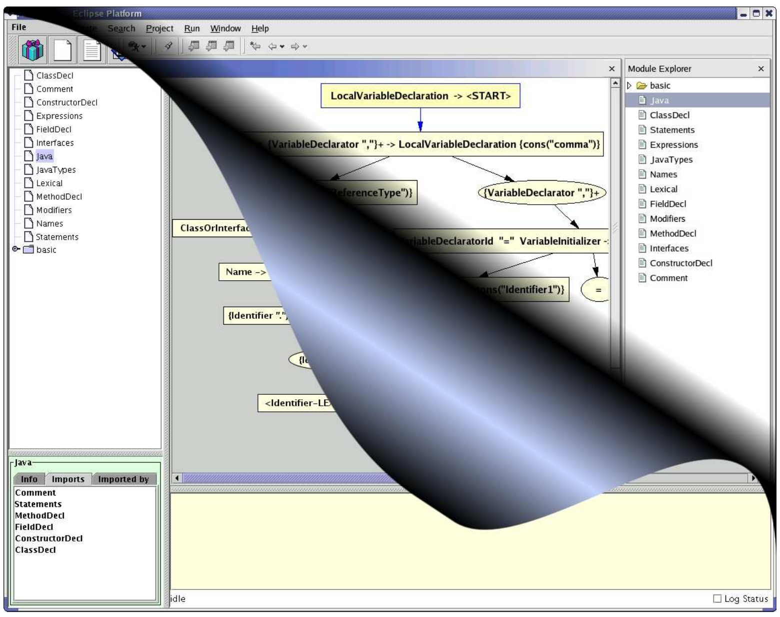
A Language Development Environment for Eclipse

A.T. Kooiker

January 2004



Master's Thesis in Computer Science
Programming Research Group
Faculty of Science
University of Amsterdam
Supervisors: prof. dr. P. Klint, drs. H.A. de Jong

Contents

Chapter 1

Introduction

1.1 General

The Asf+Sdf Meta-Environment is an interactive development environment with an graphical user interface written in JFC/Swing. A typical session in this environment results in an overwhelming amount of windows presenting the Meta-Environment itself next to lots of editors all trying to catch the attention of the user. Although the described scene is a little bit exaggerated the several open windows and JFC/Swing graphical user interface were the reasons to take a look into Eclipse.

Eclipse is a framework for creating programming environments, wherefore currently versions are available for C / C++, java and Cobol. Each version addresses a specific programming language with its own set of tools, commands and screen layout. Because Eclipse is a well developed environment the Asf+Sdf Meta-Environment could benefit from integration with Eclipse. Eclipse offers the possibility to make your own *perspective* (which is a screen layout) consisting of tree views, editors and other kind of views all in one place.

Eclipse on the other side can benefit from the integration with the Meta-Environment because the Meta-Environment brings generic language technology to Eclipse. Instead of developing a development environment for each programming language one development environment for the Meta-Environment is enough to provide tools for every programming language developed in the Meta-Environment.

Chapter 2 covers the results of the integration of the Meta-Environment with Eclipse. This chapter has been presented at the Eclipse workshop at OOPSLA 2004 and is being published in the ACM Digital Library[1]. To be able to understand what has to be done to integrate the Meta-Environment with Eclipse a brief introduction to both the Meta-Environment (see Chapter 3) and the underlying ToolBus technology (see Chapter 4) will be given. The technical details of the Eclipse Meta Plugin will be explained in Chapters 5 and 6.

[1]Available at `portal.acm.org/dl.cfm`

1.2 Acknowledgements

Writing a thesis is not an everyday task you can do on your own. Therefore I would like to thank the following people:

My parents for supporting me during all these years and giving me the chance to study Computer Science.

Wendy, although telling me that I did the job, she was the one that made me realize I had to finish it.

Paul Klint for giving me the possibility to work in a nice research environment with friendly colleagues and for letting me visit the OOPSLA 2003 conference in Los Angeles. It was a great experience!

Hayco, for sharing lots of fun next to some serious pair programming and supporting me in The States (which was needed, especially after someone said the nice 25 person room changed that morning to a room for 80 people).

Furthermore thanks to Mark van den Brand and Jurgen Vinju for answering all questions I had. One fool can ask more than Mark and Jurgen can answer.

Chapter 2

A Language Development Environment for Eclipse

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.*

2.1 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]Available at `www.eclipse.org/cdt`
[2]Available at `www.eclipse.org/cobol`

2.1.1 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.

2.1.2 Meta-Environment Technology

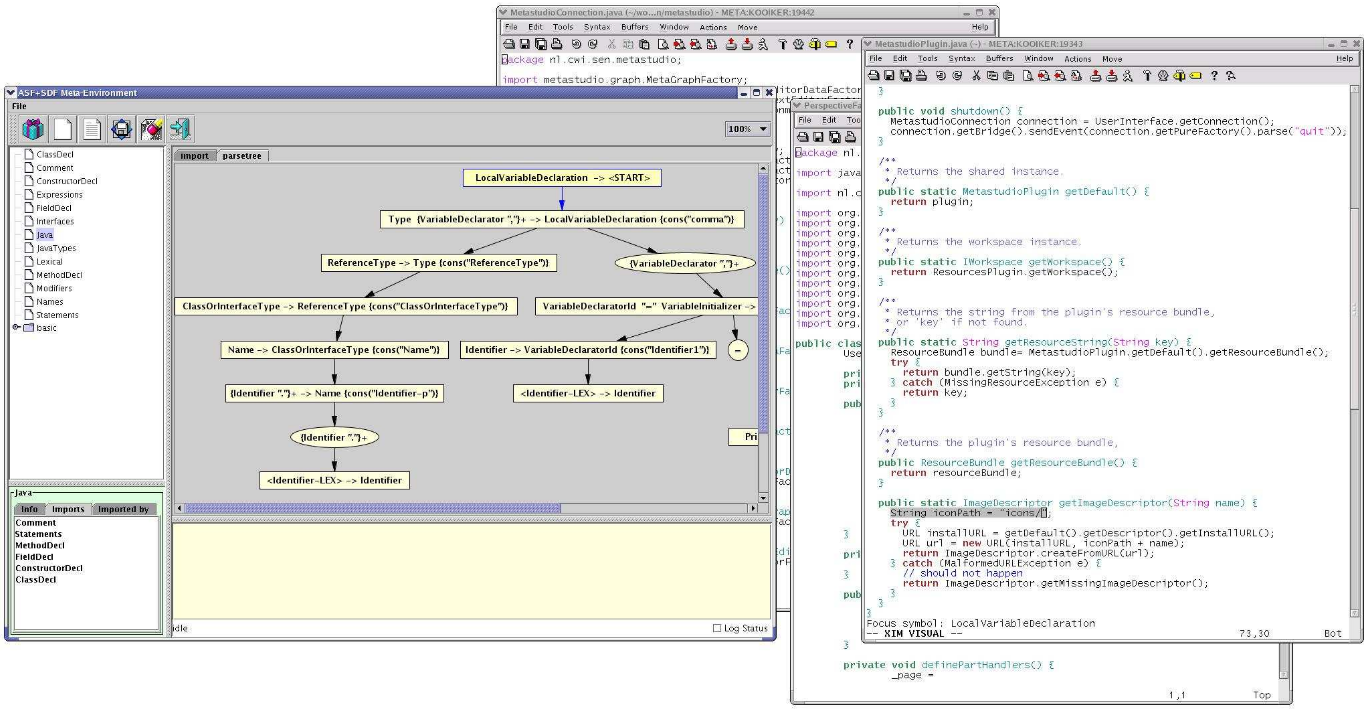
The ASF+SDF formalism [1, 4] is used for the definition of syntactic as well as semantic aspects of a language. It can be used for the definition of a range of languages (for programming, writing specifications, querying databases, text processing, or other applications). In addition it can be used for 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 2.1 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 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.


MetastudioConnection.java (~/wo...n/metastudio) - META:KOOIKER:19442
MetastudioPlugin.java (~) - META:KOOIKER:19343
File Edit Tools Syntax Buffers Window Actions Move Help
ASF+SDF Meta-Environment
File
100%
ClassDecl
Comment
ConstructorDecl
Expressions
FieldDecl
Interfaces
Java
JavaTypes
Lexical
MethodDecl
Modifiers
Names
Statements
basic
import
parsetree
LocalVariableDeclaration -> <START>
Type {VariableDeclarator ","}+ -> LocalVariableDeclaration {cons("comma")}
ReferenceType -> Type {cons("ReferenceType")}
{VariableDeclarator ","}+
ClassOrInterfaceType -> ReferenceType {cons("ClassOrInterfaceType")}
VariableDeclaratorId "=" VariableInitializer ->
Name -> ClassOrInterfaceType {cons("Name")}
Identifier -> VariableDeclaratorId {cons("Identifier1")}
=
{Identifier "."}+ -> Name {cons("Identifier-p")}
<Identifier-LEX> -> Identifier
{Identifier "."}+
<Identifier-LEX> -> Identifier
Java
Info
Imports
Imported by
Comment
Statements
MethodDecl
FieldDecl
ConstructorDecl
ClassDecl
idle
Log Status

```
package nl.cwi.sen.metastudio;

import metastudio.graph.MetaGraphFactory;
```

```
  }

  public void shutdown() {
    MetastudioConnection connection = UserInterface.getConnection();
    connection.getBridge().sendEvent(connection.getPureFactory().parse("quit"));
  }

  /**
   * Returns the shared instance.
   */
  public static MetastudioPlugin getDefault() {
    return plugin;
  }

  /**
   * Returns the workspace instance.
   */
  public static IWorkspace getWorkspace() {
    return ResourcesPlugin.getWorkspace();
  }

  /**
   * Returns the string from the plugin's resource bundle,
   * or 'key' if not found.
   */
  public static String getResourceString(String key) {
    ResourceBundle bundle= MetastudioPlugin.getDefault().getResourceBundle();
    try {
      return bundle.getString(key);
    } catch (MissingResourceException e) {
      return key;
    }
  }

  /**
   * Returns the plugin's resource bundle,
   */
  public ResourceBundle getResourceBundle() {
    return resourceBundle;
  }

  public static ImageDescriptor getImageDescriptor(String name) {
    String iconPath = "icons/";
    try {
      URL installURL = getDefault().getDescriptor().getInstallURL();
      URL url = new URL(installURL, iconPath + name);
      return ImageDescriptor.createFromURL(url);
    } catch (MalformedURLException e) {
      // should not happen
      return ImageDescriptor.getMissingImageDescriptor();
    }
  }
}
Focus symbol: LocalVariableDeclaration
-- XIM VISUAL --                                  73,30        Bot
```

```
private void definePartHandlers() {
      _page =
                                              1,1          Top
```

Figure 2.1: The Meta-Environment GUI.

2.2 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 2.2 is a (simplified) view showing these components connected to the ToolBus.

Current architecture: using JFC/Swing and external editors. Figure 2.2 shows the current implementation with separate components for the GUI and the various text editors. Currently, the GUI is 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.

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

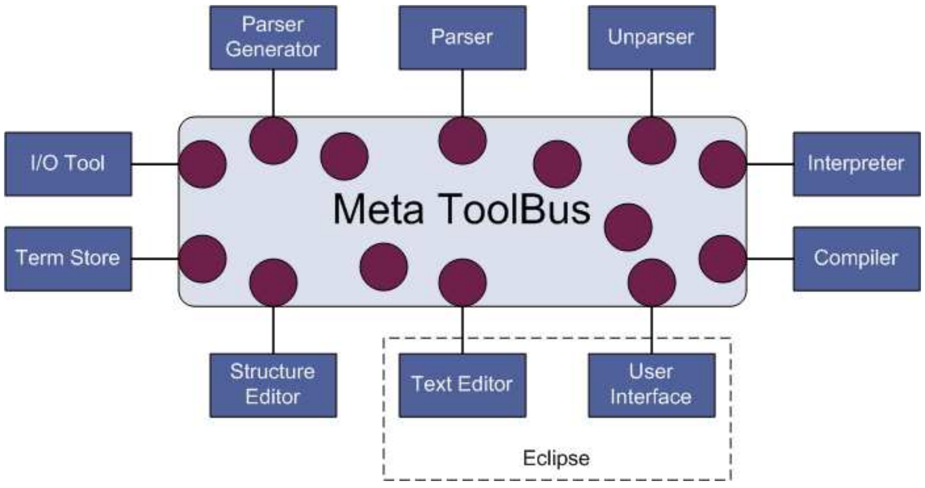


Figure 2.2: Architecture of the Meta-Environment using the ToolBus.

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 2.2). 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).

2.3 Implementation

In Section 2.3.1 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 2.3.2.

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.

2.3.2 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 2.3. 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 2.4 shows the Eclipse user interface of the Meta-Environment.

2.4 Lessons learned

We have identified several opportunities for improvement in both the JFC/Swing Meta-Environment (Section 2.4.1) as well as in Eclipse (Section 2.4.2).

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

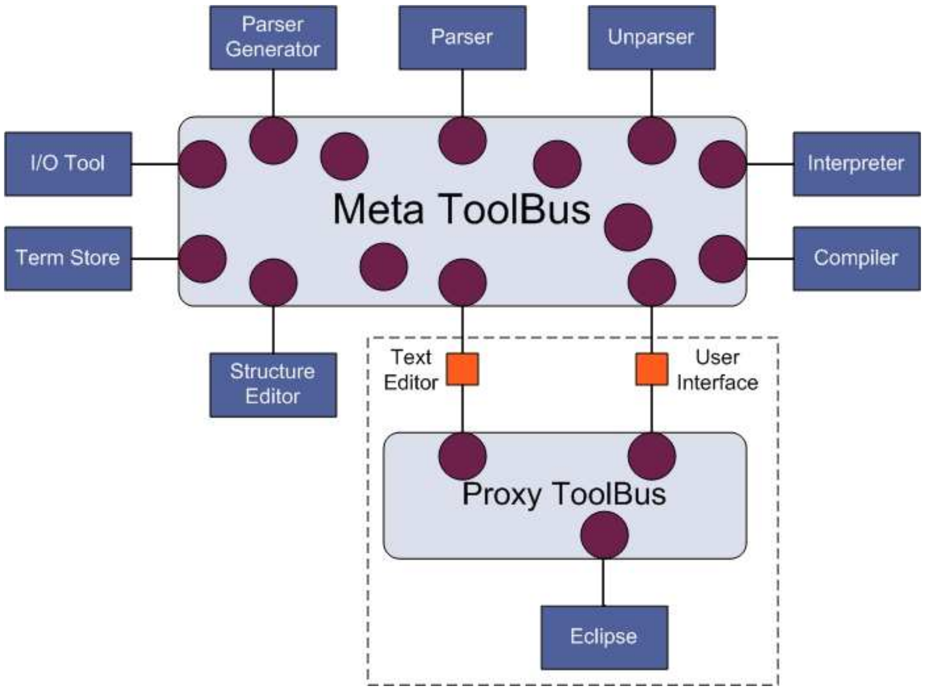


Figure 2.3: Eclipse as implementation of the GUI and text editors.

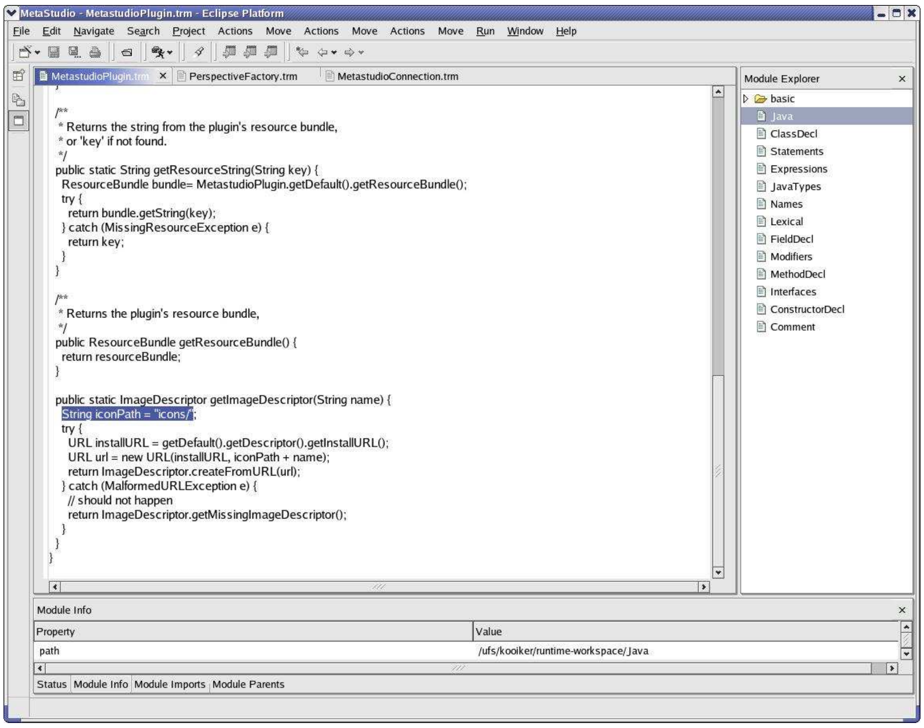
JFC/Swing implementation now uses this tool, the Eclipse setting handles the editor management inside Eclipse itself.

2.4.2 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



```
/**
 * Returns the string from the plugin's resource bundle,
 * or 'key' if not found.
 */
public static String getResourceString(String key) {
  ResourceBundle bundle= MetastudioPlugin.getDefault().getResourceBundle();
  try {
    return bundle.getString(key);
  } catch (MissingResourceException e) {
    return key;
  }
}

/**
 * Returns the plugin's resource bundle,
 */
public ResourceBundle getResourceBundle() {
  return resourceBundle;
}

public static ImageDescriptor getImageDescriptor(String name) {
  String iconPath = "icons/";
  try {
    URL installURL = getDefault().getDescriptor().getInstallURL();
    URL url = new URL(installURL, iconPath + name);
    return ImageDescriptor.createFromURL(url);
  } catch (MalformedURLException e) {
    // should not happen
    return ImageDescriptor.getMissingImageDescriptor();
  }
}
}
```

Module Info

Property	Value
path	/ufs/kooiker/runtime-workspace/Java

Figure 2.4: The Meta-Environment in Eclipse.

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.

2.5 Conclusions

The main contributions of this work are as follows:

- 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.
- The ToolBus provides a general mechanism for connecting non-JAVA tools to Eclipse.
- 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.

Chapter 3

ASF+SDF Meta-Environment

The ASF+SDF *Meta-Environment is an interactive development environment for the automatic generation of interactive systems for the construction of language definitions. Understanding what this means and what can be done with the* ASF+SDF *Meta-Environment is needed to make integration with Eclipse possible.*

3.1 Introduction

The ASF+SDF Meta-Environment is an interactive development environment for the automatic generation of interactive systems for the construction of language definitions. After a language definition has been constructed, tools for this language can be generated (e.g. parser, type checker, and compiler). A language definition typically consists of a syntax (SDF) and a semantics (ASF) definition. The ASF+SDF Meta-Environment can help in:

- Writing a formal specification for some problem where interactive support is needed.
- Developing programming languages and creating an interactive environment for it.
- Analyzing or transforming programs in some existing programming language.

Using the Meta-Environment new specifications can be made, specifications can be modified and, using terms, existing specifications can be tested. A language specification consists of modules which can be edited by invoking an external editor. These modules consist on their turn of the aforementioned syntax and semantics (equations) part.

The ASF+SDF Meta-Environment consists of about 25 components that are connected via the TOOLBUS which will be discussed in Chapter 4. Figure 3.1 shows the basic architecture with some of the most important components.

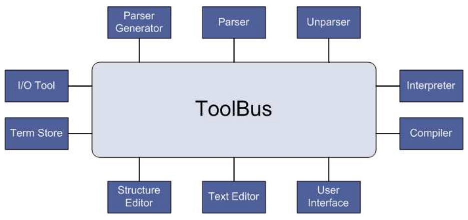


Figure 3.1: Basic architecture of the ASF+SDF Meta-Environment

Because this thesis is about integrating the Meta-Environment with Eclipse we will not further discuss the technology used in the Meta-Environment, but focus on the user interface of the Meta-Environment (see Section 3.2).

3.2 Graphical User Interface and Editors

Part of integrating the Meta-Environment with Eclipse is creating a graphical user interface in Eclipse that matches the user interface of the original Meta-Environment. In this section a description of the several parts of the Meta-Environment user interface will be given.

The main window of the Meta-Environment consists of the following parts:

- A *menubar* containing menus to import and edit modules.
- A *toolbar* showing the most common functions (e.g. `New Module`, `Open Module`)
- An *import pane* showing all opened modules as an import tree.
- An *graph pane* showing a graphical representation of the import tree mentioned above.
- An *information pane* showing parents and children of the selected module and its location.
- A *message pane* showing the message (errors and warnings) history.
- A *status bar* that shows the current activity of the Meta-Environment(e.g. `idle`, `parsing`, focus symbol).

Because the Meta-Environment does not provide any editing functionality, editing modules is done using separate editors (e.g. gvim or GNU Emacs). Any editor can be used as long as the editor of choice can be externally operated to offer menus and functionality related to the module being edited.

Chapter 4

ToolBus

The ToolBus *coordination architecture plays an important role in the connection between Eclipse and the* Asf+Sdf *Meta-Environment. We first give a brief introduction to the working of the* ToolBus*. Thereafter we show the possibilities to connect Eclipse to the Meta-Environment, whereafter the final solution, using a second* ToolBus*, will be described in detail.*

4.1 Introduction

The ToolBus *coordination architecture* is the interconnection architecture used in the Meta-Environment. The ToolBus can be compared with a hardware communication bus, except it is entirely software based. Processes described by process algebra scripts provide the coordination between the various components. These components (or tools) can be written in any language. The fact that these tools are not allowed to communicate directly with each other leads to a rigorous separation of coordination and computation. The ToolBus architecture will be briefly described in Section 4.2 and Section 4.3.

The two major processes with respect to Eclipse are the *user interface* and *editor-hive* processes. These processes form the core of the graphical user interface of the Meta-Environment. The goal is to connect Eclipse to these two processes, which is slightly different than connecting other tools to the Meta-Environment. Under normal circumstances tools are never connected to more than one process without violating the idea of separation of concerns.

In this chapter we will describe the architecture used to connect Eclipse to the Meta-Environment in such a way that it does not affect the functionality of the original JFC/Swing environment.

4.2 ToolBus coordination: processes

The coordination between ToolBus components (tools) is orchestrated by processes. Each tool connected to the ToolBus has its own ToolBus process. Figure 4.1 shows a schematic view of the ToolBus. Circles indicate the various processes, whereas squares indicate tools.

Communication between processes can be done either synchronously or asynchronously. The ToolBus script primitive `snd-msg` is used for synchronous

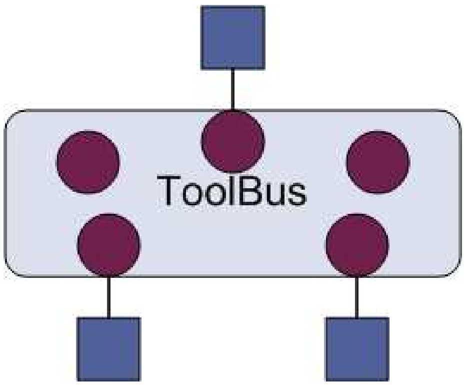


Figure 4.1: Schematic view of the ToolBus

communication. When using this primitive a process can only send a message to another process when this other process is able to receive the message. When there is no process available for receiving this message, the sending process will block until it can deliver the message.

Asynchronous communication can be used if it is not sure that there are receiving processes. The primitive `snd-note` must be used to send an asynchronous message. To be able to receive this message a process has to be subscribed to the message. Even if there are no receiving processes, the sending process will go on after sending its message.

Communication between processes and tools is bidirectional. Processes can send messages to connected tools using the `snd-do` and `snd-eval` primitives. A `snd-eval` message sends an evaluation request to a tool. After evaluating the message the tool has to return a value in a `snd-value` message. A `snd-do` message just notifies the tool and does not expect a return value.

Tools can also initialize communication by using the `snd-event` primitive. After receiving an event the process has to acknowledge this by sending a `snd-ack-event` message.

4.3 ToolBus computation: tools

Tools can be written in any programming language for which a ToolBus adapter exists. Connection to the ToolBus is provided by a language specific adapter. In the first stage of establishing a connection between a tool and the ToolBus, the tool receives the signature of the process it is connecting to. The signature consists of all messages the tool has to implement in order to function as supposed. Depending on rejection or acceptation of the signature the tool will send a message stating it cannot handle the messages provided in the signature or will be functional and do its job.

4.4 The ToolBus and Eclipse

While investigating how to make a connection between Eclipse and the Meta-Environment two possibilities came up. First Eclipse could be connected directly

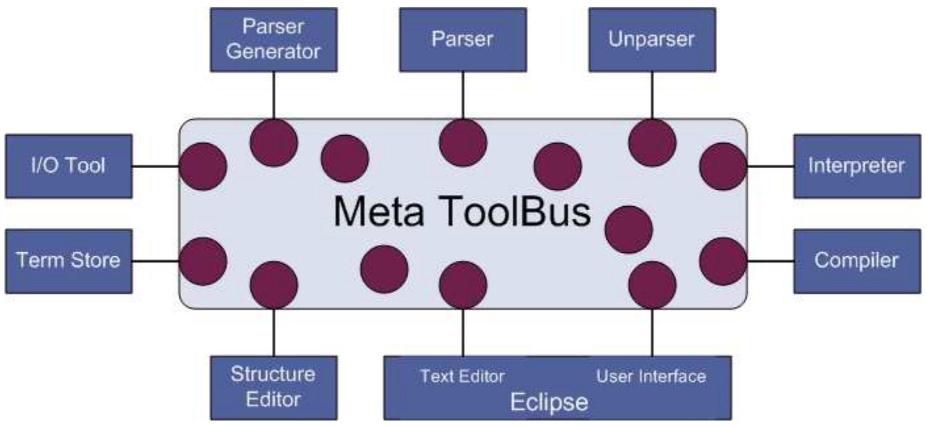


Figure 4.2: Eclipse connected via 2 JavaAdapters

to the ToolBus using JavaAdapters. The second solution connects Eclipse via a proxy to the ToolBus. The advantage of this solution is that it provides a generic way of connecting one tool to multiple ToolBus processes.

4.4.1 Solution 1: Two JavaAdapters

The general solution to connect a tool to the ToolBus is using a adapter. Because Eclipse is entirely written in java makes the decision to make use of the JavaAdapter an easy one. The fact that Eclipse has to replace two tools (*Editor Hive* and *User Interface*) and each tool has to connect to the ToolBus via its own adapter, results in the use of two JavaAdapters. Figure 4.2 shows the result of this setup: one tool connects via two JavaAdapters to two processes.

Unfortunately this design leads to threading problems. In Eclipse the user interface runs in its own thread. Non user interface methods have to run in their own threads to prevent the user interface from being irresponsive. While using one JavaAdapter to connect to one of the processes did work, using a second JavaAdapter blocked user interface and ToolBus access. It became clear that using more than two threads in java leads to problems in Eclipse, locking all threads. One solution to this problem was using another Java Virtual Machine, but we came up with another solution in the form of a proxy between the ToolBus and Eclipse.

4.4.2 Solution 2: Proxy ToolBus

While connecting Eclipse with two JavaAdapters could have worked, a more challenging solution is thinking of Eclipse as one tool that has to connect to one process, while leaving the Meta-Environment intact. This solution fits the idea that one tool connects to one process and the idea of separation of concerns.

Because Eclipse will now be connected as one tool to one process a middleware-alike solution is needed to redirect messages from the two Meta-Environment processes (*editor hive* and *user interface*) to the process that connects Eclipse and vice versa. This middleware was already present in the form of the ToolBus. To distinguish between the ToolBus used in the Meta-Environment and

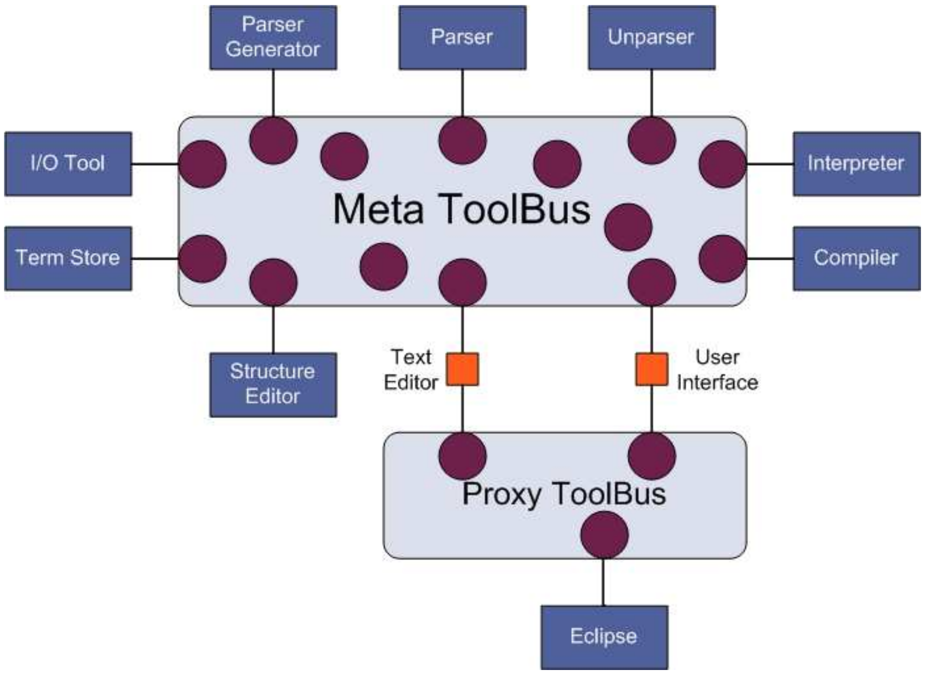


Figure 4.3: Eclipse architecture with Proxy ToolBus

this middleware ToolBus, in the remainder of this thesis the first one is referred to as Meta ToolBus while the second one is referred to as Proxy ToolBus (see Figure 4.3).

Chapter 5

Connecting Eclipse to the Meta-Environment

Before a connection between the Meta-Environment and Eclipse could be made a connection architecture had to be setup. Part of this connection architecture is the `Editor Hive` *which is a tool that takes care of spawning, communicating, and administrating editors. Another part of the connection architecture is the Proxy* ToolBus*, which is the interconnection* ToolBus *that orchestrates communication between Eclipse and the Meta-Environment* ToolBus *and provides a generic way to connect one tool to more than one process.*

5.1 Introduction

Making a connecting between Eclipse and the Meta-Environment consisted of changing the way the Meta-Environment made use of editors and creating an interconnection architecture between the Meta-Environment and Eclipse. This chapter roughly consists of two parts. The first part describes the way editors worked and why this had to be changed (Section 5.2) whereas the second part describes the interconnection architecture (Section 5.3) to provide a middleware solution to connect Eclipse to the Meta-Environment.

5.2 Editors and the Meta-Environment

Although one of the goals of this thesis was to make a connection between Eclipse and the Meta-Environment without making changes to the Meta-Environment, there were some pending changes that had to be made before a connection between Eclipse and the Meta-Environment could be made. Before making the connection with Eclipse, the Meta-Environment contained an editor manager process which would invoke editor processes when needed. For each editing session such an editor process would be spawned and registered with the editor manager.

The Meta-Environment could communicate with a specific editor via a routing mechanism. A message could be sent to a specific editor by requesting the editor identifier from the editor manager. The editor manager would then lookup the matching editor process and send the result back, whereafter the message could be sent to the editor process. The editor process in its turn takes care of spawning an editor (e.g. gvim or x-emacs) and setting up a connection with this editor to be able to communicate with it.

5.2.1 Editor Manager

While using the Meta-Environment more than one editing session at a time can be active for editing modules and terms. To be able to address each editor a unique identifier is assigned to each editing session. Administration of this session identifier is done by the editor manager. When communication with a specific editing session is needed a request is sent to the editor manager to return the session identifier. This session identifier can be used in the further communication with the editing session.

In the case that there is no session identifier found a new session identifier will be generated. This new identifier will be returned with a message notifying the receiver that a new editor has been spawned for this editing session. After an editing session has been closed the editor manager will be notified of the regarding session identifier and takes care of removing this session identifier from the registered identifiers.

5.2.2 Editor Hive

Orignally each editing session had its own correspoding ToolBus process. To be able to address each editing session a lot of administration had to be done in the ToolBus. This was thought of as a bad habit, because the ToolBus itself is primarily meant to do coordination not computation.

Besides this ethical problem it also meant that Eclipse had to connect to an undefined number of processes. For each editing session a TransAdapter would be needed, but also a stub process adjusted to receive messages for that specific editing session from the merged process. The main problem arises in this merged process. For each editing session this process has to be extended with an extra set of messages for that editing session.

Suppose the merged process already consists of $S1||S2$. When the user requests a new editing session a new editor process would be invoked. The only way the merged process can address this editor process is by introducing a new set of messages $S3$ in parallel with $S1||S2$ resulting in $S1||S2||S3$. A ToolBus process can never be extended dynamically, so the aforementioned method can not be used.

To ban all computation from the ToolBus the editor hive was introduced. The editor-hive tool takes care of invoking and administrating editors. Now only one process handling editing sessions is needed. Figure 5.1 shows the editor-hive tool and its according ToolBus process. As can be seen a whole range of editors can be used as long as these editors comply with the given framework. This framework consists of the set of messages an editor has to implement and an adapter to connect the editor to the editor-hive. Herefore the editor has to be externally managable.

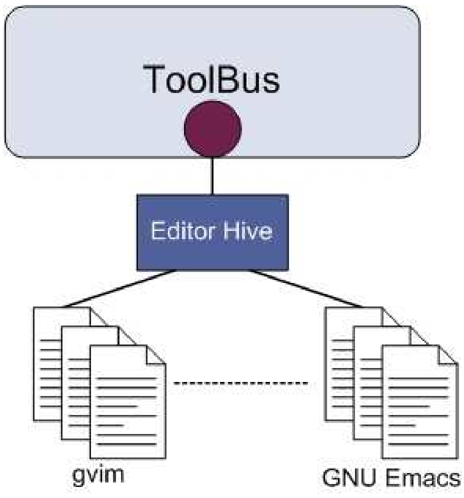


Figure 5.1: Schematic view of the editor-hive

5.3 Proxy ToolBus

The Proxy ToolBus provides a connection to the Meta-Environment on the one side and an connection to Eclipse on the other side. Every process in the Meta-Environment that has to be connected to Eclipse has a stub process in the Proxy ToolBus. For the connection with Eclipse there is one process that consists of all messages Eclipse has to implement. The set of messages Eclipse has to implement is a merge of the sets of messages of the several processes Eclipse has to connect to: in this case the set of messages of the user interface and editor-hive processes.

Finally the connection between the Meta ToolBus and the Proxy ToolBus is established by using a TransAdapter (indicated by the letter A in Figure 5.2). This tool connects to both ToolBuses and forwards any message from one ToolBus to the other. The working of this tool will be discussed in Section 5.4.

5.3.1 Stub process

The stub process P (see Figure 5.2) running in the Proxy ToolBus forwards all incoming messages from the TransAdapter to the merged process and the other way around. An example stub process for the user interface process is shown below.

```
tool stub is {}

process STUB is
let
  S : stub1,
  Event : term,
  Message : term
in
  rec-connect(S?)
  .
  (
    rec-event(S, Event?)
```

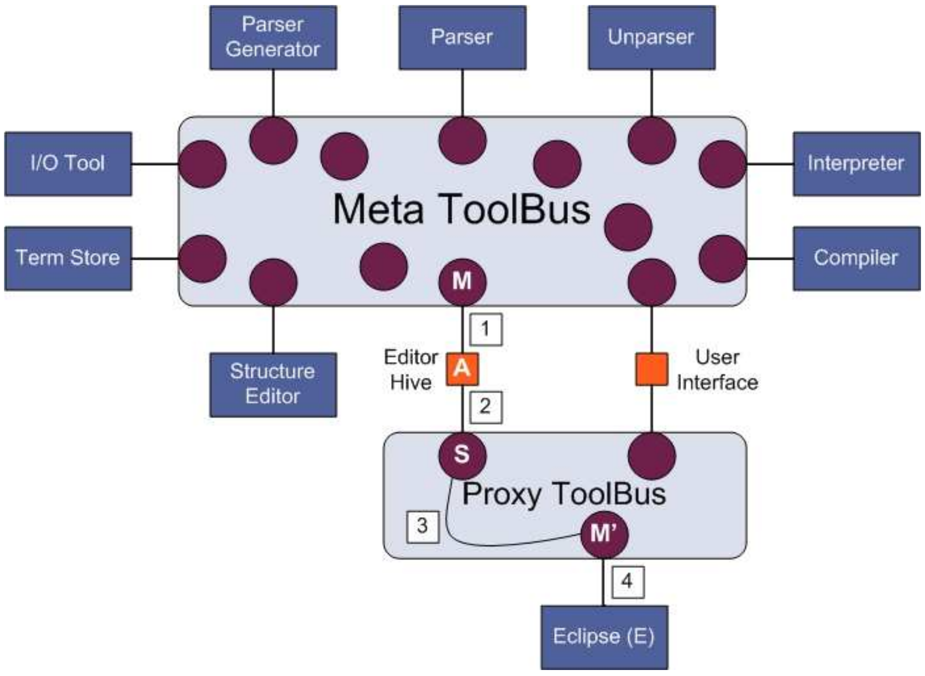


Figure 5.2: Eclipse architecture with Proxy ToolBus

```
    . snd-ack-event(S, Event)
    . snd-msg(Event)
  +
    rec-msg(ui(Message?))
    . snd-do(S, fun(Message))
  ) * delta
endlet
```

After a connection with the TransAdapter has been established two things can happen. The TransAdapter can forward a message from the Meta ToolBus which will be received as an event by the stub process. This event is then forwarded to the merged process. The second part handles incoming messages addressed to the userinferface process in the Meta ToolBus and forwards these to the TransAdapter connected to the user interface process.

The stub process has been designed as generic as possible. The only exception is that it has to provide a method to be able to address the right stub process for a message sent from the merged process. Therefore received messages has to be wrapped with an identifier (in this case `ui` for the *user interface* process).

5.3.2 Merging processes

Merging the *user interface* and *editor-hive* processes means taking out all messages sent to or received from the user interface and editor-hive tools and copying these in a new ToolBus script. Because the user interface and editors can be used in parallel the two sets of messages have to be implemented in parallel, too. If you have a set $S1$ of messages from the user interface and a set of $S2$

of messages from the editor-hive, the merged process becomes $S1||S2$ (see also Appendix A).

Sending a message to Eclipse has no effects on the implementation of the message in the merged process as long as every message from the originating processes is unique. If the user interface and editor-hive processes implemented the same messages a namespace should be introduced.

To be able to send a message from Eclipse to the Meta-Environment some changes had to be made to the original messages. The fact that Eclipse communicates with one process (the merged process, M' in Figure 5.2) means that these messages are not directed to the user interface or editor-hive process. Because all messages are unique we can distinguish the messages that have to be send to the user interface process from the messages that have to be send to the editor-hive process. After receiving a message from Eclipse it is wrapped with a message identifier and a process identifier.

The message identifier is used to distinguish return values from events. When a computation has been done a return value is sent back. By wrapping the message with a value string the TransAdapter knows it has to return a value to the Meta-Environment. Otherwise the message will be wrapped with an event string and the TransAdapter will send an event to the Meta-Environment.

The process identifier is used to distinguish between user interface and editor-hive messages. Eclipse does not distinguish between these messages and sends them all to the merged process. From here on it is known which message belongs to which process and by wrapping it with a `ui` (for *user interface*) or an `eh` (for *editor-hive*) string it is directed to the right stub process. The stub process will on its turn send the message to the TransAdapter which will deliver it to the Meta-Environment process it is connected to.

The merging of the user interface and editor-hive messages was a manual process, although it can be done automatically. By using ASF+SDF it is possible to collect all messages send to and received by a tool. For events and evaluation messages it is also needed to search for the return messages (acknowledgement of events and return values). Hereafter a namespace algorithm has to be used after which the sets are placed in parallel as described before.

5.4 TransAdapter

To be able to connect two TOOLBUSES a generic TransAdapter has been developed. The TransAdapter behaves like a tool that connects to two TOOLBUSES and therefore consists of a masquerade and a delegate side.

The masquerade side (indicated by a boxed 1 in Figure 5.2) provides a connection with the original TOOLBUS process. It therefore simulates the tool this process expects to be connected to. The toolname it has to provide to this process is one of the startup parameters of the TransAdapter. Because the TransAdapter connects to the TOOLBUS as a tool it receives a signature from the process it is connected to. This signature can be safely ignored as long as Eclipse (or any other tool that has to connect to more than one process) implements these messages. This can be ensured by merging these messages in the merged process and sending the signature of the merged process to Eclipse.

The delegate side (indicated by a boxed 2 in Figure 5.2) of the TransAdapter connects to the Proxy TOOLBUS. A stub process in the Proxy TOOLBUS will

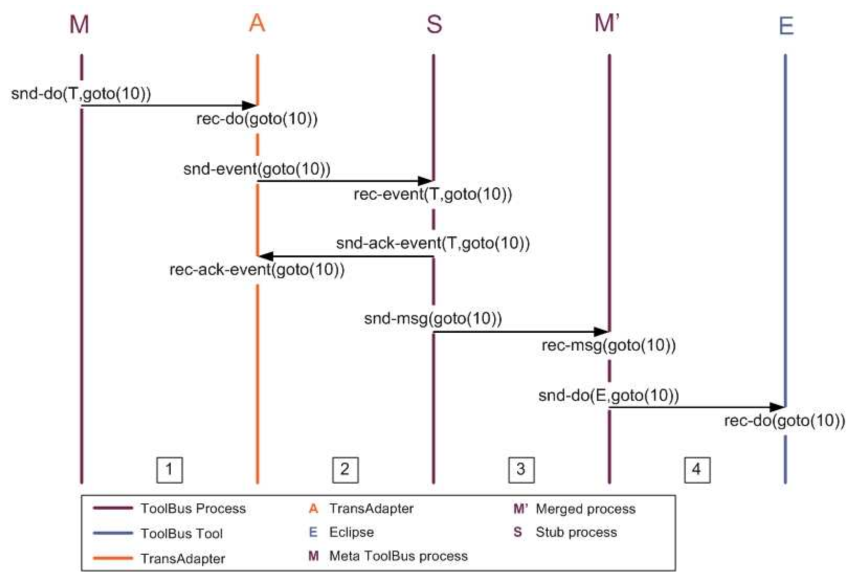


Figure 5.3: Message sequence chart of `snd-do`

forward all messages from and to the TransAdapter. A tool can only send events as initial message and therefore all messages received at the masquerade side of the TransAdapter will be forwarded as events to the stub process in the Proxy ToolBus via the delegate side.

Events and return values from Eclipse are received at the delegate side. The stub process sends them wrapped in a message identifier. Depending on this message identifier the TransAdapter will send an event or a return value via the masquerade side to the original ToolBus.

Below follows a description of the protocol used to send messages from the original ToolBus to Eclipse via the TransAdapter and Proxy ToolBus and events and return values from Eclipse to the original ToolBus.

5.5 TransAdapter protocol

In this section we will describe the protocol used for communication between Eclipse and the Asf+Sdf Meta-Environment via the TransAdapter and the Proxy ToolBus. Each ToolBus primitive `snd-do`, `snd-eval`, and `snd-event` will be described by a simple example.

5.5.1 snd-do messages

Suppose we have a SDF specification with some error. After invoking the parser with this specification we will receive an error from the parser and a linenumber where the error has occurred. To help the user fix the error we want the cursor to go to that specific line. Therefore we have to send a message with the linenumber to Eclipse. In Figure 5.2 that message follows the path from the boxed 1 to the boxed 4. Each step will be described using the message sequence chart in

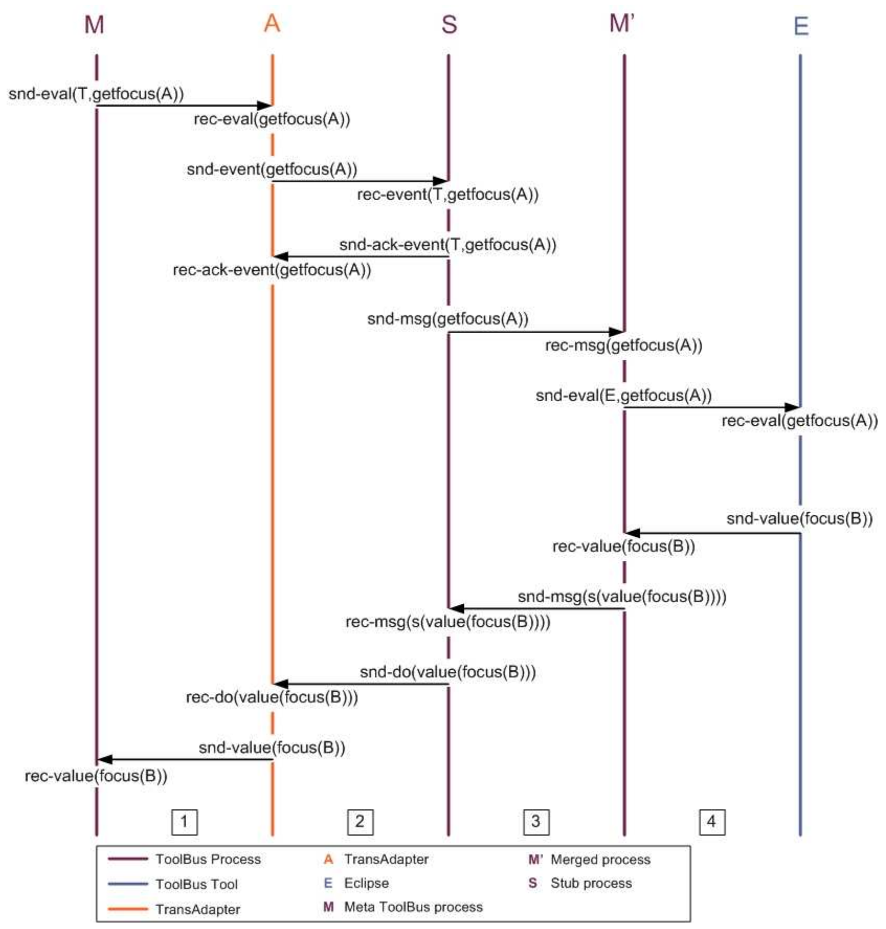


Figure 5.4: Message sequence chart of `snd-eval`

Figure 5.3. The letters and numbers in this figure refer to the corresponding letters and numbers in Figure 5.2.

Because the TransAdapter (A) has been implemented as a ToolBus tool, the first step is the same as for normal tools. The Meta-Environment process editor-hive (M) sends the message goto(10) to A via 1. The only way a tool can send a message to a ToolBus is using an event. A will therefore pack the goto(10) message in an event and sends it to the stub process P in the Proxy ToolBus via 2. P will acknowledge this event and forwards the message in the event to the merged process M' via 3. M' on its turn will send the goto(10) message to Eclipse (E) via 4 using a `snd-do` primitive. As can be seen the message sent to Eclipse by M' is the same one as sent by M.

5.5.2 Transformation of snd-eval

When a computation is needed a snd-eval message is sent to the tool that does the actual computation. After this computation the tool will send the result back. Using a TransAdapter (see Figure 5.4) the first part of a snd-

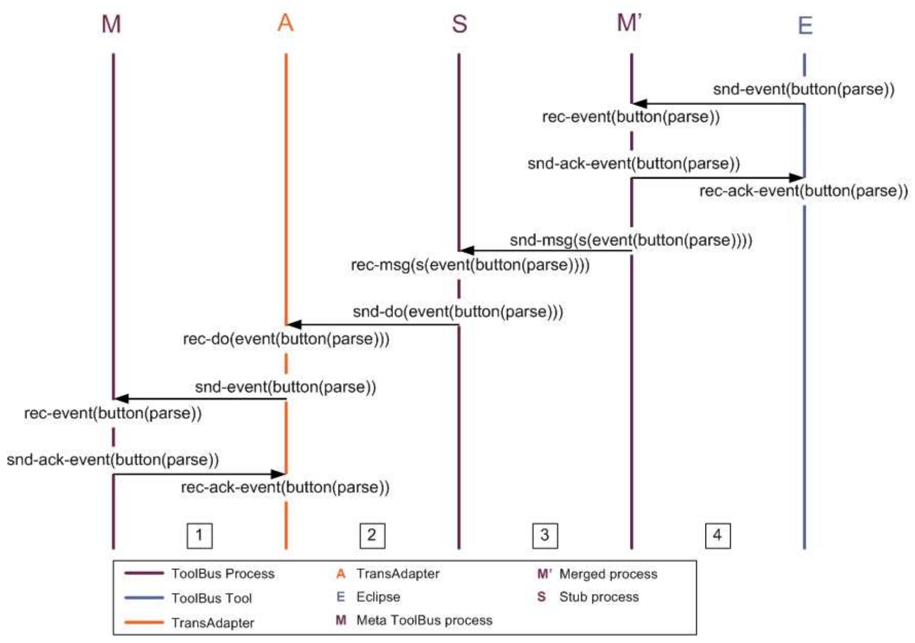

M
A
S
M'
E
snd-event(button(parse))
rec-event(button(parse))
snd-ack-event(button(parse))
rec-ack-event(button(parse))
snd-msg(s(event(button(parse))))
rec-msg(s(event(button(parse))))
snd-do(event(button(parse)))
rec-do(event(button(parse)))
snd-event(button(parse))
rec-event(button(parse))
snd-ack-event(button(parse))
rec-ack-event(button(parse))
1
2
3
4
ToolBus Process
ToolBus Tool
TransAdapter
A TransAdapter
E Eclipse
M Meta ToolBus process
M' Merged process
S Stub process

Figure 5.5: Message sequence chart of `snd-event`

eval message is handled the same as a snd-do. When the tool (E) returns the result it sends a snd-value message to M'. M' has to construct a message in such a way that the result value will be sent back to the process that asked for the computation. Therefore the received value will be wrapped in a message identifier and a process identifier. The message identifier indicates the sort of the message (i.e. an event or a return value). The message sort is in this case a returned valued. Process P is connected to the TransAdapter A and therefore can only send messages that can be received by a tool. The result value will be sent in a snd-do message. In the last step the TransAdapter will send the received value from P to M in a snd-value message.

5.5.3 Transformation of rec-event

While the abovementioned transformations were about sending messages to a tool, rec-event happens when a tool sends an event to its corresponding process. Figure 5.5 shows the message sequence chart of the `rec-event` primitive.

When E has send an event to M', this process will respond with an acknowledgement message. Hereafter it wraps the event with the message sort (event in this case) and the correct process identifier. The wrapped message will be received by P which passes the message to the TransAdapter A. The TransAdapter constructs an event and will send it to M, which will return an acknowledgement on its turn.

Chapter 6

Eclipse Meta Plugin

The Eclipse Meta Plugin is the implementation of the graphical user interface of the JFC/Swing Meta-Environment as an Eclipse plugin. This chapter covers all plugin elements and their internals in order of appearance.

6.1 Introduction

The Eclipse Meta Plugin is a plugin that enables the Eclipse user to make use of the features of the ASF+SDF Meta-Environment. Therefore all graphical user interface (GUI) elements provided by the Meta-Environment had to be implemented in Eclipse. In this chapter we will focus on the several GUI elements of the EclipseMeta Plugin and some of the shortcomings of Eclipse.

6.2 Plugin.xml

The starting point of an Eclipse plugin is the `plugin.xml` file, which describes in which way the Eclipse Platform will be extended and provides a way to make your own extension points. All GUI elements are listed in this file (e.g. Text Editor, Module Explorer, and Module Info), as well as registered file extensions (.sdf, .asf, and .trm), as well as the startup class which will be run when the plugin is started.

Unfortunately the `plugin.xml` restricts the dynamic behavior of the Eclipse Platform. On the one hand it is an easy way to determine which plugins are available at startup of the Eclipse Platform. On the other hand it defines the elements of the Eclipse plugin in a static way.

One of the important things of the Meta-Environment is the ability to define your own programming language. Most programming languages are associated with their own file extensions, which have to be registered in the `plugin.xml` if you want Eclipse to open the corresponding text editor. This makes that the user has to know beforehand which programming languages (and thus file extensions) he or she wants to use in the Meta-EnvironmentṪhose file extensions can then be added to the `plugin.xml`. After (re-)starting the Meta-Environment plugin Eclipse will use the preferred editor for the newly registered files.

6.3 Meta-Environment Perspective

After starting Eclipse and selecting the Meta-Environment Perspective, the perspective extension class as described in the `plugin.xml` file will be started. The PerspectiveFactory, as this class has been called, takes care of initiating the connection with the Proxy ToolBus described in Section 5.3 by instatiating an object of type UserInterface.

The UserInterface class takes care of setting up and handling all communication between the Proxy ToolBus and Eclipse. All important communication parameters are stored in an object MetastudioConnection. When an Eclipse component needs to communicate with the Meta-Environment it does this by creating a MetastudioConnection object and retrieving the needed connection parameters from the UserInterface.

After creating a UserInterface object, the PerspectiveFactory takes care of creating the Meta-Environment views. Views are the graphical user interface components visible to the user (e.g. Resource Navigator and Module Explorer, described in later Sections). These views are described in the `plugin.xml` file along with its class and unique name. This name is used to lookup the class that implements the view and can be used as a future reference (i.e. for other classes that need access to this view). Note that allthough text editors are graphical user interface components they are not views. To refer to all graphical user interface components including text editors the word *part* is used in Eclipse jargon.

The last thing the PerspectiveFactory takes care of is the creation of the Meta-Environment toolbar. This toolbar consists of a single button `Open Module` for opening a selected SDF module.

Because the PerspectiveFactory will be running as long as the perspective is not closed, the PerspectiveFactory is the place to implement accessors for builtin user interface elements. One of this builtin user interface elements is the statusbar. The statusbar is used in the Meta-Environment to display warnings, errors, and status messages. The statusbar is accessed through its manager, which can be retrieved from the PerspectiveFactory in the same way as every other user interface component. But there is something special with this statusbar manager. There is a different statusbar for views and for text editors. The text editor statusbar overlays the view statusbar, and therefore it is important to know which part needs access to the statusbar.

6.3.1 Resource Navigator

The first view created in the PerspectiveFactory is the Resource Navigator as shown in Figure 6.1. The Resource Navigator is one of the main parts of the Eclipse Platform. Each user has a workspace in which all project data (e.g. project metadata, files) are stored. The Resource Navigator represents the workspace in the form of a tree view. All projects and their corresponding files are listed in the tree pane of the Resource Navigator. To be able to use files in Eclipse these files have to be present in the workspace, because that is the only way Eclipse can get access to these files.

This setup leads to the following problem. One of the elements of the Asf+Sdf Meta-Environment is the library. Library modules are stored on a path in the installation directory of the Meta-Environment. To be able to

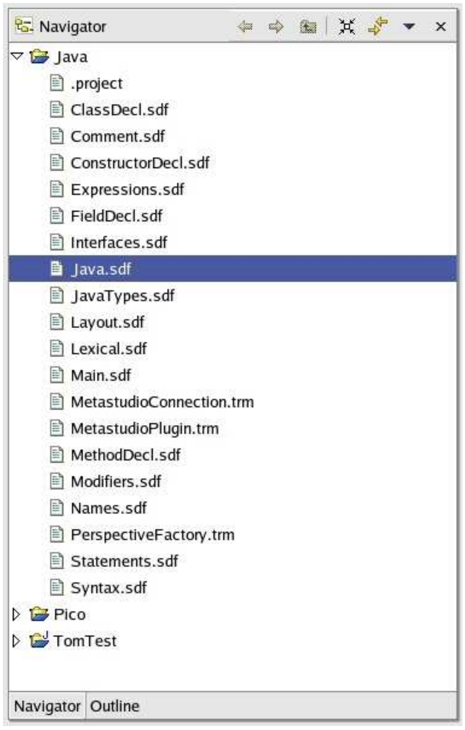


Figure 6.1: Resource Navigator

edit library modules, these modules should be imported in a separate project in the Eclipse workspace. After editing a module it has to be put back to its original location in the library. Another solution could be to redirect the location of the library in the Meta-Environment to the path of the workspace of the Eclipse installation.

When a user opens a SDF module in the JFC/Swing Meta-Environment a file open dialog will be shown. The module selected by the user will be loaded along with the transitive closure of its imports. All opened modules and libraries will be shown in a tree view. Clicking on a module results in a menu popping up, whereafter the user can choose to edit the syntax or equations of the module or opening a term over that module.

In Eclipse opening a SDF module is a little bit different. When the user opens a file in the Resource Navigator, Eclipse will open this file in the text editor registered with the file's extension. In most cases this is the behavior a user wants to have. But in the Meta-Environment you want the module to be loaded along with the transitive closure of its imports. Because the procedure of opening a file from within the Resource Navigator cannot be interrupted (yet) this results in a file being loaded in the Meta Plugin text editor, but not loaded in the Meta-Environment leaving the Meta-Environment in an inconsistent state. Therefore parsing the file for example results in errors or even worse: the Meta-Environment stops responding due to its inconsistent state.

The correct way of opening a module in the Meta Plugin is using the `Open Module` button in the Meta Plugin toolbar, which loads the module selected in

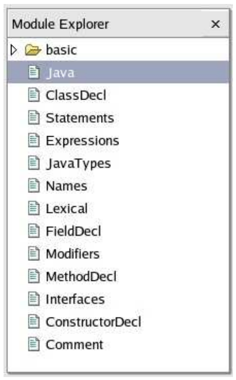


Figure 6.2: Module Explorer

the Resource Navigator and the transitive closure of its imports. The loaded modules are shown in the Module Explorer, which will be described in Section 6.3.2. When all the required modules have been loaded in the Meta-Environment a file selected in the Resource Navigator can be edited by opening it from within the Resource Navigator. A better way to edit files is to use the functionality of the Module Explorer. This ensures that all required modules are loaded.

6.3.2 Module Explorer

The Module Explorer (Figure 6.2) provides a tree view on modules loaded in the Meta-Environment. It looks like the Resource Navigator, but it does not show files but in stead shows the modules loaded by the Meta-Environment.

The Module Explorer consists mainly of two parts. The visual part is the treeview the user sees when using the Eclipse Meta Plugin. Data displayed in the visual part is stored in a tree datastructure. Changes to this structure are noticed by the visual part of the Module Explorer by using listeners. The visual part of the Module Explorer is never changed directly by others but only by changing the datastructure.

The Module Explorer view is implemented in the `ModuleExplorerPart` class. This class is also responsible for creating the tree datastructure. By instantiating an object of the `ModuleExplorerContentProvider` a listener is created that notifies the `ModuleExplorerPart` when the tree datastructure changes.

The `ModuleExplorerPart` invokes the `ModuleExplorerLabelProvider` to retrieve the names to be shown in the treeview from the tree datastructure and eventually adds icons to the different kinds of nodes of the tree (i.e. file icons or folder icons to distinguish between modules and directories).

Finally there is a mouse listener (`ModuleExplorerMouseListener`) which takes care of retrieving module information or showing a popup menu depending on the mouse button clicked. The retrieved module information will be displayed in the Module Info view.

The popup menu is context sensitive and shows the options that can be used with the selected file. To make the popup menu show the items corresponding to the file selected, Eclipse supports last minute changes to the items of the

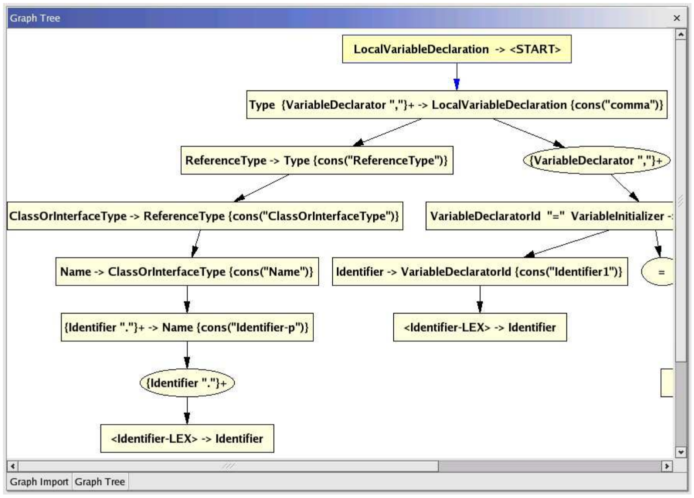

Graph Tree
LocalVariableDeclaration -> <START>
Type {VariableDeclarator ","}+ -> LocalVariableDeclaration {cons("comma")}
ReferenceType -> Type {cons("ReferenceType")}
{VariableDeclarator ","}+
ClassOrInterfaceType -> ReferenceType {cons("ClassOrInterfaceType")}
VariableDeclaratorId "=" VariableInitializer -
Name -> ClassOrInterfaceType {cons("Name")}
Identifier -> VariableDeclaratorId {cons("Identifier1")}
=
{Identifier "."}+ -> Name {cons("Identifier-p")}
<Identifier-LEX> -> Identifier
{Identifier "."}+
<Identifier-LEX> -> Identifier
Graph Import
Graph Tree

Figure 6.3: Graph View of import tree

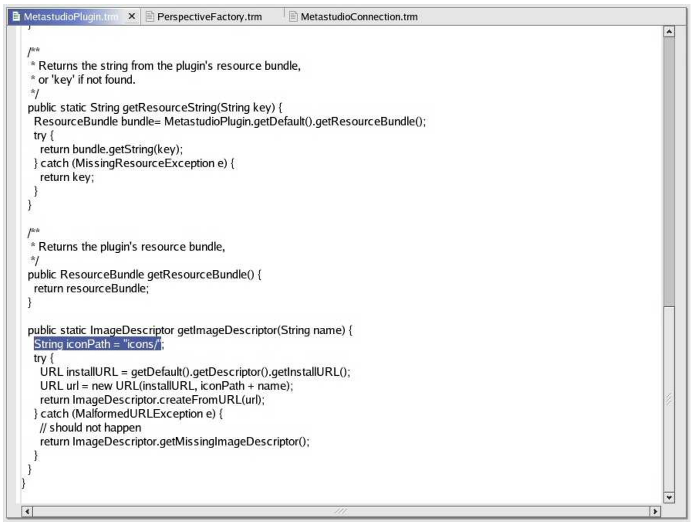
popup menu before showing it by the `menuAboutToShow` method. Problem is that menu items have to be known at the time the `menuAboutToShow` is called, but Eclipse has to ask the Meta-Environment for the right items. Therefore we stay in a loop as long as we have not received these items and set a boolean as we have received them. Unfortunately again this shows that although Eclipse is advertised as being dynamic, it lacks some dynamics in the implementation. A click of the second mouse button goes along with displaying a popup menu. The click itself cannot be intercepted, neither can a popup menu be displayed on command.

6.3.3 Graph Views

The Meta-Environment provides the user with two graph views. One of these views shows an import graph consisting of all modules loaded and their relationships to each other. This view has been implemented in Eclipse as the Import Graph View. Because the Import Graph View is a visualization of the Module Explorer, the same set of commands on modules is available for the user. Clicking a module in the graph reveals a popup menu containing the same items as the popup menu showing up when clicking a module in the Module Explorer.

The other graph the Meta-Environment provides is a parsetree of a selection in one of the editors. The Parse Tree View in the Eclipse Meta Plugin is the implementation of this Meta-Environment element. The Parse Tree View can only be shown when a specification has been parsed and some part has been selected.

The actual implementation of both views depends heavily on the graph datastructure already provided by the Meta-Environment. The graph datastructure is given in annotated data type (ADT) format and serves as input for APIGEN [5]. APIGEN generates an API for JAVA based on the given ADT. The graph datastructure ADT describes the structure of a graph including its nodes and edges. The generated API provides a way to extract the various elements of a graph from messages received from the Meta-Environment. After extracting



```
/**
 * Returns the string from the plugin's resource bundle,
 * or 'key' if not found.
 */
public static String getResourceString(String key) {
  ResourceBundle bundle= MetastudioPlugin.getDefault().getResourceBundle();
  try {
    return bundle.getString(key);
  } catch (MissingResourceException e) {
    return key;
  }
}

/**
 * Returns the plugin's resource bundle,
 */
public ResourceBundle getResourceBundle() {
  return resourceBundle;
}

public static ImageDescriptor getImageDescriptor(String name) {
  String iconPath = "icons/";
  try {
    URL installURL = getDefault().getDescriptor().getInstallURL();
    URL url = new URL(installURL, iconPath + name);
    return ImageDescriptor.createFromURL(url);
  } catch (MalformedURLException e) {
    // should not happen
    return ImageDescriptor.getMissingImageDescriptor();
  }
}
}
```

Figure 6.4: Meta Plugin editor

these elements the graph can be drawn on a canvas.

6.4 Editor

The editor used for editing modules and terms is based on the plain text editor already provided by Eclipse. This editor has been extended with methods for setting focus and moving the cursor to provide all functionality needed by the Meta-Environment.

The support for more than one editor in the JFC/Swing Meta-Environment is provided by the Editor Hive as described in Section 5.2.2. Eclipse supports multiple editors by default. Each editor in Eclipse has its own identifier, which is registered by an editor manager. The editor manager maps an Eclipse editor identifier to a Meta-Environment editor identifier.

Chapter 7

Conclusions and Future Work

The Eclipse Meta plugin sofar mimics the JFC/Swing user interface of the original Meta-Environment. While all basic functionality of the graphical user interface of the Meta-Environment has been implemented in an Eclipse plugin, there are possibilities of extending the Eclipse Meta Plugin with technologies provided by Eclipse.

7.1 Conclusions

Writing a plugin for Eclipse was far from trivial at first sight, but after a steep learning curve it has been possible to create a connection between the Meta-Environment and Eclipse.

Both Eclipse and the Meta-Environment have had their benefits from integrating both technologies. Eclipse has had the most benefits by making generic language technology available to the platform. Whereas other Eclipse plugins are focussed on one target programming language, the Eclipse Meta plugin makes it possible to create and edit language specifications and editing and parsing terms over these specifications in Eclipse.

Creating a new user interface for the Meta-Environment definitely added to the generic design of the Meta-Environment. One example is that the original JFC/Swing Meta-Environment used evaluation requests to ask for a filename using a file open dialog. It might be better to use a `snd-do/snd-event` pair, so the used user interface implementation can use its own method to open files. Another big advantage is that all cluttered windows are gone. There is one main window where all open modules and graphs recede.

While the Eclipse Meta plugin does make generic language technology available to Eclipse it is just a mimic of the original JFC/Swing environment. The following sections describe some of the future work that can be done to let the Meta-Environment benefit more from the integration with Eclipse.

7.2 Syntax Highlighting

While the Meta-Environment does not provide any form of syntax highlighting at this moment, it is possible to extend the Meta-Environment in such a way that it provides a basic form of syntax highlighting.

When a module has been loaded and parsed a table of keywords declared in the SDF specification of that module can be generated. By implementing a message in the editor hive that enables an editor to ask for the generated table, the syntax highlighting algorithm can be initialized. Editors that do not have syntax highlighting possibilities can easily ignore this message. Editors that can be programmed to use syntax highlighting, such as Eclipse editors, can ask for that table with keywords before displaying any text.

By extending the Eclipse editor with the already provided partitioner syntax highlighting in Eclipse can be enabled. The basic partitioner consists of a static table with keywords that have to be highlighted. But in stead of this static table a TOOLBUS message can be used to provide the partitioner with the keywords found in the SDF specification of the particular module. When opening a term over a module the editor first has to ask for the keyword table.

7.3 Help Facilities

The current Meta-Environment does not provide any help facilities, which could easily be provided by Eclipse via its built-in help functionality. Not only can a user manual be added, but also help on ASF+SDF modules or on grammatical structures of user defined programming languages.

Help facilities are provided in several ways:

- in a webbrowser (user manual or JavaDoc style help)
- in an Eclipse view (which has to be implemented yet)
- as tooltips when hovering with a mouse over keywords or grammatical structures

7.4 Other

Distributable plugin The Eclipse Meta Plugin is not available for download as a complete and easy to install plugin. To make the plugin available it is necessary to determine the packages the plugin depends on and make those available for distribution. Depending if the distribution has to include the source files, build path and plugin preferences have to be set.

Meta TOOLBUS and Proxy TOOLBUS startup At this moment the Meta TOOLBUS and Proxy TOOLBUS are started using a startup script. This script ensures the Meta TOOLBUS is initiated with port 8999 and the Proxy TOOLBUS with port 9000. The Eclipse Meta Plugin can be extended with a method starting these TOOLBUSES at other ports than the ones used in the script. By first finding two available ports and passing these ports in the startup parameters of the TOOLBUSES and the connecting TransAdapters the startup script can be eliminated and other ports could be used.

Bibliography

[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.

Appendix A

The merged process

The merged process as described in Section 5.3.2 is one of the processes receding in the Proxy ToolBus described in Chapter 5. For the readers who are known to the ToolBus scripting language the ToolBus script used for the Eclipse Meta Plugin to be able to handle messages from both the *user interface* and the *editor hive* is shown below.

As described in Chapter 5 the merged process consists of two parts concatenated by the parallel operator. Both parts are described as separate processes in their own ToolBus idef script (see Appendices B and C).

```
#include "stub1.tb"
#include "stub2.tb"
#include "user-interface.idef"
#include "editor-hive.idef"

tool user-environment is {}

process UE is
let
  UE : user-environment,
  Pid1 : int,
  Pid2 : int
in
  create(STUB1, Pid1?)
  . create(STUB2, Pid2?)
  . rec-connect(UE?)
  .
  (
    UI(UE)
  ||
    Hive(UE)
  )
endlet

toolbus(UE)
```

Appendix B

The user interface idef

The user interface idef handles all messages to and from the user interface part of the Meta-Environment. All messages to the Meta-Environment are directed by using a common wrapper for addressing the right TransAdapter (in the case of the *user interface* process this is *ui*).

```
process UI(UE: user-environment) is
let
  Id : term,
  Str : str,
  Pairs : list,
  Mods : list,
  FileName : str,
  Path : str,
  Extension : str,
  On : term,
  Graph : term,
  Info : list,
  LayoutedGraph : term,
  ModuleName : str,
  EnvironmentName : str,
  GraphId : str,
  ActionEvent : term,
  Tree : term,
  ButtonNames : str,
  ButtonList : list,
  Title : str,
  Answer : term,
  Type : term,
  Arguments : list,
  ModuleList : term
in
  (
    rec-msg(initialize-ui(EnvironmentName?))
    . snd-do(UE, initialize-ui(EnvironmentName))
    .
    (
      (
        rec-msg(add-status(Id?, Str?))
      . snd-do(UE, add-status(Id, Str))
    +
      rec-msg(add-statusf(Id?, Str?, Arguments?))
      . snd-do(UE, add-statusf(Id, Str, Arguments))
    +
      rec-msg(end-status(Id?))
      . snd-do(UE, end-status(Id))
    +
      rec-msg(error(Str?))
      . snd-do(UE, error(Str))
    +
      rec-msg(errorf(Str?, Arguments?))
      . snd-do(UE, errorf(Str, Arguments))
    +
      rec-msg(warning(Str?))
      . snd-do(UE, warning(Str))
    +
      rec-msg(warningf(Str?, Arguments?))
      . snd-do(UE, warningf(Str, Arguments))
    +
      rec-msg(message(Str?))
      . snd-do(UE, message(Str))
    +
      rec-msg(messagef(Str?, Arguments?))
      . snd-do(UE, messagef(Str, Arguments))
    ) * delta
  ||
    (
      rec-msg(display-graph(GraphId?, Graph?))
      . snd-do(UE, display-graph(GraphId, Graph))
    +
      rec-msg(delete-modules(Mods?))
      . snd-do(UE, delete-modules(Mods))
    +
      rec-event(UE, get-module-info(ModuleName?))
      . snd-ack-event(UE, get-module-info(ModuleName))
      . snd-msg(ui(event(get-module-info(ModuleName))))
      . rec-msg(module-info(ModuleName?, Info?))
      . snd-do(UE, module-info(ModuleName,Info))
    +
      rec-event(UE, clear-all)
      . snd-ack-event(UE, clear-all)
      . snd-msg(ui(event(clear-all)))
    +
      rec-event(UE, debugging(On?))
      . snd-ack-event(UE, debugging(On))
      . snd-msg(ui(event(debugging(On))))
    +
      rec-msg(new-graph(Pairs?))
      . snd-do(UE, new-graph(Pairs))
    +
      rec-event(UE, layout-graph(GraphId?, Graph?))
      . snd-ack-event(UE, layout-graph(GraphId, Graph))
      . snd-msg(ui(event(layout-graph(GraphId, Graph))))
  . rec-msg(graph-layouted(GraphId?, LayoutedGraph?))
  . snd-do(UE, graph-layouted(GraphId, LayoutedGraph))
+
  rec-event(UE, get-buttons(Type?, ModuleName?))
  . snd-ack-event(UE, get-buttons(Type, ModuleName))
  . snd-msg(ui(event(get-buttons(Type, ModuleName))))
  . rec-msg(buttons-found(Type?, ModuleName?, ButtonList?))
  . snd-do(UE, buttons-found(Type, ModuleName, ButtonList))
+
  rec-event(UE, button-selected(Type?, ModuleName?, ActionEvent?))
  . snd-ack-event(UE, button-selected(Type, ModuleName, ActionEvent))
  . snd-msg(ui(event(button-selected(Type, ModuleName, ActionEvent))))
+
  rec-event(UE, eclipse-open-modules(ModuleList?, Type?))
  . snd-ack-event(UE, eclipse-open-modules(ModuleList, Type))
  . snd-msg(ui(event(eclipse-open-modules(ModuleList, Type))))
+
  rec-event(UE, eclipse-open-initial-module(ModuleName?))
  . snd-ack-event(UE, eclipse-open-initial-module(ModuleName))
  . snd-msg(ui(event(eclipse-open-initial-module(ModuleName))))
+
  rec-event(UE, eclipse-edit-term-file(ModuleName?, FileName?))
  . snd-ack-event(UE, eclipse-edit-term-file(ModuleName, FileName))
  . snd-msg(ui(event(eclipse-edit-term-file(ModuleName, FileName))))
+
  rec-event(UE, button-selected(Type?, ActionEvent?))
  . snd-ack-event(UE, button-selected(Type, ActionEvent))
  . snd-msg(ui(event(button-selected(Type, ActionEvent))))
+
  rec-msg(show-file-dialog(Title?, Path?, Extension?))
  . snd-eval(UE, show-file-dialog(Title, Path, Extension))
  . rec-value(UE, file-name(FileName?))
  . snd-msg(ui(value(file-name(FileName))))
+
  rec-msg(show-question-dialog(Title?))
  . snd-eval(UE, show-question-dialog(Title))
  . rec-value(UE, answer(Answer?))
  . snd-msg(ui(value(answer(Answer))))
+
  rec-msg(clear-history)
  . snd-do(UE, clear-history)
+
  rec-event(UE, element-selected(ModuleName?, Tree?))
  . snd-ack-event(UE, element-selected(ModuleName, Tree))
  . snd-msg(ui(event(element-selected(ModuleName, Tree))))
+
  rec-event(UE, node-info(ModuleName?, Tree?))
  . snd-ack-event(UE, node-info(ModuleName, Tree))
  . snd-msg(ui(event(node-info(ModuleName, Tree))))
+
  rec-msg(update-list(ModuleName?, ButtonNames?))
  . snd-do(UE, update-list(ModuleName, ButtonNames))
)
*
```

```
      (
        rec-event(UE, quit)
        . snd-msg(ui(event(quit)))
      +
        rec-disconnect(UE)
      )
      . shutdown("MetaStudio exiting\n")
    )
  )
endlet
```

Appendix C

The editor hive idef

The *editor hive* idef handles all message to and from the Meta-Environment *editor hive*. All messages to the Meta-Environment are directed by using *eh* as wrapper to address the right TransAdapter.

```
process Hive(UE: user-environment) is
let
  ActionList : list,
  EditorId : term,
  Editor : str,
  FileName : str,
  Focus : term,
  FocusText : str,
  Location : int,
  MenuEvent : term,
  Message : str,
  MouseEvent : int
in
  (
    rec-msg(edit-file(EditorId?, Editor?, FileName?))
    . snd-do(UE, edit-file(EditorId, Editor, FileName))
    . rec-msg(set-actions(EditorId?, ActionList?))
    . snd-do(UE, set-actions(EditorId, ActionList))
  +
    rec-msg(reread-contents(EditorId?))
    . snd-do(UE, reread-contents(EditorId))
  +
    rec-msg(set-focus(EditorId?, Focus?))
    . snd-do(UE, set-focus(EditorId, Focus))
  +
    rec-msg(get-contents(EditorId?, Focus?))
    . snd-do(UE, get-contents(EditorId, Focus))
  +
    rec-msg(clear-focus(EditorId))
    . snd-do(UE, clear-focus(EditorId))
  +
    rec-msg(display-message(EditorId?, Message?))
    . snd-do(UE, display-message(EditorId, Message))
  +
    rec-msg(set-cursor-at-location(EditorId?, Location?))
    . snd-do(UE, set-cursor-at-location(EditorId, Location))
  +
    rec-msg(set-cursor-at-focus(EditorId?, Focus?))
    . snd-do(UE, set-cursor-at-focus(EditorId, Focus))
  +
    rec-msg(editor-to-front(EditorId?))
    . snd-do(UE, editor-to-front(EditorId))
  +
    rec-msg(kill-editor(EditorId?))
    . snd-do(UE, kill-editor(EditorId))
  +
    rec-event(UE, contents-changed(EditorId?))
    . snd-ack-event(UE, contents-changed(EditorId))
    . snd-msg(hive(event(contents-changed(EditorId))))
  +
    rec-event(UE, mouse-event(EditorId?, MouseEvent?))
    . snd-ack-event(UE, mouse-event(EditorId, MouseEvent))
    . snd-msg(hive(event(mouse-event(EditorId, MouseEvent))))
  +
    rec-event(UE, menu-event(EditorId?, MenuEvent?))
    . snd-ack-event(UE, menu-event(EditorId, MenuEvent))
    . snd-msg(hive(event(menu-event(EditorId, MenuEvent))))
  +
    rec-event(UE, contents(EditorId, FocusText?))
    . snd-ack-event(UE, contents(EditorId, FocusText))
    . snd-msg(hive(event(contents(EditorId, FocusText))))
  +
    rec-event(UE, editor-disconnected(EditorId?))
    . snd-ack-event(UE, editor-disconnected(EditorId))
    . snd-msg(hive(event(editor-disconnected(EditorId))))
  )
  * delta
endlet
```