiskunde en Informatica (CWI), August 2002. To appear in Journal of Logic and Algebraic Programming.

[6] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[7] Eclipse platform technical overview. Object Technology International, Inc., 2003.