
The Extension Points of The Meta-Environment

Paul Klint
Jurgen Vinju

2008-05-01 11:37:12 +0200 (Thu, 01 May 2008)

Table of Contents

| | |
|--|----|
| Introduction | 1 |
| Goals of Extensions | 1 |
| Types of Extensions | 2 |
| Step-by-Step Guide | 3 |
| Disclaimer | 4 |
| Adding a new Package | 5 |
| Setting-up the Build Environment with <code>reconf</code> | 5 |
| Configuring the Build Environment with <code>configure.ac</code> | 5 |
| Defining the Package with <code><packagename>.pc.in</code> | 6 |
| Standard Directories: <code>src</code> , <code>lib</code> , <code>tbscripts</code> , <code>spec</code> , <code>utils</code> , <code>scripts</code> | 7 |
| Defining the Build Process with <code>Makefile.am</code> | 7 |
| Maintaining Change Information in <code>ChangeLog</code> | 8 |
| Adding Standard Files: <code>COPYING</code> , <code>AUTHORS</code> , <code>README</code> , <code>NEWS</code> | 8 |
| Adding a Configuration Script | 8 |
| Adding new ToolBus Scripts | 12 |
| The relation between configuration scripts and ToolBus scripts | 12 |
| The standard ToolBus Actions | 14 |
| Error messages | 17 |
| Instantiating the generic module manager for a specific language | 17 |
| Creating a language specific structure editor with <code>syntax highlighting</code> | 17 |
| Implementing new Tools | 17 |
| Implementing GUI Extensions | 17 |
| Implementing a Start-up Shell Script | 18 |

Warning

This is work in progress; this document has not yet been reviewed and approved.

Introduction

How can the The Meta-Environment be adjusted or extended? This document describes what the extension points of The Meta-Environment are and how they are used in typical applications. Each type of extension points may have several goals.

Goals of Extensions

Extensions may have different goals (from less to more ambitious):

- Temporarily extending The Meta-Environment with user-defined menu options:
- Constructing a prototype of an IDE for a domain specific language.

- Automating repetitive or otherwise cumbersome GUI interaction for large software analysis or transformation projects.
- Connecting third-party command line tools.
- Permanently extending (or expanding) The Meta-Environment with language independent tools:
 - Adding version management support.
 - Adding generic visualizations.
 - Adding ...
- Permanently extending (or building on top of) the Meta-Environment with language specific tools:
 - Instantiating an IDE for a domain specific language.
 - Building an extension to the SDF Meta-Environment.
 - Building an extension to the ASF+SDF Meta-Environment.

Some of these goals can be achieved by adding some configuration information and some scripting, others require full-fledged software development.

Types of Extensions

There are various types of extension points to achieve these goals. For some of these goals, more than one type of extension point should be considered. We distinguish:

- New (extension) packages:
 - Depend on other (Meta-Environment) packages.
 - Use C and Java libraries and ToolBus scripts from the other packages.
 - Are separately compilable and distributable.
 - Preferably, the share the configuration, build and install interface of other Meta-Environment packages.
 - Introduce a start-up shell script for loading the Meta-Environment and the new extensions.
 - Contain implementations of the following extension points. *Explain this better: following?*
- Configuration scripts (i.e., files with the `.actions` extension):
 - Can add menu options to MetaStudio.
 - Can load other configuration scripts.
 - Can load additional ToolBus scripts.
 - Can add syntactic categories for syntax highlighting.
 - Can add workspace locations and library locations to the file dialog plugins of MetaStudio.
- ToolBus scripts (i.e., files with the `.tcb` extension):
 - Can use any existing ToolBus scripts that make generic tools of The Meta-Environment available.
 - Can connect any tool to the ToolBus.

- Can implement the behavior for new menu options.
- Can instantiate language specific, syntax highlighting, text editors.
- Can instantiate a language specific module manager and module browser.
- Any new tools (optional):
 - Are written in languages that we provide a stable ToolBus adapter for (C, Java, ASF+SDF), or
 - Provide their own ToolBus adapter.
- MetaStudio plugins written in Java (optional):
 - Can (dynamically) load new jars into the virtual machine of the MetaStudio GUI.
 - Can add and coordinate new tabs to the collection of viewable windows in MetaStudio.
 - Can add new menu options to MetaStudio.
 - Can send events and receive evaluation requests from ToolBus scripts.
- TIDE adapters (optional)
 - Can instantiate a language specific debugger with source browser and stack viewer.
 - Can add new visualizations to TIDE (written as MetaStudio plugins with a more specialized interface).

Step-by-Step Guide

Firstly, the source code for a particular extension needs to be organized in a new *package*. Secondly, this new package needs a *configuration script*. Then, we need new ToolBus scripts, new tools and new MetaStudio GUI plugins that implement the extension and coordinate its behavior with the rest of the system. Finally, if the extension represents an executable language, a debugging adapter may be included.

To make the above a bit more explicit, we describe the steps that are needed to build an IDE for a domain specific language on top of the Meta-Environment:

1. Create a new package with a GNU style build environment (**make**, **configure**, etc):
 - The package should at least depend on the meta package.
 - The package usually also depends on the aterm package or the aterm- java package.
 - Make directories for the source code new tools, ToolBus scripts, Configuration Files, GUI Plugins, and the like.
 - The implementation of a single extension may be spread over several packages (i.e., for separating data representation, coordination, computation and user-interaction from each other).

See Adding a new package.
2. Add a configuration script:
 - Import the standard meta.actions configuration script.
 - Add menu options with the names of ToolBus processes to be executed when the button is pressed, i.e., Compile program, Run program, etc.

See Adding a configuration script .

3. Add ToolBus scripts:

- For each menu option there needs to be at least one ToolBus process to handle it (i.e., `CompileAction`, `RunAction`).
- Dispatch the work to other ToolBus processes that already exist from the `meta` package (i.e., `ShowFileDialog`)
- Or, write new ToolBus processes that call possibly new tools (i.e., **`CompileDSLFile`**).
- For each new kind of file that can be edited by the IDE, a ToolBus script must be added that binds a language specific parse-table to the file extension, and binds editor specific actions to editor events (file saving, editing, menu options).
- If you are building an IDE for a modular language, you should instantiate the module manager by initializing it, and registering module event note listeners.

See Adding ToolBus scripts .

4. Implement (or generate) new tools:

- Write the tool in C, Java or ASF+SDF (i.e., a DSL compiler).
- Write a ToolBus process as an API for the tool.

See Adding ToolBus scripts.

5. Implement (in Java) the GUI extensions:

- Implement the `StudioPlugin` interface.
- Implement several `StudioComponent` interfaces (windows).
- Write a ToolBus process for dynamically loading the jar into the `MetaStudio`.

See Adding GUI extensions.

6. Implement a small shell script for starting the ToolBus with the appropriate scripts.

See Adding a start-up script .

For some extensions not all of the above steps are necessary. The IDE construction goal is the most demanding extension, while the other goals will require less effort.

Disclaimer

The Meta-Environment has many extension points, and some of them are still in flux. Still, this document provides an overview of the more stable extension points, and a brief explanation of how to use them. It is a first attempt at making these APIs more widely usable (meaning outside of the Meta-Environment development team).

When this document does not provide you with enough detailed information, please refer to the following ToolBus scripts:

- `meta/tbscripts/*.tb`
- `sdf-meta/tbscripts/*.tb`

- asfsdf-meta/tbscripts/*.tb
- any .idef files as found by the following command: `find . -name "*.idef.src"`

These files implement or directly control all extension points of the Meta-Environment, since they control all features of The Meta-Environment.

Note that all ToolBus scripts (.tb and .idef.src) files can be found online via the [online Meta-Environment API reference](#), although the syntax highlighting and linking does not work correctly for these files.

Adding a new Package

Extensions are to be separated from the rest of the Meta-Environment. This promotes configurability, maintainability, understandability, and independence of the extensions. All Meta-Environment packages have a similar package structure. For extensions, we advise to reuse this design. This will enable easy integration and deployment of your extension. For details regarding the build tools for Meta-Environment package see the [Meta-Environment Build Environment document](http://homepages.cwi.nl/~daybuild/daily-docs/project/build-environment.html) [http://homepages.cwi.nl/~daybuild/daily-docs/project/build-environment.html].

Every package corresponds to a module in a version control system, and contains at least the following files.

- reconf
- configure.ac
- <packagename>.pc.in
- Standard directories: src, lib, tbscripts, spec, utils, scripts
- Makefile.am
- ChangeLog
- Standard files: COPYING, AUTHORS, README, NEWS

Setting-up the Build Environment with reconf

For each new package you have to set-up the build environment. This is done as follows in the `./reconf` file:

```
#!/bin/sh
meta-build || ("Please make sure meta-build is in your PATH" && false)
```

This file bootstraps the build environment of a package. Make **reconf** executable and run the command `./reconf`, and the **meta-build** command runs the appropriate tools like **automake**, **libtool**, **aclocal**, and **autoconf** to set up the package's configure script and makefiles. These tools generate files, and create several soft links in the working directory for later use.

Configuring the Build Environment with configure.ac

A major issue in software development is how to configure the software to compile and run on as many computing platforms as possible. Typical issues are: Where is the C compiler? Which command line options does it accept? How do I run the Java compiler? How do I build libraries? Where should

the binaries or documentation be installed? And so on and so forth. We use an extended version of the **autoconf** system to achieve this. Consider the following example of `configure.ac`:

```
AC_INIT
META_SETUP
META_C_SETUP
META_JAVA_SETUP
AC_PROG_LIBTOOL
AC_OUTPUT
```

This file is used by autoconf to generate a configure script. The example file here prepares the package for C and Java code. The C code will use dynamically linked libraries. `configure.ac` must at least contain (in this order): `AC_INIT`, `META_SETUP`, and `AC_OUTPUT`. It may contain `META_C_SETUP`, or `META_JAVA_SETUP`, or both. Furthermore, any autoconf macros are allowed if your package needs more detailed configuration. Please read the autoconf manual [<http://www.gnu.org/software/autoconf/manual/autoconf-2.57/autoconf.html>] for details.

Note that the macros with the prefix `META_` take care of inter-package dependencies, see the `<packagename>.pc.in` file.

Defining the Package with `<packagename>.pc.in`

Having set-up the build environment, it is now time to define the properties of the new package such as its name, version, required packages and the like. Here is a skeleton that also contains some example information:

```
prefix=@prefix@
Maintainers=jurgenv@cwi.nl,economop@cwi.nl
Name:<packagename>
Version:<packageversion>
Description:<packagedescription>
Requires:<dependency1>,<dependency2>,<...>
Cflags:-I${prefix}/include
Libs:-L${prefix}/lib -l<libraryname1> -l<libraryname2> <...>
#uninstalled Libs:-L@abs_top_builddir@/lib -l<libraryname1> \
-l<libraryname2> <...>
#uninstalled Cflags:-I@abs_top_builddir@/lib
ToolBusFlags=-I${prefix}/share/<packagename>
JarFile=<packagename>.jar
Jars=${prefix}/share/${JarFile}
MainClass=<mainclassname>
#uninstalled UninstalledJars=@abs_top_srcdir@/${JarFile}
Packages=<javapackagename>
TestClass=<toptestclassname>
```

For a package with name `<packagename>` the package description file has the name `<packagename>.pc.in` after, of course, replacing `<packagename>` by the actual name, e.g., `sglr.pc.in` or `meta.pc.in`. This file declares all there is to know about a package. This information is typically used by the **reconf** script, by continuous integration toolkits, by package managers, and other deployment and product composition tools.

Before using the package definition file, it is instantiated by substituting all variables like `@prefix@`, `@abs_top_builddir@` by their actual, installation-dependent, values. The first line regarding the `prefix` variable is an obligatory implementation detail. The `#uninstalled` variables are used for compiling compositions of packages before they are installed (packages may refer to the source locations instead of the installed locations using this information).

The `Requires` field is very important. It declares the dependencies on other packages. The package definition file given above supports three different programming languages (C, Java and ToolBus scripts). If a package contains any of these, the associated variables are needed, otherwise not.

C packages that export libraries use `Cflags`, `Libs` and their uninstalled variants. This information is used by the makefiles of other packages. Namely, for each package they depend on those variables will be made available. Example: If your package depends on the `aterm` package, then you will have `ATERM_CFLAGS`, and `ATERM_LIBS` at your disposal in the makefiles.

Java packages export jar files, and have a test interface. From this information a full **ant** build configuration is created. This assumes that the source code is located in the subdirectory `src`. The other variables explain where the jar file should be installed and which packages should be included. The `MainClass` variable is used to properly instantiate a jar file manifest. This is also relevant for constructing correctly working plugins of the MetaStudio.

Packages containing ToolBus scripts explain where the scripts are installed such that dependent packages may set their search paths automatically.

Note that the implementation of the semantics of the `.pc.in` file can be found in the package `meta-build-env`. This package contains extension to **automake** and **autoconf**, written as shell scripts and m4 macros. Understanding, writing or changing `meta-build-env` is considered to be a black art, but it does allow us to remove a lot of code duplication, and improve consistency, in the infrastructure of packages.

Standard Directories: `src`, `lib`, `tbscripts`, `spec`, `utils`, `scripts`

The following naming conventions are used for source code subdirectories:

- `src` contains Java source code packages.
- `lib` contains source code for C libraries.
- `tbscript` contains ToolBus scripts.
- `spec` contains ASF+SDF specifications.
- `utils` contains C programs with a main that link the libraries in `lib`.
- `scripts` contain shell scripts.

Each of these directories, except for `src`, has a `Makefile.am` file. Most (but not all) Meta-Environment packages follow this convention.

Defining the Build Process with `Makefile.am`

Now we have reached the level of actual source files and their build instructions. We use automatically generated Makefiles that have been tailored with installation-dependent information. The recipe for generating each Makefile is given in an `Makefile.am` file. The suffix `.am` refers to the tool **automake** that is used to do the actual Makefile generation. The entire tool chain of **automake**, **autoconf**, **aclocal**, **configure**, etc. starts with the `Makefile.am` file. Here is an example:

```
include $(top_srcdir)/Makefile.top.meta

SUBDIRS = <subdirs>

ACLOCAL_AMFLAGS = -I .
```

Makefiles are notoriously sensitive to spaces. For instance, the exact number of spaces as given on the line `ACLOCAL_AMFLAGS` should be used (this is also due to tools such as **autoreconf** that scan the contents of `Makefile.am` files using rather brittle regular expressions).

The example above is relevant the top level directory of each package. For C packages and ToolBus packages there are additional `Makefile.am` files. For these kinds of packages, each subdirectory should contain its own `Makefile.am`. Please read the automake manual on how to write `Makefile.am` files for compiling C libraries and C programs. Each `Makefile.am` may include `Makefile.meta`, such that make rules and other utilities become available and can be reused. You are advised to look at the `Makefile.am` files of existing packages for inspiration. Inclusion of the `Makefile.meta` file is not obligatory however. *What do we illustrate with the following:*

```
include $(top_srcdir)/Makefile.meta
```

There is an exception, in case of a Java package. The *oplevel* `Makefile.am` should look like this:

```
include $(top_srcdir)/Makefile.java.meta
ACLOCAL_AMFLAGS = -I .
```

Java packages should have all their code in a subdirectory called `src`. This subdirectory may contain packages, but the corresponding directories do not need copies of a `Makefile.am` file. Note that by including `Makefile.java.meta` the package will use **ant** to compile your source code using the information from `<package>.pc.in`.

Maintaining Change Information in ChangeLog

It helps other developers to describe the changes you have made to source files. The file `ChangeLog` is the place for collecting this information. Usually, it contains verbatim copies of the entries in the version management system. Here is an example:

```
Fri Jun 30 21:43:04 CEST 2006 <jurgenv@cwi.nl>
* libmept/MEPT-utils.[ch]: reorganized this long file into a number
  of smaller files that each deal with a specific topic.
* libmept/*.c: added much documentation.
* libmept/*.h: fixed bug #774
* pt-support.pc.in: Bumped version to 0.1231
```

`ChangeLog` contains remarks on all changes to the source code of a package, in reverse order of date. Maintaining this information in this file makes it independent of a version management system. It should contain references to bug numbers when bugs are fixed. The `ChangeLog` file has a fixed format. Editors such as **gvim** and **emacs** have plugins for its syntax. `ChangeLog` files are maintained very well in Meta-Environment packages.

Adding Standard Files: COPYING, AUTHORS, README, NEWS

The `COPYING` file contains the license for the source code located in this package. This is usually the LGPL license for Meta-Environment packages. The `AUTHORS` file lists the authors email addresses. The `README` file should contain a brief description of the package. The `NEWS` file should list remarks on the evolution of the package on a high level of abstraction. Note that `README` and `NEWS` files are usually not maintained too well in Meta-Environment packages.

Adding a Configuration Script

Each instance of a Meta-Environment IDE has one toplevel configuration script. This script may import other scripts. At startup time, the configuration script is used to initialize the ToolBus with the right scripts, then after the ToolBus is started, the same script is used to initialize the configuration

manager tool. At run-time this tool provides tools and scripts with configuration information on-demand.

This is the syntax of a configuration file, as found in `sdf-library/library/basic/Configuration.sdf`. Note that currently the set of configuration options is fixed. Obviously, for real extensibility, this set needs to be opened. For now, this is what is offered to configure the existing generic components of The Meta-Environment.

```

sorts Configuration
context-free syntax
  "configuration" "(" "[" list:{Property ","}* "]" ")"
                                -> Configuration {cons("configuration")}

sorts Property
context-free syntax
  %% imports another configuration file:
  "import" "(" path:StrCon ")"          -> Property {cons("import")}

  %% loads another ToolBus script into the ToolBus:
  "load-script" "(" path:StrCon ")"      -> Property {cons("script")}

  %% extends the script path of the ToolBus:
  "script-path" "(" path:StrCon ")" -> Property {cons("script-path")}

  %% Registers an action with the selection of a certain
  %%user-interface element:
  "action" "(" "[" list:{ActionDescription ","}* "]" ","
              action:StrCon ")" -> Property {cons("action")}

  %% Couples a language name (SDF module name),
  %% with a file extension for use in file browsers:
  "extension" "(" language:StrCon "," extension:StrCon ")"
              -> Property {cons("extension")}

  %% Extends the list of workspaces with a new path:
  "library-path" "(" label:StrCon "," path:StrCon ")"
                  -> Property {cons("library-path")}

  %% Extends the list of workspaces with a new path,
  %% takes precedence over library paths when searching:
  "module-path" "(" label:StrCon "," path:StrCon ")"
                  -> Property {cons("module-path")}

  %% Registers the font properties of a certain text category name,
  %% for use with syntax highlighting:
  "text-category" "(" category:TextCategoryName ","
                  "[" map:{TextAttribute ","}* "]" ")"
                  -> Property {cons("text-category")}

sorts ActionDescription
context-free syntax
  %% describes a GUI element. The context is the name of the element,
  %% the event is may happen to it.
  "description" "(" context:TermCon "," event:Event ")"
                  -> ActionDescription {cons("description")}

sorts Event
context-free syntax

```

```

%% here is a list of things that may happen to a GUI element,
%% these are used for construction of the
%% GUI, as well as identifying which process is called finally:
"popup"                                     -> Event {cons("popup")}
"click" "(" "[" list:{KeyModifier ","}* "]" ","
        button:VirtualButton ")"           -> Event {cons("click")}

"icon" "(" title:StrCon "," path:StrCon ")" -> Event {cons("icon")}
"menu" "(" "[" labels:{Item ","}* "]" ")"  -> Event {cons("menu")}
"menu-shortcut" "(" "[" labels:{Item ","}* "]" ","
                 shortcut:ShortCut ")"     -> Event {cons("menu-shortcut")}

sorts Item
context-free syntax
  %% describe paths in menu's:
  "label" "(" name:StrCon ")" -> Item {cons("label")}

sorts TextCategoryName
context-free syntax
  %% There are three built-int text categories,
  %% and we can add our own:
  "focus"                                     -> TextCategoryName {cons("focus")}
  "selection"                                 -> TextCategoryName {cons("selection")}
  "normal"                                    -> TextCategoryName {cons("normal")}
  "extern" "(" name:StrCon ")" -> TextCategoryName {cons("extern")}

sorts TextAttribute
context-free syntax
  "foreground-color" "(" color:Color ")"
                        -> TextAttribute {cons("foreground-color")}
  "background-color" "(" color:Color ")"
                        -> TextAttribute {cons("background-color")}
  "style" "(" style:TextStyle ")" -> TextAttribute {cons("style")}
  "font" "(" name:StrCon ")" -> TextAttribute {cons("font")}
  "size" "(" point:NatCon ")" -> TextAttribute {cons("size")}

sorts ShortCut
context-free syntax
  %% The shortcut syntax is based on the Java virtual key interface
  "shortcut" "(" "[" list:{KeyModifier ","}* "]" ","
             key:VirtualKey ")" -> ShortCut {cons("shortcut")}

sorts TextStyle
context-free syntax
  "bold" -> TextStyle {cons("bold")}
  "italics" -> TextStyle {cons("italics")}
  "underlined" -> TextStyle {cons("underlined")}

```

Here is a part of an example configuration file called `standard.sdf.actions`:

```

configuration([
  import("__META__/share/meta/meta.actions"),
  script-path("__SDFMETA__/share/sdf-meta"),
  load-script("__SDFMETA__/share/sdf-meta/init-sdf.tb"),
  library-path("SDF Grammar Library", "__SDF_LIBRARY__"),
  library-path("ASF+SDF Library", "__ASF_LIBRARY__"),
  action([description(tree-panel, click)], "ShowAreaAction"),
  action([

```

```

description(
  studio-menubar,
  menu([label("File"), label("Exit")]))],
  "ExitAction"),
action([
  description(
    studio-menubar,
    menu-shortcut([label("Module"), label("New...")],
      shortcut([M_ALT,M_SHIFT],VK_N))),
  description(
    studio-toolbar,
    icon("New Module", "New24.gif")),
  "NewModuleAction"),
action([
  description(
    term-editor,
    menu-shortcut([label("Term"),label("Add Brackets")],
      shortcut([M_ALT,M_CTRL],VK_B))),
  "AddBracketsAction"),
action([
  description(
    syntax-editor,
    menu-shortcut([label("Module"), label("Edit"), label("Term")],
      shortcut([M_CTRL,M_SHIFT],VK_T))),
  "EditorEditTermAction"),
action([description(syntax-editor, popup)], "EditorPopup"),
action([description(equations-editor, popup)], "EditorPopup"),
action([description(term-editor, popup)], "EditorPopup"),

action([
  description(
    module-popup,
    menu([label("Close...")]))],
  "CloseModuleAction"),

text-category(normal,[foreground-color(rgb(0,0,0)),
  background-color(rgb(255,255,255)),
  font("Monospaced"),
  size(14)]),
text-category(focus,[foreground-color(rgb(0,0,0)),
  background-color(rgb(255,250,148))]),
text-category(selection,[foreground-color(rgb(255,255,255)),
  background-color(rgb(65,101,172))]),
text-category(extern("MetaKeyword"),
  [style(bold),foreground-color(rgb(123,0,82))]),
text-category(extern("MetaVariable"),
  [style(italics),foreground-color(rgb(0,0,255))]),
text-category(extern("MetaAmbiguity"),
  [style(bold),foreground-color(rgb(186,29,29))]),
text-category(extern("Todo"),
  [style(bold),background-color(rgb(255,255,255)),
  foreground-color(rgb(123,157,198))]),
text-category(extern("Comment"),
  [style(italics),foreground-color(rgb(82,141,115))]),
text-category(extern("Constant"),
  [style(italics),foreground-color(rgb(139,0,139))]),
text-category(extern("Variable"),
  [foreground-color(rgb(144,238,144))]),

```

```
text-category(extern("Identifier"),
               [foreground-color(rgb(255,69,0))] ),
text-category(extern("Type"),
               [foreground-color(rgb(255,127,36))] ),
text-category(extern("Error"),
               [style(bold),foreground-color(rgb(255,0,0))] ),
text-category(extern("Warning"),
               [style(bold),foreground-color(rgb(0,0,255))] ),
text-category(extern("Info"),
               [style(bold),foreground-color(rgb(0,255,0))] ),
text-category(extern("Fatal"),
               [style(bold),background-color(rgb(255,0,0))] )
])
```

Note that the configuration file contains names of ToolBus processes like `ShowAreaAction`, `ExitAction`, and others. These are described below.

Once a configuration file is constructed it will be used by the start-up script, and by an initial ToolBus process .

Adding new ToolBus Scripts

Since the ToolBus is the coordination infra-structure of the Meta-Environment, this is the centralized domain for extending and manipulating The Meta-Environment. The ToolBus scripts control each individual tool that is connected to the environment in every way: when they are started and stopped, how they are initialized, and when and how they are used to compute or store information.

This section explains which tools are configurable by writing ToolBus scripts and how.

The relation between configuration scripts and ToolBus scripts

Recall that the configuration manager provides a ToolBus API for the configuration parameters that have been loaded from the configuration scripts. This information is used by a number of generic tools in the Meta-Environment:

- `MetaStudio`: uses it to construct the menu bar and all of its contents.
- `editor-plugin`: uses it to construct editor specific additions to the menubar in `MetaStudio`, and for (language specific) popup menus in the editors, and for selecting font properties for syntax highlighting.
- `navigator-gui`: uses it to construct navigator specific additions to the menubar in `MetaStudio`, and for popup menus on the navigator tree.
- `graph-gui`: uses it to construct graph display specific additions to the menubar in `MetaStudio`, and for popup menus on graph nodes.
- `error-gui`: registers listeners for clicking on errors.
- ... in principle, this list is extensible with any tool interested in configuration information.

Let's focus on 'action' configuration parameters. Each action identifies for which component it is relevant, which menu option or other user-interface component it should be associated with, and which ToolBus process should be called when the action is activated (see configuration scripts).

The above named tools use the configuration information to build up menu's and register listeners to GUI events. They translate the events to ToolBus events, and then *inside the ToolBus script* such an

event is used to call the specific ToolBus process. Each component may pass different parameters to the configured process name.

For example, the editor-plugin provides an `EditorId`. Suppose there is a menu option called `CompileDSL` configured for a certain editor:

```
action([
  description(
    term-editor,
    menu-shortcut([label("DSL"),label("Compile")],
                  shortcut([M_ALT,M_CTRL],VK_C))],
  "CompileDSL"),
```

Which means that the process `CompileDSL` will be called when somebody clicks `DSL->Compile`, or types `ALT+CTRL+C`. Then, we need to implement the following ToolBus script:

```
process CompileDSL(EditorId : term) is
let
  ...
in
  ...
endlet
```

The `EditorId` term can be used to look up information about the editor, such as filename (see the standard ToolBus actions).

These are currently the ways processes are called from each component:

Table 1.1. Process call interface per configured component

| Component name | Process definition (first alternative) | Process definition (second alternative) | Reference to source code |
|----------------|--|---|--|
| MetaStudio | process <name> is ... | | meta/tbscripts/gui-utils.tb |
| editor-plugin | process <name>(EditorId : term) is ... | process <name>(EditorId : term, EditorType: term, Sort : str) | meta/tbscripts/editing.tb |
| navigator-gui | process <name>(ModuleId : term) is ... | | sdf-meta/tbscripts/sdf-module-actions.tb |
| error-gui | process <name>(Location : term) is ... | | meta/tbscripts/gui-utils.tb |

The above information is enough for anybody implementing actions for existing components.

When you implement a completely new component, you should instantiate a connection between defined actions in a configuration script, and the way a ToolBus process is called. The basic design pattern for this is as follows. We provide it here because the referenced source code is more complex. The real source code introduces one more layer of abstraction, by not letting a tool interface communicate directly with the configuration manager. This is also advised for any new components you develop, but we have left this out in the following example for clarity:

```
tool exampleTool is { ... }
process ExampleToolInterface is
let
```

```

T : exampleTool,
Id: term,
Event : term,
Events : list,
Action : str
in
rec-connect(T?)
. snd-msg(cm-get-events(example-tool-identifier))
. rec-msg(cm-events(Events?))
. snd-do(T, register-events(Events))
.
(
rec-event(T, event(Event?))
. snd-msg(cm-get-action(example-tool-identifier, Event))
. rec-msg(cm-action(Action?))
.
(
printf("Warning: process not found: %s\n", Action)
+>
Action()
)
+
...
)*
rec-disconnect(T) /* This process terminates when the tool disconnects */
endlet

```

The above skeleton can be extended by calling `Action` with different parameters, by adding more kinds of events, by adding different 'example-tool-identifiers' in the same tool, etc. Of course, any tool interface will probably have more `snd-do`'s and `snd-eval`s than is shown in the above skeleton.

The standard ToolBus Actions

The ToolBus scripts that implement The Meta-Environment already provide a wealth of functionality that you can reuse in configuration scripts. This functionality is separated in three layers: meta, sdf-meta and asfsdf-meta. The meta layer contains only language independent processes. The sdf-meta layer introduces SDF specific scripts. The asfsdf-meta layer introduces ASF specific scripts. Note that the sdf-meta layer depends on the meta layer, and the asfsdf-meta layer depends on the sdf-meta layer. To load them use any or all of the following include statements:

```

#include <meta.tb>
#include <sdf-meta.tb>
#include <asfsdf-meta.tb>

```

Including these files makes a number of things readily available:

- *tools* (such as in-output) with an interface that can be accessed via `snd-msg/rec-msg`.

These interfaces are found in files that have the `.idef` extension.

- *processes* that translate the `snd-msg/rec-msg` interface of selected tools to process definitions for your convenience.

These interfaces are found in files that have the `.tb` extension.

- *aggregated processes* that implement specific reusable scenarios using the above two interfaces.

These interfaces are found in files that end with `-utils.tb`.

- *specific processes* that are already tied to specific menu's or user actions.

These processes are found in files that end with `-actions.tb`.

So, for implementing a certain Action, you should start looking for functionality in `-utils.tb` files, if not found you should fall back to `.tb` files, if still not found, there is a chance of finding things in the `.idef` files of specific tools. Note that the `-actions.tb` files are usually too specific for reuse, but in some very simple cases they may come in handy.

The best resource for finding out about tools and processes is reading the source code of `.idef` and `.tb` files at the `Meta-Environment` online API reference.

| | | |
|---|--|--|
| | term) | module that the current editor is associated with. |
| asf | EditorRunAsfTestsAction(EditorId : term) The Extension Points of The Meta-Environment | Runs all ASF test equations for the module that the current editor is associated with. |
| Table 1.2. Standard ToolBus actions available for configuration scripts. | | |
| asf | PrettyPrintAction(EditorId : term, str) | ?? |
| asf | PrintModuleAction(ModuleId : term) | Starts the scenario for exploring a full ASF+SDF module to a certain text file. |
| asf | ReduceAction(EditorId : term) | Starts the scenario of rewriting the term in the current editor over the equations for the language that the current editor is associated with. |
| asf | RunAsfTestsAction(ModuleId : term) | Starts the scenario of running all test equations for the referenced module. |
| ?? | SetEditActions(Sid: term, Type: term, Name: str) | ?? |
| asf | ShowOriginAction(EditorId : term) | If the current editor is the result of a reduction, then this action takes the focus of the referenced editor, extracts origin information, and opens a new editor for displaying this origin. |
| asf | CompileModuleAction(ModuleId : term) | Compiles the module associated with this Module Id to an executable |
| asf | DebugReduceAction(EditorId : term) | Starts the scenario of rewriting the term in the current editor over the equations for the language that the current editor is associated with. The rewriter is started with a debugging interface (TIDE). |
| asf | DebugRunAsfTestsAction(ModuleId : term) | Starts the scenario of running all test equations for the referenced module. The rewriting is done with a debugging interface (TIDE). |
| asf | DumpEquationsAction(ModuleId : term) | Starts the scenario of collecting all equations for the referenced module and dumping them to a certain file. |
| asf | DumpEquationsParseTableAction(ModuleId : term) | Starts the scenario of exporting the parse table for parsing the .asf file of the referenced module to a certain file. |
| asf | EditorDumpEquationsParseTableAction(EditorId : term) | Starts the scenario of exporting the parse table for parsing the .asf file of the module associated to the referenced editor to a certain file. |
| asf | EditEquationsAction(ModuleId : term) | Starts an ASF equation editor for the referenced module. |
| asf | EditSyntaxAction(ModuleId : term) | Starts an SDF syntax editor for the referenced module. |

Error messages

Todo

Instantiating the generic module manager for a specific language

Todo

Creating a language specific structure editor with syntax highlighting

Todo

Implementing new Tools

- refer to ToolBus manuals, and ATerm manuals
- refer to -t flag of asfc compiler
- refer to generic adapter (does it still work?)

Implementing GUI Extensions

We refer to the javadoc documentation of MetaStudio. Each individual plugin needs at least:

- A ToolBus interface file (.idef.src). From this file the ToolBus adapter files are generated (...Bridge.java, ...Tif.java, and ...Tool.java)
- An implementation of the StudioPlugin interface. This contains the main method the plugin.
- An implementation of the StudioComponent interface. This is a window that will be registered with the studio by the plugin.

The Java code should be compiled to a .jar file. This .jar file will be loaded into the MetaStudio by the ToolBus. The following example illustrates how a jar file containing a plugin is send to MetaStudio for loading in the context of a certain class path:

```
process LoadMyPlugin is
  snd-msg(load-jar( "/home/mydir/myplugin.jar" , "/home/mydir/aterm-java.jar:/usr/
```

Make sure that you substitute the correct directories in this ToolBus script at installation time.

Note that the main class that is executed must implement the StudioPlugin interface, and it must be declared to be the MainClass in the <package>.pc.in file (if you are using the Meta-Environment package structure). If you are not using the automated package infra-structure, you must make sure that the .jar file contains an appropriate META-INF/MANIFEST.MF that contains the following line:

```
Main-Class: package.path.to.my.main.PluginClass
```

The ToolBus interface ideo script should look like the following skeleton:

```
tool myplugin is { }

process MyPlugin is
```

```

let
  T : myplugin
in
  rec-connect(T?)
  .
  (
    rec-msg(do-something)
    . snd-do(T, do-something)
  +
    rec-event(T, some-event)
    . snd-note(some-event-happened)
    . snd-ack-event(T, some-event)
  ) *
  rec-disconnect(T)

```

- How to implement the StudioPlugin interface
- How to implement the StudioComponent interface
- How to load the jarfile into the MetaStudio
- How to add menu options via the MetaStudio interface
- How to load StudioComponents at certain tab positions using the MetaStudio interface

Implementing a Start-up Shell Script

The startup shell script of a Meta-Environment based IDE looks like this:

```

#!/bin/sh

__TOOLBUS__ /bin/toolbus \
`__CONFIG_MANAGER__ /bin/configmanager -I -i __PACKAGEPREFIX__ /share/<configfi
`__CONFIG_MANAGER__ /bin/configmanager -s -i __PACKAGEPREFIX__ /share/<configfi

```

This script needs to be instantiated with the correct paths at installation time. Use a `Makefile.am` rule for this for example. The script simply starts the ToolBus and uses the **configmanager** tool (wrapped in backquotes for inline substitution) to generate the necessary command line options to the ToolBus. The following is the usage information from this command line tool:

```

usage: configmanager -[Is] -i <config-file>
       -I print ToolBus include path
       -s print ToolBus main scripts
       -i input configuration file [multiple]

```

See the ToolBus manual and ToolBus usage information for more information on ToolBus command line parameters.