
The Architecture of The Meta-Environment

Paul Klint
Jurgen Vinju

2008-05-14 11:13:57 +0200 (Wed, 14 May 2008)

Table of Contents

Motivation	2
Introduction	2
Stakeholders and Views	3
The Elements in Views	4
Rationale for the Architecture	7
Mapping between Elements and Views	8
The End-User View (USR)	10
Element Catalog	10
Context Diagram	11
Variability Guide	11
Architecture Background	11
Other Information	12
The Decomposition View (DCP)	12
Element Catalog	13
Context Diagram	13
Variability Guide	13
Architecture Background	13
Other Information	13
The Coordination View (CRD)	13
Element Catalog	14
Context Diagram	14
Variability Guide	14
Architecture Background	14
Other Information	15
The Detailed Coordination View (DCRD)	15
Processes	16
Element Catalog	21
Context Diagram	21
Variability Guide	21
Architecture Background	21
Other Information	22
The Data Layer View (DAT)	22
Element Catalog	23
Context Diagram	23
Variability Guide	23
Architecture Background	23
Other information	23
The Package View (PCK)	23
Element Catalog	24
Context Diagram	24
Variability Guide	24

Architecture Background	25
The Package Dependency View (DEP)	25
Element Catalog	27
Context Diagram	28
Variability Guide	28
Architecture Background	28
Further information	28
The Power-User View (PWR)	29
Element catalog	31
Context diagram	31
Variability guide	31
Architecture background	31
Other information	31
The PIMP Meta-Environment view (PMV)	31
Same high-level architecture	31
Know about IMP	31
IMP services implemented as language parametric ToolBus tools	32
Eclipse features reified as ToolBus tools	32
To Do	33

Warning

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

Motivation

The Meta-Environment is a powerful system that can be viewed from different angles. From the angle of the end-user who creates languages and tools using the graphical user interface, from the angle of the power-user who builds applications using the command line interface and the system libraries to the angle of the system developer who is maintaining and extending the system. For all these stakeholders we present architectural views that will help them to better understand the system and its possibilities.

Introduction

The Meta-Environment is a framework for *language engineering* (language design and implementation, source code analysis and source code transformation) consisting of:

- Syntax analysis tools.
- Semantic analysis and transformation tools.
- An interactive development environment.

The Meta-Environment is an open framework that

- can be easily extended with third-party components;
- can be easily tailored, modified, or extended;
- is supported by an open source community.

The Meta-Environment is a generalization of the ASF+SDF Meta-Environment that has been successfully used in a wide variety of analysis, transformation and renovation projects.

The Meta-Environment can be used for many different purposes, including:

- Parsing (new and old) programming languages, for further processing the trees, (e.g. COBOL, C, Java, PL/I, SQL)

- Analysis of source code (fact extraction, type analysis).
- Transformation of source code.
- Generation of source code.
- Design and implementation of domain-specific languages.
- Generation and rapid prototyping of IDE's (Integrated Development Environments) for programming languages and domain specific languages
- Generation of documentation from source code.
- Compilation of domain specific languages (DSLs).
- Formal description of the syntax and semantics of (programming) languages.

The *objectives* of The Meta-Environment can now be summarized as follows:

- To give non-expert users access to the above mentioned techniques and application areas.
- To provide expert users with a powerful tool suite.
- To create a flexible testbed for experimentation with new research approaches and technologies.

Stakeholders and Views

We identify the following stakeholders of The Meta-Environment:

- *End-users* that only use the system via the graphical user-interface (GUI).
- *Power-users* that use both the GUI, the command-line interface of the system as well as libraries and Application Programming Interfaces (APIs) provided by the system.
- *System developers* involved in the design, implementation, testing, maintenance and deployment of the system.

The following views on the system will be described:

- The end-user view (USR) describes the system as an interactive black box.
- The decomposition view (DCP) describes how the system is organized in components.
- The coordination view (CRD) describes the cooperation between components.
- The detailed coordination view (DCRD) describes the high level design of several main coordination processes within The Meta-Environment
- The data layer view (DAT) describes shows the relation between datatypes.
- The package view (PCK) describes the mapping between architecture elements and implementation packages.
- The package dependency view (DEP) shows how the various packages use each other.
- The power-user view (PWR) describes the system as a suite of commands and libraries.
- The PIMP Meta-Environment view (PMV) describes the difference of the MetaStudio based Meta-Environment and Eclipse-IMP based Meta-Environment.

Obviously, not all views are relevant for all stakeholders. Table Mapping between stakeholders and views shows the details.

Table 1.1. Mapping between stakeholders and views

Stakeholder	USR	DCP	CRD	DAT	PCK	PWR	DEP
End-user	d	s					
Power-user	s	d	s	s	s	x	s
System developer	s	d	d	d	x	d	x

Key: d = detailed information, s = some details, overview information, x = anything

The Elements in Views

For later reference, we give here a comprehensive overview of all language elements, data elements, and processing elements that occur in the various architecture views.

The Language Elements

The language elements are formalisms for specification, programming or scripting. Programs in these languages are read and written by humans and are intended to solve specific problems.

Abstract Data Type (ADT). Structural description of the interface of a component. Used by APIGEN to generate an Application Programmer's Interface (API) for the component.

ASF. Algebraic Specification Formalism. This a notation for describing rewrite rules and is mostly used for defining software analysis, fact extraction, and software transformation.

ASF+SDF. The combination of the formalisms ASF and SDF. ASF+SDF can describe both the syntax of a language and the operations on that language (checking, execution, analysis, transformation).

ASF+SDF library. A collection of elementary datatypes (lists, tables, etc.), language grammars (C, COBOL, Java, SDF, etc.) and utilities.

ASF+SDF modules. The ASF+SDF modules describe:

- the syntax of the base language,
- the functions that can be applied to base language programs.

Note that the ASF+SDF modules may define several base languages at the same time.

Input term. A text that conforms to the syntax defined in the ASF+SDF modules. This includes:

- the syntax of the base language (e.g., C, Java, Boolean expressions, a domain-specific language),
- the functions that can be applied to base language programs (e.g., typecheck, extract facts, compile, remove unused methods).

The input term can be freely edited and is checked for syntactic correctness before any function is applied to it. It is possible to simultaneously edit different input terms in different base languages.

Output term. A text that describes the result of applying a function to a program in the base language. Note that this text conforms to the syntax R, where R is the result sort of the function that was

applied. With Java-to-Java transformation the result sort will be Java. When no function is applied, the output term is identical to the input term.

Rscript. A small scripting language for defining relational expressions. Used for the analysis of facts extracted from software.

Warning

Rscript is not yet integrated in V2.0.

SDF. Syntax Definition Formalism. A notation for describing the grammar of programming and application languages.

Tscript. The script that describes the cooperation between components in a ToolBus-based application.

The Data Elements

The data elements are formats for representing domain-specific data. The corresponding data are written and read by programs.

AsFix. ASF+SDF Fixed format. The dataformat used to represent parse trees. AsFix is a specialized view on ATerms. Important features are:

- The AsFix format is a full parse trees that contains all the original layout and comments from the original source code program that was parsed.
- The AsFix format is self-descriptive: each subtree contains information about the exact grammar production that has been used to parse the text that has resulted in that parse tree.
- The AsFix format does not contain source code coordinates per se, but a separate tool (**addPosInfo**) can easily compute these coordinates and add them to the parse tree in the form of annotations.

ATerm. Annotated terms. A dataformat used for the internal representation of all data. Distinguishing features are:

- ATerms are language-independent and can be processed by programs in any language.
- ATerms can be annotated with auxiliary information that does not affect the tree structure.
- ATerms preserve *maximal subterm sharing*. This means that common parts of the data are not duplicated but shared. This leads to considerable size-reduction of the data.

Box. Intermediate representation of the prettyprinter. A parse tree is first converted to a box term that includes all desired formatting directives (alignment, font and color directives, and the like). Next, the box term is converted to various output formats (plain text, HTML, etc.).

Compiled specification. It is possible to compile the ASF+SDF into a very efficient executable form. Compiled specifications can be used via the command line interface of The Meta-Environment by power users.

Graph. Data format to represent graphs. Used as representation of import graphs and parse trees that are displayed in the GUI.

Location. Data format to describe locations in source code.

Summary. An error or message summary. A dataformat for the internal representation of errors and messages. Summaries are produced by checker and compilers and are used by the GUI.

Parse table. A parse table is an efficient representation (ATerm) of the base language as defined by the ASF+SDF modules and enables efficient parsing.

Parse tree. Tree-structured representation (in AsFix) of a text that has been analyzed by a parser.

The Processing Elements

The processing elements are "active" parts of the system that usually transform inputs to outputs. Inputs and outputs may both be machine or human readable or writable.

APIGEN. Application Programming Interface (API) generator. Given a datastructure definition (in the form of an Abstract Data Type), APIGEN generates C or Java code to access that datastructure. Internally, the datastructure is represented as ATerm.

ASF checker. ASF checker performs static checking on ASF definitions.

ASF compiler. The ASF compiler transforms ASF specifications to C code. That C code is compiled and linked with a run-time library. This leads very efficient execution of specifications.

ASF interpreter. The ASF interpreter takes the equation sections from the ASF+SDF specification and applies them to the parsed input term. It produces another parse tree as output.

ASF operations. ASF operations provides all operations to read and modify ASF definitions.

ASF+SDF checker. The ASF+SDF checker checks an SDF definition for compatibility for use with ASF, and also checks some production attributes that are specifically interpreted by ASF rewriting engines. It is an extension of the SDF checker.

Configuration manager. The configuration manager handles user settings and preferences.

Debugger. The debugger allows a step-by-step execution of the rewrite rules defined in ASF+SDF specifications.

Errors & warnings. Errors & warnings pinpoint any errors in the ASF+SDF modules or input terms.

Graphical user interface (GUI). The graphical user-interface (GUI) gives end-users access to the system's functionality. It is a "sovereign" user-interface that occupies the complete desktop window and provides functionality like:

- Opening, editing and closing ASF+SDF modules.
- Opening, editing, reducing and closing input terms.
- A graphical and tree-structured display of the import relations between ASF+SDF modules.
- A graphical display of parse trees.
- Error and progress indications.

Module manager. The module manager is responsible for all information related to the ASF+SDF modules that reside in the system.

Parse table generator. The parse table generator takes syntax sections from the ASF+SDF specification and converts them to a parse table to be used for the parsing of terms.

Parser. The parser takes a parse table (as produced by the parse table generator) and text (as provided by a text editor) as input and produces a parse tree as output. Any errors are shown in the error display of the GUI.

Prettyprinter. The prettyprinter converts parse trees to text. The prettyprinter uses default rules to insert layout in a parse tree so that its corresponding text is presented in a uniform way. Optionally, the ASF+SDF specification may contain formatting rules that can replace this default behaviour.

Warning

The prettyprinter is not yet fully integrated in V2.0.

Sisyphus. A system for continuous integration that rebuilds the system after each change that is committed by a developer.

SDF operations. SDF operations provides all operations to read and modify SDF definitions.

SDF checker. SDF checker performs static checking on SDF definitions.

Structure editor. A syntax-directed editor that closely cooperates with the text editor. It is mostly used for syntax-directed navigation through the text. The structure-editor does not appear as such in the GUI but all its functionality is visible through the text editor.

Term store. The term store contains all parse tables, parse trees and other intermediate data that is generated during execution of The Meta-Environment. This includes:

- The parse tree for each ASF+SDF module.
- The parse tree for each parsed input term.
- The parse tree for each generated output term.

Text editor. The text editor allows text editing on ASF+SDF modules and input terms. Multiple editors may be opened; each appears as a tabbed window in one of the panes of the GUI.

The Meta-Environment. The architecture of The Meta-Environment (or just "the system") is the primary object of study of this document.

ToolBus. The ToolBus coordination architecture enables the flexible and controlled combination and orchestration of software components. It is used as backbone for The Meta-Environment. The ToolBus has the following characteristics:

- Components (or *tools* in ToolBus parlance) can be written in different programming languages.
- Components can be running on different machines.
- All interactions between components are regulated by a ToolBus script (or Tscript for short) that is executing in the ToolBus. Tscript is a concurrent language that allows the definition of parallel processes, messaging between these processes and interaction between processes and tools.

Rationale for the Architecture

The following quality attributes guide the design of the architecture of the Meta-Environment.

Usability. We want to support our users at all levels of experience in as many ways as possible:

- Through a GUI that automates many of the tasks involved in language engineering. For instance, the user only sees a grammar and the fact that it can be used for parsing but is unaware of the fact that parse table generation is going on behind the scenes. The same is true for rewriting, formatting and the like.
- Through a command line interface that allows the batch-like combination of components.

- Through libraries that package specific functionality that can be called directly from user-written programs.
- Through ToolBus components that can be included in interactive applications.

Adaptability. The Meta-Environment is at the same time a framework for building applications in the area of language engineering and a testbed for doing research. This implies that we want to be able to combine existing components in new and flexible manners and to enable the easy integration of new components.

Performance. The Meta-Environment is not primarily intended for large scale industrial or commercial use. However, the performance should be such that experiments with real source code of industrial size (millions of lines of code) are possible.

Testability. It should be possible to test individual components separately.

We pay less attention to other important quality attributes such as *availability* and *security*, but we expect that they will become more important as the system evolves.

Mapping between Elements and Views

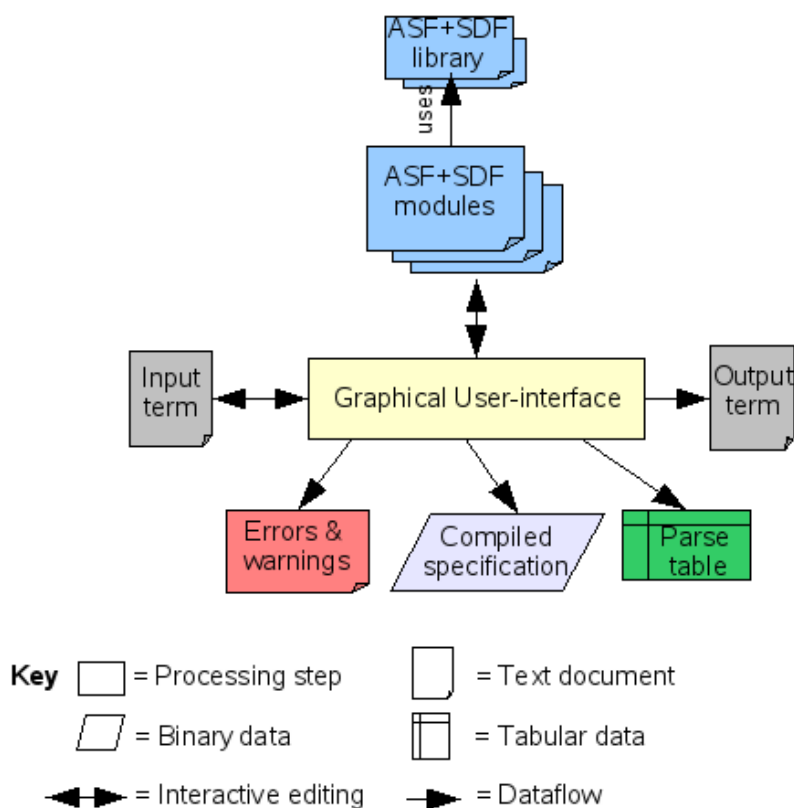
Figure Mapping between elements and views keeps track of which elements are used in which views.

Element	VSP	DCP	CRD	DAT	PCK	PWR	DEP
APIGEN						x	
ASF checker			x			x	
ASF compiler			x			x	
ASF interpreter		x	x			x	
ASF operations			x			x	
ASF+SDF library							
ASF+SDF modules	x	x				x	
ASF+SDF checker			x			x	
Compiled specification	x					x	
Configuration manager							
Errors & warnings	x					x	
Graphical user interface (GUI)	x		x			x	
Input term	x	x				x	
Module manager			x			x	
Output term	x	x				x	
Parse table generator		x	x			x	
Parse table	x	x				x	
Parser		x	x			x	
Parse tree		x				x	
Prettyprinter			x			x	
SDF checker			x			x	
SDF operations			x			x	
Structure editor			x			x	
Term store							
Text editor			x			x	
The Meta-Environment	x	x	x		x	x	x
ToolBus			x			x	

The End-User View (USR)

Figure End-user view of The Meta-Environment shows how simple The Meta-Environment is for an end-user. Via a graphical user interface, the user can edit ASF+SDF modules that describe the syntax and semantics of some user-defined language, let's say Java. The ASF+SDF modules will define the grammar of the Java language (this is actually already provided in the ASF+SDF library) and some functions on Java programs such as extracting facts or describing transformations on the Java code. The user can also edit an input term, that is a text that conforms to the definitions in the ASF+SDF specification (say, a Java program with a transformation function applied to it). Via the user-interface the user can now activate the application of the equations in the ASF+SDF specification to the input. The answer is an output term that will consist -- in this example -- of a transformed Java program. Of course, errors may occur and they are shown in the GUI. As an aid for power-users some additional information like compiled ASF+SDF specifications and parse tables can be exported.

Figure 1.1. End-user view of The Meta-environment

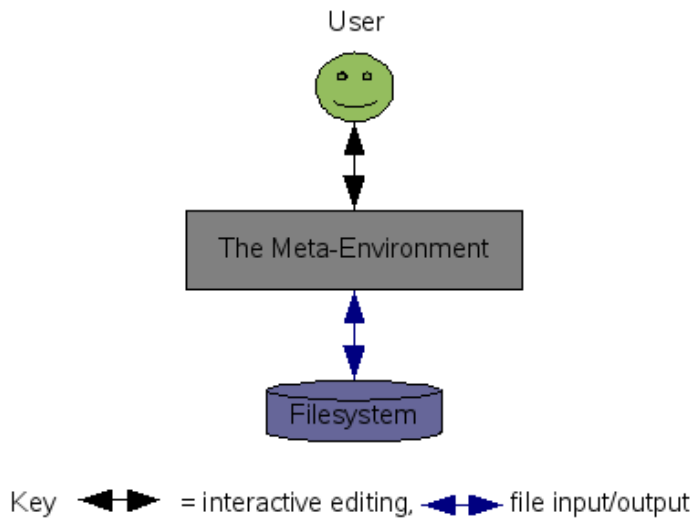


Element Catalog

ASF+SDF library, ASF+SDF modules, Compiled specification, Errors & warnings, Input term, Output term, Parse table.

Context Diagram

Figure 1.2. Context end-user view



Variability Guide

The user has the following options for variability:

- The filesystem directory from which The Meta-Environment is started (using the command **asfsdf-meta**) determines the working directory to be used during the session with the system.
- The file configuration file `meta.actions` (in the current working directory) can be used to extend the behaviour of the system. This is further explained in Extension Points of The Meta-Environment. (*LINK*)
- The GUI provides some options to change the overall graphical appearance (skinning) and the way in which graphs are displayed.

Architecture Background

Design Rationale

In the end-user view, we try to make life as easy as possible for the user and try to automate whatever we can. For instance, when a user reduces the input term to the output term, he needs not be aware of the fact that first a parse table has to be generated and that the input term is then parsed. The fact that after editing an ASF+SDF module, it may be necessary to regenerate the parse table is also implicit. The end-user is shielded from all internal representations.

Analysis of Results

The Meta-Environment mostly satisfies the design goal of usability. Some known usability weaknesses are:

- The error messages generated by the parser are poor.
- There is no support for finding errors in syntax definitions.
- Certain features found in other IDEs are still missing (e.g., autocompletion, context-dependent help menus, and others).

- The user documentation is still not yet complete and up-to-date, but we are working on this ;-)

Assumptions

The current system assumes that the user can write ASF+SDF specifications without further automatic support. It turns out that writing syntax definitions is not so easy and that some automatic support is welcome.

Other Information

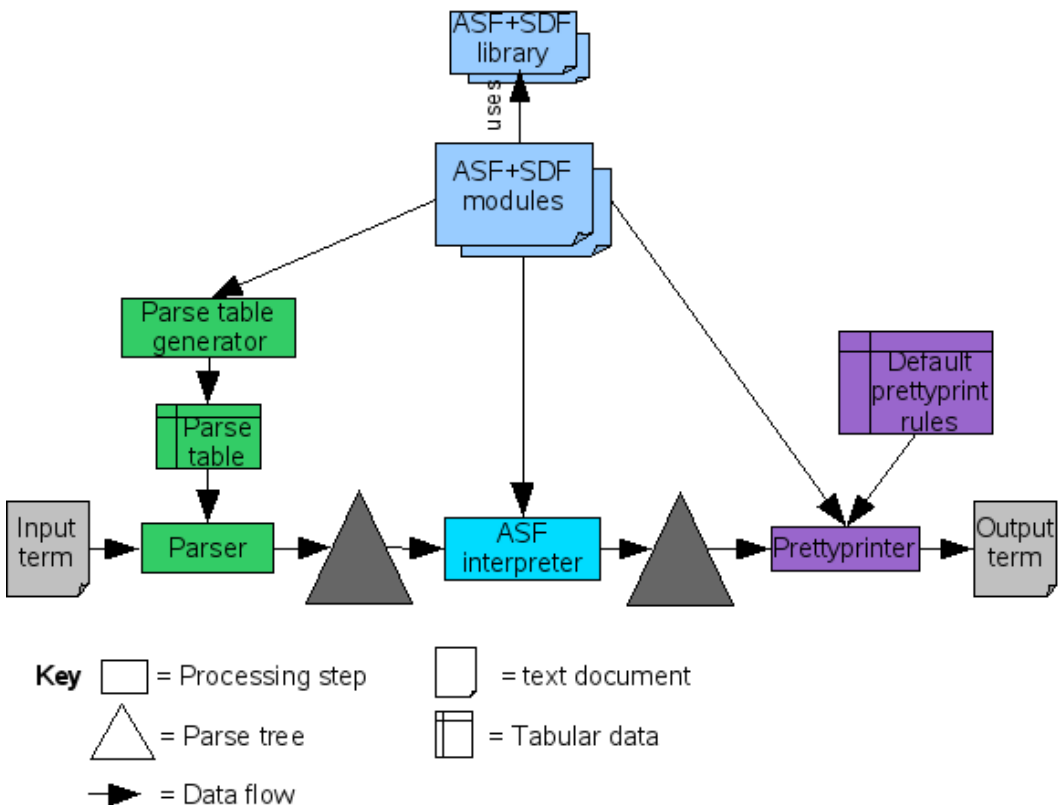
This view does not provide any details on the actual operation of the user-interface. Some details are helpful:

- The user-interface is a *sovereign application*, i.e., it can occupy the full window of the desktop.
- The user-interface provides a graphical view on the import relations between ASF+SDF modules and the parse trees of selected modules or terms.
- The user-interface provides a debugging view on the execution of ASF+SDF specifications.
- This view only shows one input term and corresponding output term. In reality, several input terms may be edited and be reduced to the corresponding output term.

The Decomposition View (DCP)

The decomposition of the system in parts is shown in figure The decomposition view. It emphasizes the parts that are needed to transform an input term to an output term. Not shown are the interaction with the user-interface, the precise processing of the ASF+SDF modules, and the generation of compiled specifications and parse tables.

Figure 1.3. The decomposition view



Element Catalog

ASF interpreter, ASF+SDF library, ASF+SDF modules, Default prettyprint rules, Errors & warnings, Input term, Output term, Parser, Parse table, Parse table generator.

Context Diagram

Identical to the context diagram of the end-user view.

Variability Guide

In addition to the ASF+SDF modules that determine the characteristics of the languages that are being defined, the user can override the default pretty printing rules for each language by giving extra ASF+SDF modules containing specialized pretty printing rules.

Architecture Background

Design Rationale

This system decomposition is rather traditional. It distinguishes parser, semantic operations and prettyprinter. For reasons of uniformity, pretty printing rules are also specified in ASF+SDF.

Analysis of Results

None.

Assumptions

The following assumptions have been made:

- Full parse trees (e.g., parse trees containing all structural and textual information, including layout and comments, of the original source text) are the best intermediate representation for programs. This is motivated by the need in software renovation projects to perform source code transformations that preserve as much of the original source text as possible.
- All extra information that has to be attached to parse trees can be expressed as annotations to that parse tree. This regards source code coordinates, information about constructors, and the like.

Other Information

Various simplifications have been applied in this view:

- The graphical user-interface is not shown.
- We abstract from the internal processing of ASF+SDF specifications.
- The various internal formats that play a role are not shown.
- The compilation of specifications is not shown.
- The generation of parse trees (for external) use is not shown.

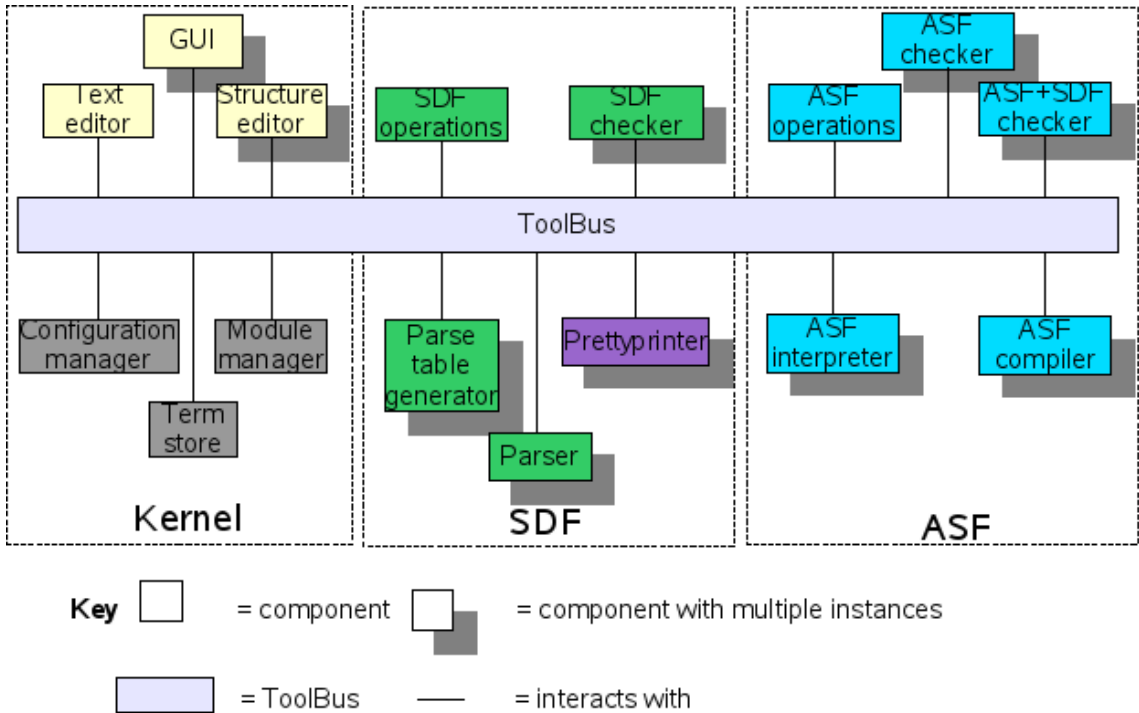
The Coordination View (CRD)

Figure The coordination view shows the interaction between the components of The Meta-Environment via the ToolBus middleware layer. This view can be separated in three parts:

- Kernel: components providing basic functionality, like user-interface and editing.
- SDF: all components related to SDF and syntax analysis.
- ASF: all components related to ASF and term rewriting.

The connections between the components are established by the Tscript that is executing in the ToolBus. Note that multiple instances may exist of some components. For instance, a separate instance of the text editor exists for editing different terms or modules.

Figure 1.4. The coordination view



Element Catalog

ASF checker, ASF compiler, ASF operations, ASF interpreter, ASF+SDF checker, Default prettyprint rules, Errors & warnings, GUI, Input term, Module manager, Output term, Parser, Parse table, Parse table generator, Prettyprinter, SDF checker, SDF operations, Structure editor, Text editor, ToolBus.

Context Diagram

None.

Variability Guide

This architecture was designed with variability in mind. By replacing the Tscript in the ToolBus, largely different applications can be built.

Architecture Background

Design Rationale

The rationale for this architecture is to:

- Decouple the components as much as possible.
- Enable the composition of components written in different languages.
- Enable the execution of components in a distributed fashion on more than one computer.
- Enable variability by allowing the composition of existing and new components in new ways.

Analysis of Results

It is fair to say that the above design goals have been achieved. Components become less aware of their context and are more easily composable. Unavoidably, the Tscript has to describe all these compositions and becomes the central information hub. In large applications like The Meta-Environment these Tscripts become quite large and complex, but they are fortunately the only location where this composition information resides.

Assumptions

The assumptions in this view are as follows:

- Components can be written in different programming languages.
- The information exchange between components and ToolBus is by way of ATerms.
- For all relevant languages adapters exist to connect programs written in those languages to the ToolBus.
- It is possible to execute components on different, connected, physical computers. In some situations firewall settings may prevent this.

Other Information

- The ToolBus uses TCP/IP sockets to implement the connections with tools.

The Detailed Coordination View (DCRD)

As the coordination view explained, the main coordination architecture of The Meta-Environment is provided by the ToolBus. The ToolBus runs a ToolBus script containing many processes that coordinate the features of The Meta-Environment. This Detailed Coordination View highlights these processes. These are the major coordination processes:

1. Editor management - the process of editing files, which includes triggering several tools that provide feedback to the user.
2. Module management - the process of managing modules, builds and dependencies, which includes triggering tools that provide feedback to the user.
3. Configuration management - the processes of managing the configuration of The Meta-Environment in terms of the visible menu options and other keyboard and mouse user interactions.
4. Transaction management - a generic mechanism to define critical sections that lock the use of certain available data and tools to prevent racing conditions

Note that the main coordination processes of The Meta-Environment are designed to be language independent. They reside in the meta package which is a part of the kernel layer. Other packages that depend on the meta package, such as sdf-meta (SDF layer) and asfsdf-meta (ASF+SDF layer), bind

configuration parameters and specialize the generic functionality to obtain programming language specific IDE's. The way that the language specific layers extend the kernel layer is governed by this architecture description.

Processes

The common scheme of all ToolBus script processes is that there are 'listener', 'util', 'action' and 'other' processes. They are grouped into files with a straightforward file naming scheme.

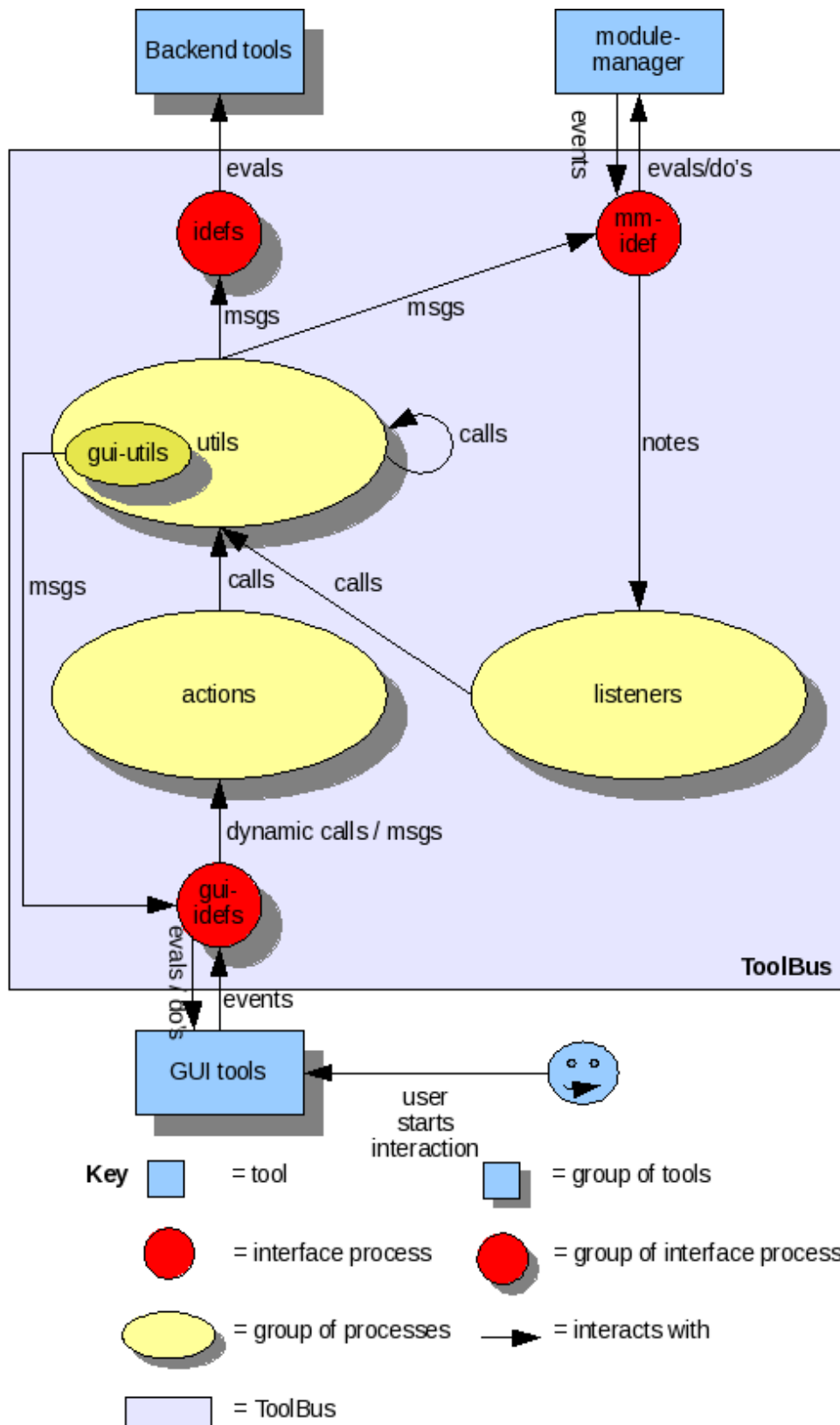
- Idefs are ToolBus processes that manage the connectivity between a specific tool with the ToolBus. Each idef describes exactly when and how other processes may interact with its tool, and when and how the tool interacts with the ToolBus. The abbreviation "idef" stands for interface definition.
- Listener are processes that catch notes that are send by mostly the idef of the module manager and sometimes by utility processes. The listeners are a facade only, they forward to util processes. Listener files begin with a language name or a feature name and end with "-listeners.tb"
- Actions are processes that are triggered directly by a user interaction, which forward to util processes. Files begin with a language name or a feature name and end with "-actions.tb"
- Utils are the implementation processes, they do the actual work. Files begin with a language name or a feature name and end with "-utils.tb"

Note that communication can be done in ToolBus scripts in three ways: sending messages, sending notes and calling processes. In general the rules communication of The Meta-Environment management processes are:

- tool processes - which are single processes each associated with a single tool, also called 'idefs' - send and receive synchronous *messages*, except for manager tools which can also broadcast asynchronous *notes*.
- *notes* are received by listener processes, which *call* util processes
- utility processes *call* eachother (synchronously), and send messages to tool processes, where it is tried to wrap single messages to tools in simple process definitions

The following picture provides an overview of the kinds of processes and their interaction with eachother and tools. The module manager is just a tool, but it is very central to the scheduling of all work in the Meta-Environment, so it has been singled out. See also the following section on module management.

Figure 1.5. A high level view of groups of processes and their control flow in The Meta-Environment



Editor management

The goal of the editor manager is to manage the complexity of appearing and disappearing editors. An editor in the Meta-Environment is the pair of a visual text editor, implemented in Java, and a structural tree 'editor' implemented in C. Editors are data-sources for several (interactive) usage scenarios in the Meta-Environment. Because they appear and disappear when the user opens or closes files, this is where the first racing conditions start to occur in The Meta-Environment. The editor-manager pre-dates the transaction management system and implements it's own kind of transactions to lock resources that are associated with editors.

There are four ToolBus tools involved:

- editor-manager - a C tool that maintains a registry of editing processes. Each process is identified with a unique id. The creation and garbage collection of editors is managed here. Also this tool provides the binding between a text-editor instance and a tree that is stored in the structure-editor.
- structure-editor - a C tool that stores parse trees with editor id's and supports a number of interesting queries on these trees. The tool actually is not an editor in the sense that it never changes a tree
- addPosInfo - the editing infra-structure highly depends on the fact that all parse trees are passed through this tool before anything else happens
- editor-plugin - a Java program that uses the Swing editor kit to provide visual editing of source files

Most of the coordination work is implemented in ToolBus scripts:

- editing.tb - provides generic and reusable processes
- <lang>-editor-utils - binds the parameters of editing.tb processes and makes them language specific. This is where it is decided what happens when the user clicks the mouse or activates a pop-up menu.

There is always at least one ToolBus process per file edited, which stays alive until the last transaction as declared using the editor-manager has ended. This means that although an editor is not visible on the screen anymore, the data stored with it in Java and in the structure-editor is available until the transaction ends.

Module management

The goal of module management is to trigger functionality (utils) on source files, such as .sdf or .asf files, at the right time and to provide the user with a consistent view of the state of the source files that are edited by the user in the environment.

- The module-manager tool connects to the Bus and maintains an administration of which modules are currently loaded and what their dependencies are
- The module-manager maintains a list of key-value attributes (ATerms) for each module. Each key has a namespace and a name. Any T-script can set or change the attributes of a module.
- The module-manager broadcasts notifications on the ToolBus whenever a module is added or removed, a dependency is added or removed, or an attribute is changed.
- Language specific scripts in the SDF or ASF+SDF layer register 'event rules' with the module-manager. The event rules define how some attributes on modules are set automatically by propagating information across the dependency graph between modules. This event rule mechanism can deal with cyclic dependencies. A single change of an attribute on a module, can thus spawn a number of changes on other modules, all changes are broadcasted as independently on the ToolBus.
- Event listeners (note receivers in T-script) listen to the changes in the module manager and trigger appropriate utils. Some utils change the attributes of modules, triggering other utils via other listeners.

There is one central attribute that is used a lot by The Meta-Environment called "status". A particular view on The Meta-Environment module management process is that it is a collection of state machines, one per each module:

- The status attribute of a module can have a fixed set of values, called the "states" of the module.
- Specific processes are tied to states by the event listeners. When the status attribute changes to a particular state, the processes that belong to that state are executed using listeners
- Processes that are finished set the next state of modules.
- Like any other attribute, the state attribute can also be updated by event rules, such that modules change to a different state without ToolBus script intervention. Examples of common state values are
 - unknown - for modules that somebody depends on, but nothing is known about yet
 - identified - for modules that have been located on disk and have been given a name and a path attribute
 - error - for modules that contain errors that need to be solved by the user
 - child-error - for modules that depend on modules that have errors that need to be solved by the user
 - complete - for modules that are in an executable state

See fig. the detailed coordination view on the role of the module manager in its environment. As an example, consider the scenario in which a user saves an SDF file in The Meta-Environment:

1. The editor is a GUI tool, which will send an event to the ToolBus, containing for which language (SDF), and which file the event is meant for
2. The idef of the editor will receive the event and call the action process that deals with SDF file saving
3. The action process will first find out to which module the file belongs to by querying the 'path' attribute in the module manager. Then it will call the appropriate utility process to do the work.
4. The utility process coordinates everything that needs to be done for the *current* module. It works by activating tools such as the in-output tool which can save files to disk. When the work for file saving is done, the utility process will also update the status attribute in the module manager by sending a message to the idef of the module manager
5. The idef of the module manager relays the updated status attribute value to the module manager
6. The module manager receives the updated status attribute value and - starting from the initial change and using the dependencies between modules - it will change the status attributes of other modules. The changing is governed by several attribute update rules.
7. For every changed status attribute for a specific module, the module manager sends out an event
8. The event is caught by the idef of the module manager, which broadcasts the updated status attribute values information using a ToolBus note
9. Several listener processes may be subscribed to such status notes. One example is the SDF module parser listener. This listener will trigger the SDF parsing utility process for a specific module.
10. The SDF parsing utility process, and many other processes triggered by other listeners are now working in parallel as in Step 4. Some utility processes may send messages to GUI tools, for example the new tokens to highlight are send to the editor tool from Step 1.
11. Steps 4 to 10 will repeat until a fixed point is reached and no more attributes in the module manager change, or no more listeners react to the changes in attributes.

Note

Utility processes should generally deal with only one module at a time. The distribution of work for several modules should be scheduled by attribute event rules and the notes that are sent out by the module manager because of these rules. If a utility process asks for the dependencies of a module, then this is a sign of code that goes against this architecture.

Note

After the initial attribute change, the listeners will trigger many instances of many processes which will run mostly in parallel and asynchronously. This is when racing conditions can occur and transaction management starts playing its role.

Configuration management

Although preferences and other kinds of start-up and run-time configuration parameters have been factored out in the config-support and configuration-manager packages, they have not been generalized or parameterized. These two packages support a fixed set of parameters that is specific for the ASF+SDF Meta-Environment. Which configuration options are supported is defined in `sdf-library/library/basic/Config.sdf`. Although this functionality resides in the core of the Meta-Environment, it is explicitly linked with features in ASF+SDF. This needs to be fixed in the future, because it is hard to know where and how to extend this implementation to add new configuration parameters.

Configuration management works as follows:

- At start-up time, the Meta-Environment ToolBus script reads two `.actions` files from disk. One contains the default bindings of the parameters of all tools in The Meta-Environment (i.e. `asfsdf-meta/tbscripts/standard.asfsdf.actions`), the other is a `.actions` file in the user's workspace directory. The files are registered with the configuration-manager, which resolves import statements and maintains a hierarchy where user preferences have precedence over default system preferences.
- Several utility processes distribute the configuration information on demand to the tools that need it. These processes first query the configuration manager for information (using `cm-...` messages), and then forward this information to the appropriate tools. For example, when an editor is started, the editing scripts ask the config-manager what kind of menu's need to be shown. This information is forwarded to the editor. The editor tool will receive an ATerm encoding of the menu structure, which is interpreted to produce the right menu's.
- Some configuration parameters define *what* needs to be done and *when*. Since this is a responsibility of the ToolBus scripts there is a bridge. Such configuration parameters carry the names of processes to run. The idef of a tool that interprets such a configuration parameter will call the appropriate process using a *dynamic process call*. For example, the MetaStudio knows which menus to show. Each menu option will have an associated Process name. When the menu option is selected by the user the MetaStudio sends an event to its idef, which calls the process using a dynamic call.

Transaction management

Due to the design of the module manager and the scheduling of processes via listeners, many things in the Meta-Environment happen asynchronously. Still the processes share common data sources which they read and write to. The transaction management system is based on simple locking. For each shared resource the `transactions.tb` script allows to start a TransactionManager process. A lock is obtained by sending a message to this unique process, which only allows the next process to obtain a lock until the previous has released it.

This system is completely incremental and demand driven. Processes are not obliged to obtain a lock to obtain access to shared resources. Instead, when the programmer knows that his process could lead to racing conditions, he decided to first obtain a lock to the resource. Processes that are robust against

changing values, such as progress views, do not lock the resources they read from, which improves efficiency by allowing parallelism.

Element Catalog

Errors & warnings, GUI, Module manager, Structure editor, Text editor, ToolBus, Editor Manager, Configuration Manager, Transaction Manager

Context Diagram

See Coordination view

Variability Guide

For each of the main coordination processes there are different variability stories

Editor management

Each editor has a type name to identify the language it is for. It therefore can behave differently by getting different configuration parameters, and show different menus and do different things for mouse actions. Also, each type of language has a different editing process, which extends the functionality of `editing.tb` with listeners that act differently for every language.

Module management

The generic nature of the module manager makes it easy to adapt and extend it to new programming languages.

1. Add a new 'name space' to add a new language
2. Add a new 'event rule' to propagate new information across the dependency graph
3. Add a new note listener to add new automatic features to the environment.

Configuration management

The configuration manager has all parameters more or less hard wired. They are generated from the `basic/Config.sdf` file. The manager also knows about specific parameters, taking care for example that menu options are treated in order.

Transaction management

The transaction mechanism is totally orthogonal and can be extended with new kinds of locks using a ToolBus script one-liner. Note however that adding new requests for existing locks always has the possibility of introducing deadlock. This occurs naturally when requests for locks of the same resource are nested in a single scenario. It can also happen if ToolBus scripts accidentally become dependent on each other in a circular fashion. Care must be taken to fully understand which resources are needed and when.

In other words, adding locks to the ToolBus scripts of The Meta-Environment is not a modular activity. One must fully understand all dependencies between processes and shared resources to make sure this is done correctly.

Architecture Background

Design Rationale

In general, the design rationale behind the processes in The Meta-Environment is *reusability*. The main processes are to be designed in such a way that they are programming language independent.

The goal is to be able to instantiate an IDE by providing the right parameters to the generic ToolBus scripts to obtain language specific IDE behavior.

The other main rationale behind these designs is *extensibility*. For example, the asynchronous, parallel and independently addable event listeners make sure that the system can be layered, and also extended without changing the existing functionality. However, this does introduce the need for transaction management.

For *traceability* of ToolBus scripts we try to use ToolBus process calls over snd-messages whenever possible, except to communicate with tools. This allows the ToolBus type checker to find common mistakes early in the process. Also the trace of process calls is more easy to get an overview of than the trace of messages and notes, since every process call has a single caller and a single callee. Messages on the other hand are send by one message but could be received by anybody.

Analysis of Results

The configuration management process of The Meta-Environment is not general enough to be easily reusable or extensible. There is still a lot of editing in existing source code involved when reusing or extending it.

The focus on reusability and extensibility has introduced strains on the ToolBus in terms of efficiency. Specific optimizations that help managing large volumes of notes, messages and process calls have helped a lot in making the systems run more smoothly:

- Statically computing and dynamically maintaining a map that finds possible communication partners for both messages and notes, as opposed to looping over all available processes.
- Changing the use of highly generic snd-msg's for binding configuration parameters to the use of dynamic process calls. The benefit is that there is no need to find a communication partner since the process is already named.

Assumptions

The programmer needs to understand the contracts between tools and the ToolBus well in order to handle the management processes of the Meta-Environment. These contracts are expressed in the idef files for each tool.

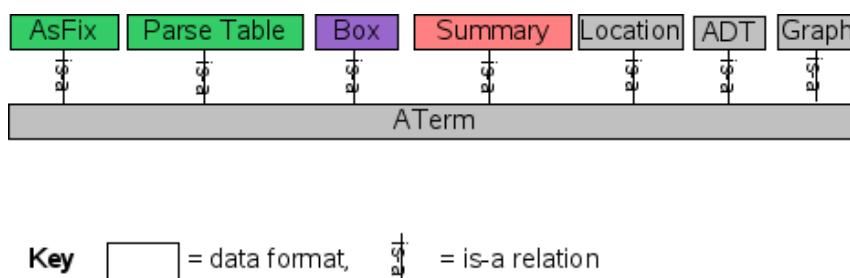
Other Information

- The ToolBus uses TCP/IP sockets to implement the connections with tools.

The Data Layer View (DAT)

The Meta-Environment uses ATerms as pervasive data format to store and exchange data. On top of the ATerms, various specialized data formats have been defined as shown in figure The data layer view.

Figure 1.6. The data layer view



Element Catalog

ADT, AsFix, ATerm, Box, Graph, Location, Parse table, Summary.

Context Diagram

None.

Variability Guide

New data formats can be added by writing new data definitions (using ADT) and generating the interfaces for these data using APIGEN.

Architecture Background

Design Rationale

The fundamental assumption is that tools exchange data in the form of ATerms. Since specialized services require specialized data formats, it makes sense to build them on top of ATerms.

Analysis of Results

The above data formats have been added out of necessity. It is likely that more data format will be added for:

- Rstores: (name, relation) pairs as used by Rscript.
- Configuration information.

An issue that requires attention is whether the ToolBus should be aware of these datatypes. Currently, it just pumps ATerms around, but more specialized typing of the tool interfaces would help implementors with early error detection.

Assumptions

The assumptions in this view are:

- These data format are versatile enough.
- The mapping of these data formats is sufficiently efficient.

Other information

None.

The Package View (PCK)

For reasons of modularity, reuse and maintenance, the implementation of The Meta-Environment is subdivided in *packages*. A package is the smallest unit of software that can be distributed or re-used. Currently, The Meta-Environment provides over 60 packages that provide a wide range of functionality. The table Mapping between elements and packages shows how the architecture elements in the various views are implemented by one or more packages. Note that some packages (e.g., aterm, aterm-java, shared-objects, JJTraveler) are used by most package; they are not listed in the table.

Table 1.3. Mapping between elements and packages

Element	Implementing packages
APIGEN	apigen
ASF checker	asf
ASF compiler	asf, asc-support
ASF interpreter	asf
ASF operations	asf-support
ASF+SDF checker	asf
ASF+SDF library	asf-library, sdf-library
ASF+SDF modules	
Compiled specification	asf
Configuration manager	config-manager, config-support
Debugger	tide, tide-support
Errors & warnings	error-gui, error-support
Graphical user interface (GUI)	dialog-gui, graph-gui, graph-support, meta-studio, module-details-gui, navigator-gui, progress-gui
Input term	pt-support
Module manager	module-manager, module-support
Output term	pt-support, pandora
Parse table generator	pgen
Parse table	pgen
Parser	sgr
Parse tree	pt-support
Prettyprinter	pandora
SDF checker	pgen
SDF operations	sdf-support
Structure editor	structure-editor
Text editor	editor-manager, editor-plugin
Term store	term-store, io-support
The Meta-Environment	asfsdf-meta, sdf-meta, meta
ToolBus	toolbus, toolbuslib, toolbus-java-adapter

Element Catalog

All architecture elements and nearly all packages appear in this view.

Context Diagram

None.

Variability Guide

The following variability issues have been identified.

- All packages have been written with portability in mind.

- We use configuration tools (autoconf) to adjust packages to local installation requirements.
- All software development has been done on Linux. However, the majority (but certainly not all) of the packages have been compiled successfully on Windows as well.

Architecture Background

Design Rationale

The reasons for splitting up the system in so many packages is the following:

- Each package represents a clearly defined functionality. This localizes the impact of future modifications.
- Each package can be reused independently.
- The package structure makes explicit dependencies between packages mandatory. This makes it easier to study the impact of more global changes.

Analysis of Results

On the positive side, we do have achieved an incredibly modular and flexible architecture. On the negative side, we do have to admit the following:

- Although "all packages are considered equal" in our approach, in reality this is not the true. A handful of primary packages (asfsdf-meta, aterm, pandora, pgen, sglr, shared-objects, toolbus) are frequently downloaded and reused in a variety of applications, the others are just implementation details for those primary packages.
- Initially, we exposed the flat list of all our packages to the outside world. This turned out to be very confusing.
- In some cases, a more monolithic structure for subcomponents would make the life of implementors somewhat easier.
- There are some costs involved in creating and maintaining a new package. We are therefore continuously simplifying and optimizing our build environment to reduce these costs.

Despite these critical notes, we do think that the overall package approach is highly effective.

Assumptions

We have made the following assumptions that are mostly hardwired in this architecture:

- Most packages can read/write data in the form of ATerms or any of the dataformats built on top of ATerms.
- Most packages provide their functionality in three forms;
 - As command line tools.
 - As a library that can be called from other programs.
 - As a ToolBus tool that can be connected to a ToolBus-based system.

The Package Dependency View (DEP)

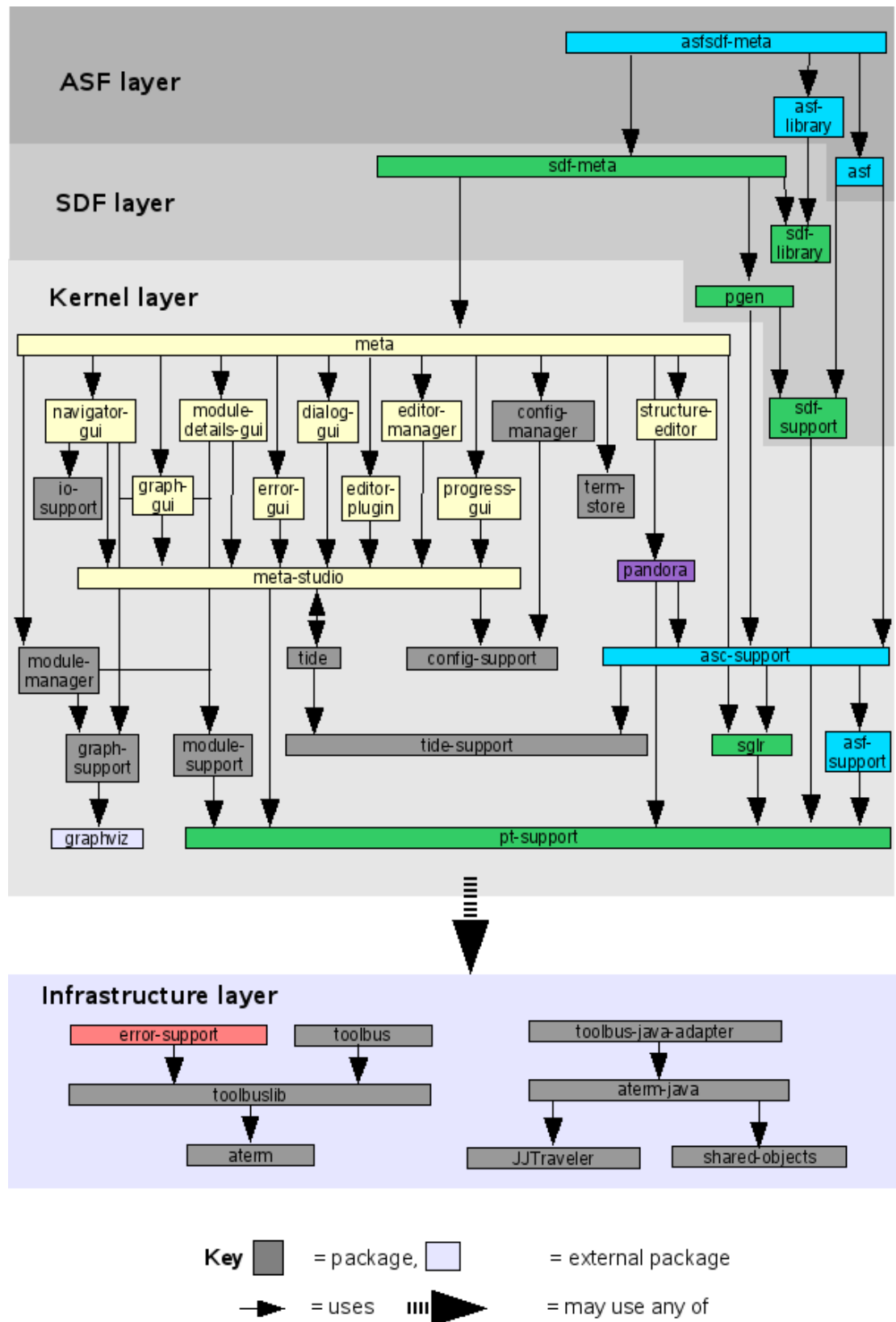
The dependencies between the packages of the Meta-Environment are shown in figure The package dependency view. Clearly, four separate layers can be distinguished (from bottom to top):

- The *infrastructure layer* that provides common facilities.
- The *kernel layer* that provides a kernel meta-environment.
- The *SDF layer* that provides a meta-environment for editing SDF specifications.
- The *ASF layer* that provides the full ASF+SDF Meta-Environment.

In order to reduce the visual clutter in this figure, various simplifications have been applied:

- All uses by elements in the top layers of facilities in the bottom layer are summarized by one arrow: "may use any of".
- An element uses all other elements to which it is directly or indirectly connected.

Figure 1.7. The package dependency view



Element Catalog

All packages.

Context Diagram

None.

Variability Guide

This package structure has been designed to accommodate variability; it enables arbitrary combinations of packages. This variability can be achieved by building new packages that depend on combinations of existing packages.

Architecture Background

Design Rationale

The design rationale for this package structure is the following:

- We come from a situation in which many packages had dependencies on ASF+SDF. We considered this undesirable, from the perspective of modifiability and extensibility.
- By organizing the package dependencies in layers, we achieve an architecture that can be easily extended. Starting with a widely used infrastructure layer, we first provide a kernel meta-environment that has a user-interface and basic editing facilities. Next, we add an SDF layer that provides syntax definitions via SDF, parser generation and parsing. Finally, we add ASF to provide semantic definitions and term rewriting.
- Each of these layers can form the starting point for the implementation of systems with widely varying functionality.

Analysis of Results

The fact that various "clones" of the Meta-Environment exist proves the success of this approach.

Assumptions

- The primary program representation is the parse tree (using AsFix). This assumption is becoming more and more true in the context of software analysis, program transformation and software renovation. However, in the context of ordinary compilation this maybe overkill and lead to inefficiencies.
- All packages need to use the same build interface in order to cooperate in this package structure.

Further information

For the record, there are some issues that should be resolved in this view:

- Only the dependency on graphviz is made explicit in the package definitions.
- The mutual dependency between tide and meta-studio is curious and should be investigated.
- It is a pity that there are still dependencies between the "meta" layer and asc-support. This means that not yet all ASF dependencies have been cut out of the kernel meta-environment.

Running The Meta-Environment is currently dependent on the following external packages:

- Graphviz [<http://www.graphviz.org>] for graph layout algorithms.

- Prefuse [<http://prefuse.org>] for graph visualization.
- Infonode [<http://www.infonode.net>] for docking windows.
- GCC [<http://www.gnu.org>] the GNU C compiler.

The Power-User View (PWR)

Power-users can use the system in four ways:

- By using the GUI.
- By using some of the commands.
- By writing programs and using the provided libraries.
- By modifying or extending the standard ToolBus scripts that are provided.

In advanced applications all three ways will play a role. This can be used to implement completely tailored Meta-Environments.

The relevant information is shown in the two tables Main commands and All commands and libraries per package.

The power-user is also confronted with the various data formats that are used: ATerms, AsFix, Parse tables

Table 1.4. Main commands

Command	Inputs	Output
pt-dump	Module name (String)	Parse table (ATerm)
sdf2table	SDF definition	Parse table (ATerm)
eqs-dump	Module name (String)	Equations (AsFix)
asfe	Equations (AsFix)	Parse tree (AsFix)
asf2c		
sgr	Parse table (ATerm), String	Parse tree (AsFix)
unparsePT		
apply-function	Function name (String), Sort name (String), Module name (String), Term (AsFix)	Term (AsFix)

		termstoarray sdf2table sdfchecker		
progress-gui	GUI component The Architecture of The Meta-Environment that display progress of an operation		progress.jar	T
Table 1.5. All commands and libraries per package				
pt-support	Support libraries for parse trees	addPosInfo ambtracker apply- function comparePT flattenPT implodePT liftPT unparsePT unparseProd	libPTMEPT.a libmept.a	C, T
sdf-library	Source library of SDF definitions			ASF
sdf-meta	An SDF-only Meta-Environment	def-dump pt-dump sdf-meta		T
sdf-support	Support libraries for handling SDF definitions	sdf-modules sdf- renaming	libPT2SDF.a libSDF2PT.a libSDFME.a	T
sglr	SGLR parser	dump-actions dump-gotos dump- priorities dump- productions restorebrackets sglr	libsglr.a	C, T
shared-objects	Library for maxiamlly shared objects in Java		shared- objects-1.4.jar	
structure-editor	Tools for syntax- directed editing	structure-editor	libStructureEditor.a	T
term-store	Generic store for terms	term-store		T
tide	Generic debug framework	tide tide-gdb	tide.jar	T
tide-support	Tools for connecting C- based tools to tide		libtide-adapter.a	C, T
toolbus	ToolBus coordination architecture	bc-adapter emacs- adapter idef2tif merge-tifs start- emacs tblog toolbus uri-encode ctif gen-adapter informer perl- adapter tbgraph tifstoc toolbus- adapter wish- adapter		C
toolbus-java- adapter	Adapter for coupling of Java to ToolBus	java-adapter tifstojava	toolbus-java- adapter-1.0.jar	
toolbuslib	Library for C- based ToolBus tools	tbunpack	libATB.a	C
JJTraveler	Traversal pattern in Java.		jjtraveler-0.5.jar	

Element catalog

This view shows all packages and some of their most important contents.

Context diagram

None.

Variability guide

See Package Dependency View.

Architecture background

See Package Dependency View.

Other information

Also see Extension Points for The Meta-Environment and Build Environment for The Meta-Environment. (*Add Links*).

The PIMP Meta-Environment view (PMV)

There are two versions of The Meta-Environment. One has a home grown GUI and GUI extension framework (MetaStudio), the other is based on and fully integrated with Eclipse using IMP (<http://www.eclipse.org/imp>). This view describes the differences between the two instances. The goal of this view is to help the designers and implementers of the Eclipse based Meta-Environment make design decisions.

Same high-level architecture

The main thing to notice is that both environments have the same high-level architecture: the *ToolBus* is the central mechanism for coordination. However, as Eclipse already integrates and composes several tools and does not use the ToolBus there may be overlapping functionality, or functionality from Meta-Environment needs to be replaced by or ported to Eclipse.

The ToolBus design allows us to maintain a heterogeneous implementation of The Meta-Environment. This allows phased evolution and separation of concerns, at the price of integration which is mitigated by the ToolBus.

Know about IMP

- IMP builds on Eclipse, and in particular on Eclipse's plugin mechanism that uses the concept of *extension points*. Read about extension points at eclipse.org. It is important to understand OSGI and Eclipse's plugin mechanism in some detail.
- IMP revolves around the concept of a *language descriptor*, which identifies the name of a language using an identifier and the associated file extensions. A language descriptor is declared using an extension point.
- IMP *services* are predefined interfaces that can be implemented for every language. The binding between an implementation of such an interface and a language is done using an extension point.
- IMP uses the implementation of services to orchestrate IDE features of Eclipse. The central orchestration is done in the "UniversalEditor" class. Other orchestration is done via the Eclipse Nature and Builder facilities.

IMP services implemented as language parametric ToolBus tools

In our case, the languages we are talking about are SDF and ASF+SDF. So, we could instantiate IMP by implementing the IMP services specifically for these languages. This is not the way Meta-Environment is designed. As said in other views, there is a kernel layer which contains both language independent and language parametric functionality. The goal of a PIMP'ed Meta-Environment is to bridge Meta-Environment to IMP at this kernel layer in such a way that we can easily use Meta-Environment to get other IMP-based IDE's. We expect for example to produce IDE's for RScript and newer version of ASF+SDF. Therefore, we need language parametric implementations of all IMP services.

- A language that builds on the Meta-Environment kernel, such as SDF, will declare an extension point for each service that points to the language parametric service implementation.
- This service implementation will (in 99% of the cases) be a ToolBus tool (a facade basically), that forwards the work to a ToolBus script.
- The ToolBus script implements the features that the service needs and communicates information back to the language parametric service implementation class, and/or activates other tools.
- Note that for each IMP service there will be
 - a Java class that both is a ToolBus tool and implements the IMP service interface
 - a ToolBus idf process that communicates with this tool integrates with the rest of the Meta-Environment coordination scripts

Examples of IMP services that are ToolBus tools are:

- IParseController - is called when a file needs to be parsed according to the UniversalEditor, the IParseController implementation relays this message to the ToolBus, which calls sglr, etc. etc.
- ITokenColorer - is called when the UniversalEditor decides it is time to add highlighting to the editor. The ITokenColorer implementation receives a handle to the UPTR parse tree (an ATerm), sends this to the ToolBus to be processed by the structure editor and receives a list of source locations tupled with font names back. Then it returns the TokenIterator which is required by the UniversalEditor which iterates over the source locations.

Eclipse features reified as ToolBus tools

IMP does not hide all of Eclipse. Especially concepts such as projects, resources, and error markers & annotations originate from Eclipse.

- To communicate with these Eclipse features, small tools that bridge to the ToolBus should be created, and data that are interfaces and classes in Eclipse should be reified as ATerms. This is completely parallel to the implementation of IMP services. The difference is that the number of IMP services is a closed and small set, while Eclipse features are open-ended, especially considering the plethora of Eclipse plugins.
- In many cases, the Meta-Environment has similar concepts that only need to be adapted to fit with Eclipse IMP. One example is errors. The error ATerms can be registered as Eclipse error annotations by a simple adapter written in Java that is connected to the ToolBus.
- Some concepts have "impedance mismatch". The concept of an Eclipse resource closely resembles the Meta-Environment concept of a module. However, the two are actually rather different. The module concept of the Meta-Environment needs to be adapted to fit the resource concept of Eclipse. The end result of the PIMP'ed Meta-Environment should be something that satisfies the expectations of Eclipse users.

- Note that each Eclipse feature that is reified as a ToolBus tool will have:
 - A Java class that is a ToolBus tool which takes ATerms from the ToolBus and gives them a meaning by calling Eclipse functionality.
 - A ToolBus idef process that communicates with this tool and integrates with the rest of the Meta-Environment coordination scripts.

Examples of tools that reify Eclipse features are:

- `org.meta_environment.eclipse.errors.ErrorViewer`: it implements the same idef as the Meta-Environment error viewer, receiving lists of error messages. The implementation of the tool simply registers the errors with the Eclipse annotation manager
- `org.meta_environment.eclipse.files`: communicates all identified resources and registers them with the module manager.
-

To Do

- *Remove all remarks from the text.*