

ASF+SDF Meta-Environment: Guided Tour

Revision : 1.16

M.G.J. van den Brand and P. Klint

Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

20th September 2004

Abstract

This is the guided tour for the ASF+SDF Meta-Environment Release 1.5.

The purpose of this document is to give a brief introduction in using the ASF+SDF Meta-Environment. Details on writing ASF+SDF specifications are not discussed in this guided tour, but in the ASF+SDF user manual.

This guided tour is under permanent construction.

Some images © 2001-2002 www.arttoday.com.

Contents

1 Overview	2
1.1 When to use the ASF+SDF Meta-Environment?	2
1.2 Global Structure of the Meta-Environment	3
1.3 About this Manual	3
1.4 Downloading the ASF+SDF Meta-Environment	4
1.5 Further Reading	4
2 Starting the System	5
3 The Main Window	6
3.1 The File menu	8
3.2 The Cache menu	11
3.3 The Tools menu	11
3.4 The Panes of the Main Window	12
3.4.1 The Import Pane	12
3.4.2 The Module Pane	12
3.4.3 The Module Menu	13
4 Editing Specifications	15
4.1 Editing the Syntax Part of a Module	17
4.2 Editing the Equations Section of a Module	17
4.3 Editing Terms	17
5 Message Tabs	18
5.1 Log Message Tab	19
5.2 Info Message Tab	19
5.3 Error Message Tab	19
5.4 Parse Errors	20

5.5	Type check warnings for plain SDF	20
5.6	Type check errors for plain SDF	21
5.7	Type check warnings for ASF+SDF	21
5.8	Type check errors for ASF+SDF	22
5.9	Type check warnings for ASF	22
5.10	Type check errors for ASF	22
6	Libraries	23
7	Guided Tour	23
7.1	Before you start the Guided Tour	23
7.2	Beginning the Guided Tour	23
7.3	The Module Booleans	24
7.3.1	The Module Editor for Booleans	24
7.3.2	A Term Editor for Booleans	27
7.3.3	Modifying Booleans	29
7.4	The Pico Specification	30
7.4.1	The Module Editor for Pico-Syntax	31
7.4.2	A Term Editor for Pico-syntax	31
7.4.3	More Exercises to Study the Pico Specification	33
7.4.4	Module Pico-typecheck	33
7.4.5	Module Pico-eval	33
7.4.6	Module Pico-compile	34
	section*Update with respect to previous version	

- Separated the guided tour and the ASF+SDF user manual.
- Synchronized the screen dumps with the newest version of the ASF+SDF Meta-Environment (version 1.5).

1 Overview

1.1 When to use the ASF+SDF Meta-Environment?

The ASF+SDF Meta-Environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. The generation process is controlled by a definition of the target language, which typically includes such features as syntax, pretty printing, type checking and execution of programs in the target language. The ASF+SDF Meta-Environment can help you if:

- You have to write a formal specification for some problem and you need interactive support to do this.
- You have developed your own (application) language and want to create an interactive environment for it.
- You have programs in some existing programming language and you want to analyze or transform them.

The ASF+SDF formalism allows the definition of syntactic as well as semantic aspects of a (programming) language. It can be used for the definition of languages (for programming, for writing specifications, for querying databases, for text processing, or for dedicated applications). In addition it can be used for the formal specification of a wide variety of problems. ASF+SDF provides you with:

- A general-purpose algebraic specification formalism based on equational logic.

- Modular structuring of specifications.
- Integrated definition of lexical, context-free, and abstract syntax.
- User-defined syntax, allowing you to write specifications using your own notation.
- Complete integration of the definition of syntax and semantics.

The ASF+SDF Meta-Environment offers:

- Syntax-directed editing of ASF+SDF specifications.
- Testing of specifications by means of interpretation.
- Compilation of ASF+SDF specifications into dedicated interactive environments containing various tools such as a parser, a pretty printer, a syntax-directed editor, a debugger, and an interpreter or compiler.

The advantages of creating interactive environments in this way are twofold:

- *Increased uniformity.* Similar tools for different languages often suffer from a lack of uniformity. Generating tools from language definitions will result in a large increase in uniformity, with corresponding benefits for the user.
- *Reduced implementation effort.* Preparing a language definition requires significantly less effort than developing an environment from scratch.

1.2 Global Structure of the Meta-Environment

You can create new specifications or modify and test existing ones using the Meta-Environment. Specifications consist of a series of modules, and individual modules can be edited by invoking editors for the syntax part and the equations part of a module. All editing in the Meta-Environment is done by creating instances of a *generic syntax-directed editor*.

After each editing operation on a module, its *implementation* is updated immediately. It consists of a parser, a pretty printer, and a term rewriting system which are all derived from the module automatically.

A module can be tested by invoking a *term editor* to create and evaluate terms defined by the module. Term editors use the syntax of the module for parsing the textual representation of terms and for converting them to internal format (syntax trees). The equations of the module are then used to reduce the terms into normal form. This result is, in its turn, converted back to textual form by pretty printing it.

1.3 About this Manual

This manual is intended for those users that want to try out the ASF+SDF Meta-Environment. This manual is still under development and we welcome all feed back and comments.

The focus of this manual will be on using the system to write a specification like a type checker or evaluator for the toy language PICO. It follows the user-interface to explain the capabilities of the system. Topics that will be addressed include:

- How to start the system and exit it.
- How to create, open, and save a specification.
- How to edit the syntax and/or equations part of a module.
- How to edit a term.
- How to evaluate a term.
- How to compile a specification.

- How to parse a term outside the ASF+SDF Meta-Environment.
- How to rewrite a term using a compiled specification outside the ASF+SDF Meta-Environment.
- How to unparse parsed and/or normalized terms.

We do *not* explain in detail:

- The formalism ASF+SDF, see the ASF+SDF reference manual.
- The architecture and implementation of the system.
- The stand-alone usage of various parts of the system.

1.4 Downloading the ASF+SDF Meta-Environment

You can download the ASF+SDF Meta-Environment from the following location:

<http://www.cwi.nl/projects/MetaEnv/>

It provides links to the software as well as to related documents. Furthermore, via this link bugs can be submitted.

1.5 Further Reading

There are many publications about the ASF+SDF Meta-Environment itself, about the implementation techniques used, and about applications. We give here a brief overview of selected publications:

Overviews: [24], [30], [3], [4].

General ideas: [25], [26], [22].

ASF: [1].

SDF: [23], [35].

ASF+SDF: [1], [21].

Parser generation and parsing: [32], [27], [34], [33], [35], [17], [14].

Pretty printing: [19], [28].

Rewriting and Compilation: [10], [6], [18], [11], [7].

ToolBus: [2].

ATerms: [8].

Applications: [5],[16], [15].

Generic debugging: [31].

Traversal functions: [12], [13].

User manuals: [20], [29], [9].

Acknowledgements

Peter D. Mosses, Albert Hofkamp, Akim Demaille, Jurgen Vinju.

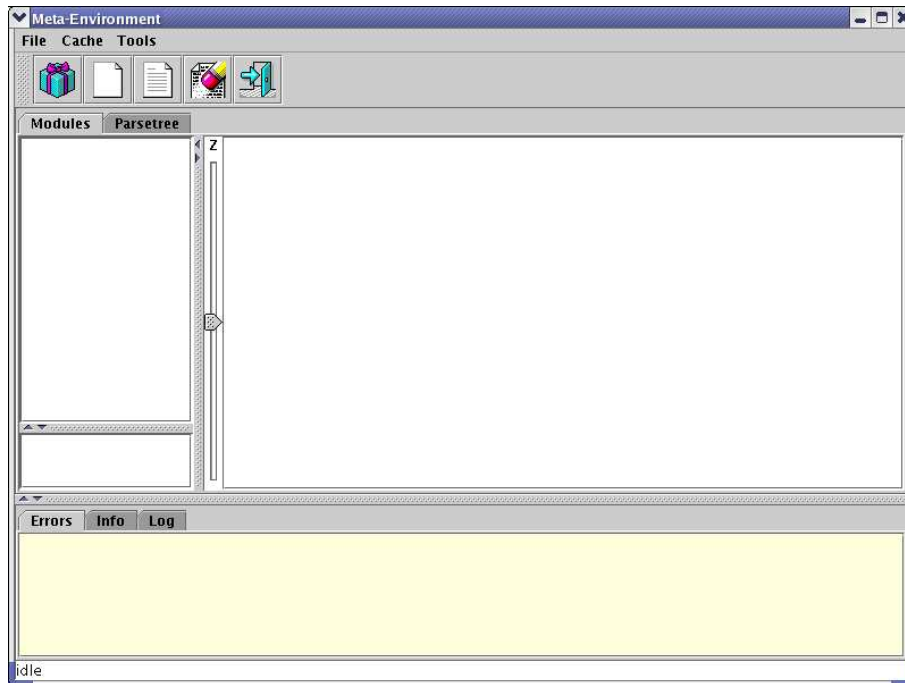


Figure 1: Main window of ASF+SDF Meta-Environment

2 Starting the System

The ASF+SDF Meta-Environment can be invoked via the command `meta`. As a result, the ASF+SDF Meta-Environment main window pops up. This is shown in Figure 1.

The `meta` command has the following options, which may come in handy later on. Note, the `meta` command delegates the actual invocation of the ASF+SDF Meta-Environment to the command `generic-meta`, this means that a number of option described below are not relevant for `meta`. As a novice user, you may want to skip the remainder of this section and continue with the description of the Main Window (Section 3).

- `-C file` specifies the configuration file to be used, in the default case the configuration file `meta.conf` is used.
- `-I dir` specifies the location of extra ToolBus scripts.
- `-d` starts the ASF+SDF Meta-Environment in debug mode. As a result, an interactive viewer will be started that allows the study of the internal behaviour of the system, the so-called “ToolBus viewer”. This viewer is shown in Figure 2.
- `-e` use the Emacs editor, this is the default editor.
- `-g` use the gvim editor, the support for this editor is in an experimental stage and will not be discussed in this guided tour.
- `-h` shows help information for the `meta` command.
- `-m modulename` starts the Meta-Environment and the module with the name `modulename` is automatically opened.
- `-o file` this option only works for the command line tools when dumping parse tables or equations.

- `-p path` adds a new search path to the list of search paths obtained from `meta.conf` in the Meta-Environment.
- `-r file` uses the given file as `term-store`. The term-store is normally saved under the name of `meta.termstore`.
- `-s` saves the term-store to disk, this option only works for the command line tools when dumping parse tables or equations.
- `-S file` executes the given `tb-script file`.
- `-t int` the Meta-Environment will abort after `int` seconds.
- `-T port` controls the communication ports that will be used for communication between the components of the ASF+SDF Meta-Environment. Note that these ports are also controlled by the environment variable `TB_PORT`. The default value is `8999`, but this port may be in use by someone else (or by an aborted previous run of the ASF+SDF Meta-Environment). In that case, it is advisable to use other values in the range `9000` and up.
- `-v` runs the ASF+SDF Meta-Environment in verbose mode.
- `-V` shows the version number of the ASF+SDF Meta-Environment you are running.

Search paths can be initialized by creating a file “`meta.conf`” in the directory from which the `meta` command is initiated. This file may contain a list of absolute and/or relative path names (each on a separate line) that will be searched when opening modules. For instance, in the `pico` directory (see the Guided Tour, Section 7) you will find an example `meta.conf` file which only contains the path `'.'`, i.e., only the current directory will be searched.

3 The Main Window

The main window of the ASF+SDF Meta-Environment immediately after starting the system was already shown (Figure 1). After loading a specification it may look as shown in Figure 4.

The main window consists of the following parts:

- At the top of the window is a menu bar that contains the following menu:

- File: for
 - * opening a library module,
 - * opening an existing module,
 - * creating of a new module,
 - * closing a specification, and
 - * exiting the ASF+SDF Meta-Environment.

The `File` menu is described in Section 3.1.

- Cache: for
 - * loading a saved term store and
 - * saving the current term store.

The `Cache` menu is described in Section 3.2.

- Tools: for
 - * clearing the info and log panel and
 - * refreshing of buttons.

The `Tools` menu is described in Section 3.3.

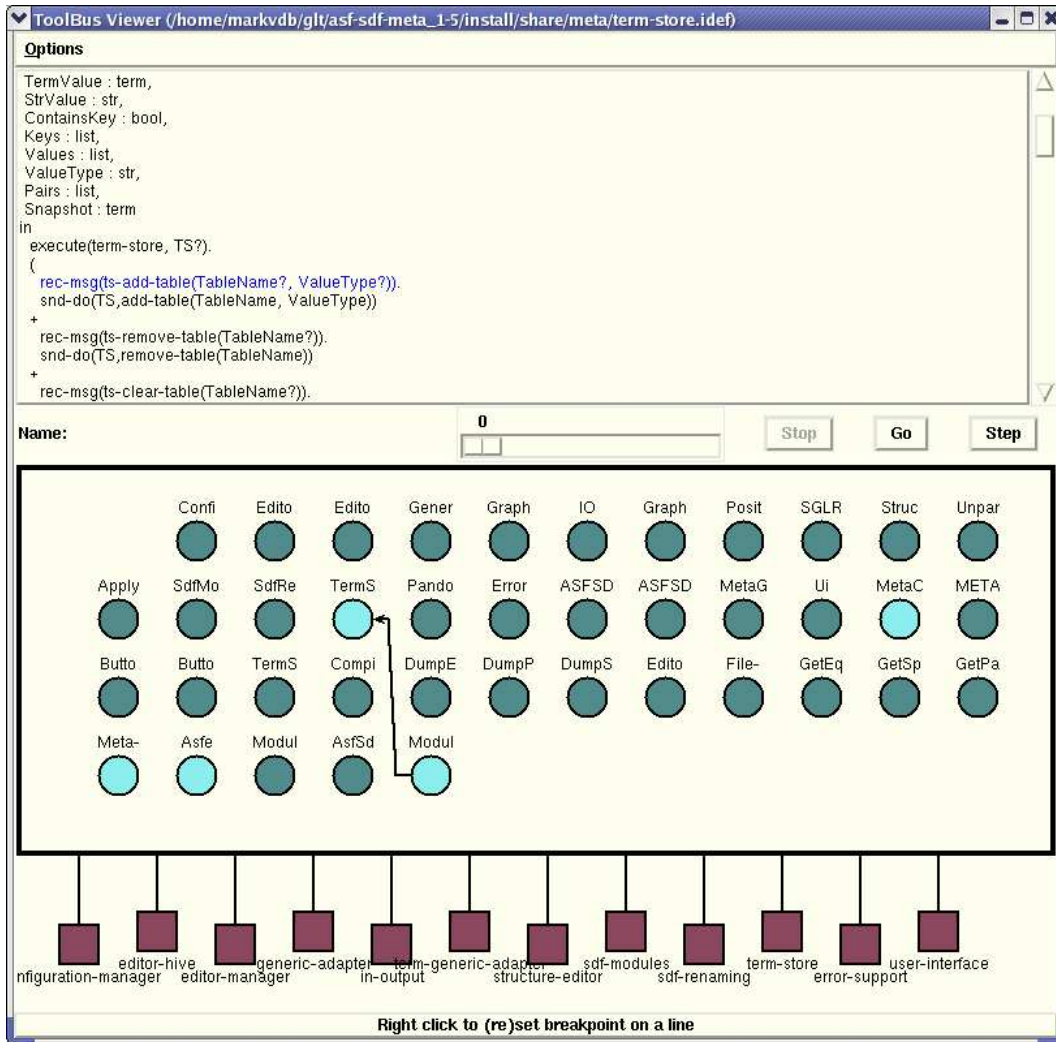


Figure 2: ToolBus viewer

- Next to File a list of icons is shown:
 - Open Library Module (the nicely wrapped box) for opening a library module.
 - New Module (the empty page) for creating a new module.
 - Open Module (the non-empty page) for opening an existing module.
 - Clear History (the eraser) for clearing the Info and Log panels.
 - Exit (the open door) for exiting the ASF+SDF Meta-Environment.

These icons are described in Section 3.1.

- The pane *import*: A graphical canvas (either empty in Figure 1, or containing rectangles and arrows in Figure 4) at the right hand side of the window shows the import graph of the specification you are editing. The import pane is described in Section 3.4.1.
- The pane *parsetree*: Again a graphical canvas which will be used to visualize the parse tree of a selected piece of a parsed term.

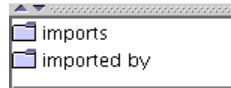


Figure 3: Part of the main window showing the imports and imported by "folders".

- The *module pane*: a vertical list (either empty in Figure 1, or containing names like `Pico-syntax`, `basic/Booleans`, ... in Figure 4) at the left part (in the middle) of the window that shows the names of all modules in the current specification. The module pane is described in Section 3.4.2.
- Below the *module pane* shows rectangle which will show the import relations of a selected module. This is shown via two folders:

`imports`: showing the list of modules the selected module imports.

`imported by`: showing the list of modules that import the selected module.

If no module is selected this part remains empty. By clicking on one of these shows the list of imports or imported modules. These folders are very convenient when processing a big specification.

- Below the module panel three panels are shown: *Errors*, *Info*, and *Log*. The *Errors* panel will be used for displaying warnings and error messages. The messages shown in this panel are clickable and will invoke the editors in which the warning or error is found. The *Info* panel shows general information derived during processing a specification. The *Log* panel shows all information which is also shown in the status bar, see below. The *Info* and *Log* panels can be cleared via the `Clear History` button in the main window.
- A *status bar* at the bottom of the window that shows the current activity of the system. Examples are: `idle` (the system is doing nothing), `parsing` (the system is performing a syntactic analysis of some module or term), and `rewriting` (the system is rewriting a term).

3.1 The File menu

The `File` menu is used for creating, opening, and saving specifications as well as for quitting the ASF+SDF Meta-Environment. It is shown in Figure 5.

`About` loads a module which contains version information and a number of links to web pages with more detailed information on the Meta-Environment. `Open Library Module` is used for opening a predefined ASF+SDF module. A dialog window (see Figure 6) appears. ASF+SDF library modules, see Section 6, are very convenient when developing a specification. It provides a number of predefined basic data structures and grammars.

`New Module` and `Open Module` are used for creating a new module and opening an existing one, respectively. In case of `New Module` a dialog window (Figure 7) appears. In case of `Open Module` a dialog window (Figure 8) appears.

`Close All` removes all modules from the Meta-Environment. If modules have been modified, you are explicitly asked to save them. The same effect can be achieved by exiting the Meta-Environment (using `Quit`, see below) and starting a new version of the Meta-Environment using the `meta` command.

`Refresh Buttons` reloads the file `meta.buttons`, if available, which contains a description of which buttons have to be added to which term-editors.

`Exit` ends the execution of the ASF+SDF Meta-Environment, before exiting the user is explicitly asked whether he/she wants to save the term store.

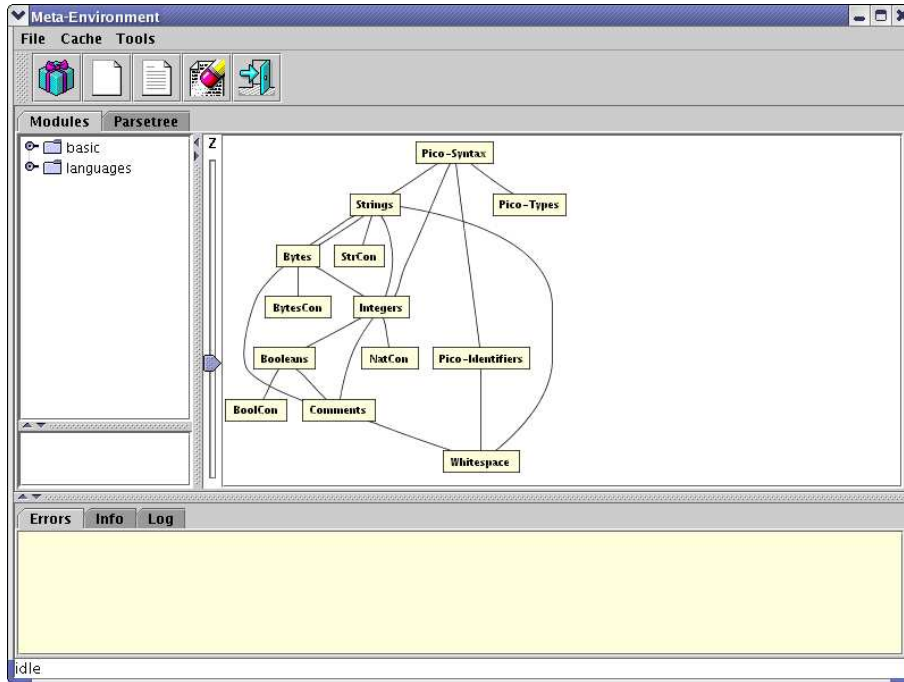


Figure 4: Main window after loading the Pico specification



Figure 5: File menu (main window)

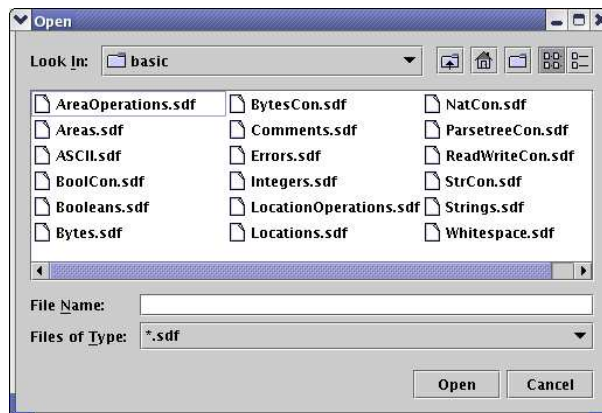


Figure 6: Dialog for opening a library module (File menu)

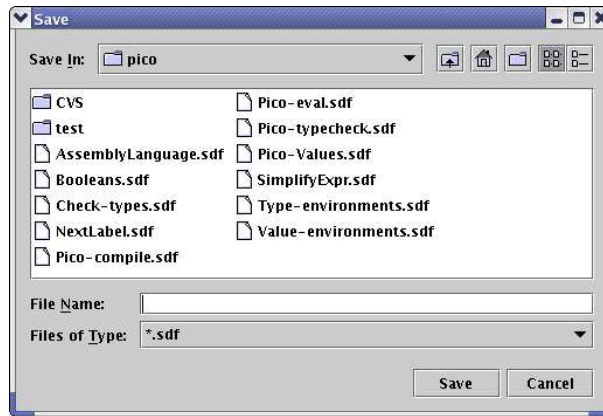


Figure 7: Dialog for creating a new module (File menu)

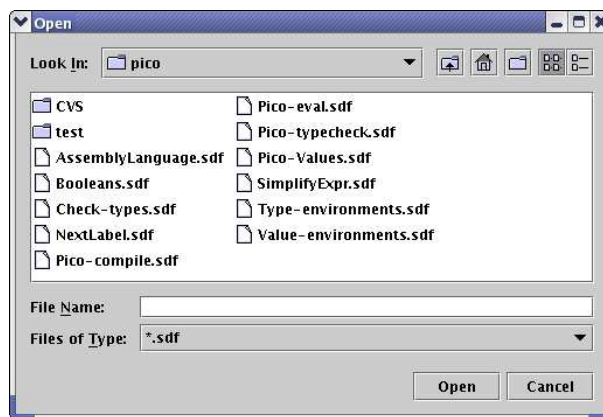


Figure 8: Dialog for opening a module (File menu)

Load Term Store...
Save Term Store...

Figure 9: Cache menu (main window)

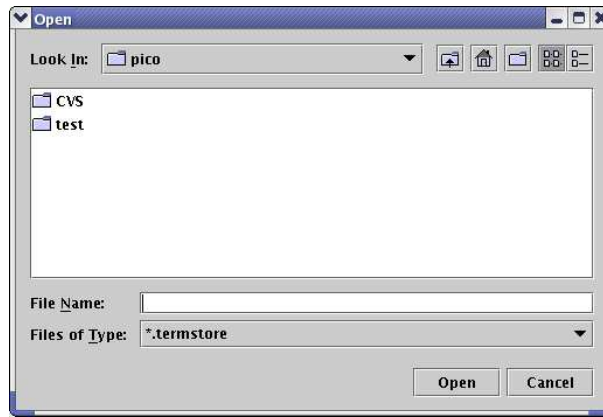


Figure 10: Dialog for loading a termstore (Cache menu)

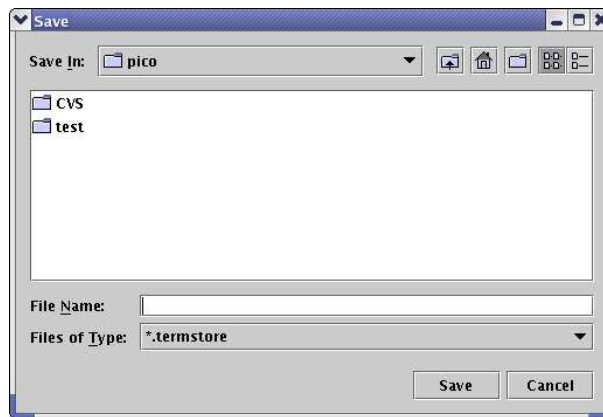


Figure 11: Dialog for saving a termstore (Cache menu)

3.2 The Cache menu

The `Cache` menu is used for loading and saving the term store in the ASF+SDF Meta-Environment. It is shown in Figure 9.

`Load Term Store...` reads a previously saved term store and replaces/initializes the term store. The termstore can be selected via the dialog window(Figure 10).

`Save Term Store...` saves the internal term store to disk. All information available in the internal term store is saved to disk, among others, generated parse tables, parsed equations, derived import relations. Using the term store when (re-)starting the system leads to a speed up because saved parse tables, etc. need not be regenerated. The name of the saved termstore can be selected or entered via the dialog window(Figure 11).

3.3 The Tools menu

The `Tools` menu is used for clearing the `Info` and `Log` panels of the ASF+SDF Meta-Environment and for reloading the `meta.buttons` file. It is shown in Figure 12.

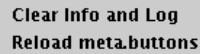


Figure 12: Tools menu (main window)



Figure 13: Pop up menu for module operations (import pane)

3.4 The Panes of the Main Window

The two panes below the icons of the main window give two, alternative, views on the ASF+SDF specification that has been loaded into the Meta-Environment. In the rightmost pane (Import Pane) you see the import graph, in the left-most pane (Module Pane) you see the module tree. Using one of these views, the same set of operations is available via a pop up menu.

3.4.1 The Import Pane

The pane with the name *import* gives a graphical view of the specification by displaying the *import* relation between modules in the form of a graph. A module M_1 imports another module M_2 if M_1 contains an import statement of the form `imports M_2` .

Each module is represented by a rectangle. An arrow between two rectangles represents an import relation between the two corresponding modules.

The import pane has the following interaction facilities:

- Different parts of the import graph can be displayed by using the horizontal or vertical scrollbar at the right and at the bottom of the import pane.
- The import graph can be scaled using the 100% icon.
- By clicking and holding the one mouse button outside any module, the import graph can be dragged across the import pane.
- Clicking *on* a module yields a pop up menu as shown in Figure 13. The functionality of the various entries is discussed in the module menu (see Section 3.4.3).

3.4.2 The Module Pane

The import pane is particularly useful when you want to understand the overall structure of a specification but it may become unwieldy for very large specifications. For large specifications the module pane may give you quicker access to the modules in the specification. It presents a vertical, scrollable, tree like view of all the modules in the specification.

The main purpose of the module pane is to select a module from the specification on which an operation from the button pane (Section 3.4.3) is to be performed. One module is selected by clicking on the corresponding module name in the module pane.

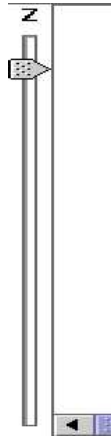


Figure 14: Ruler in the import pane

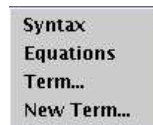


Figure 15: Pop up menu for invoking various editors for a module

After making the selection, an operation can be performed on all the selected module by pushing a button from the button pane. For instance, pushing the `Edit Syntax` button will create editors for the syntax of all the selected modules.

The import pane can be manually resized via the ruler on left of the import pane or automatically by clicking on the `Z` at the top of the ruler.

3.4.3 The Module Menu

First, a module can be selected via the import or module pane. Next, one of the following operations can be applied to it: `Edit`, `Check`, `Exports`, `Tools`, `Refactor`, `Close . . .`, and `Reopen`.

Edit By clicking on `Edit` a pop up menu as shown in Figure 15.

`Syntax`, `Equations`, and `Term . . .` activate (structure) editors for editing syntax, equations, or terms, respectively. `New Term . . .` enables the creation of a new file to be edited.

Edit By clicking on `Check` a pop up menu as shown in Figure 16.

Via the `Syntax` entry it is possible to invoke the checker which checks the well-formedness of SDF definitions. The `Equations` entry invokes the `asf`-checker to check the equations. The entry `Run Unit Tests` allows you to run the tests that are in the selected module.

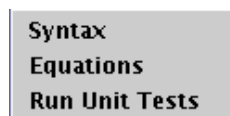


Figure 16: Pop up menu for invoking various checkers for a module

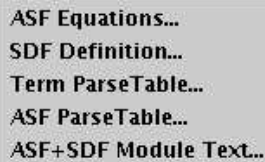


Figure 17: Export menu

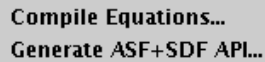


Figure 18: Tools menu

Export The `Export` menu opens a new menu (see Figure 17) in order to perform a number of export operations.

For all operations described below a file selector will be launched for selecting a the appropriate file or entering a new file name.

The `ASF Equations...` entry allows you to dump the equation of the selected module and all its imports. This feature is needed in order to run the evaluator in a stand-alone way, or to compile the specification on the command line.

The `SDF Definition...` entry allows you to dump the transitive closure of SDF modules into one file.

The `Term ParseTable...` entry allows you to dump the parse table of the selected module and all its imports in order to parse terms. This functionality is needed in order to use the parser in a stand-alone way.

The `ASF ParseTable` entry allows you to dump the parse table of the selected module and all its imports in order to parse the equations text of module. This functionality is only needed for debugging purposes.

The `ASF+SDF Module Text` entry allows you to print the text representation of both syntax and the equation part together in one file.

Tools The `Tools` menu opens a new menu (see Figure 18) in order to perform a number of operations.

For all operations described below a file selector will be launched for selecting a the appropriate file or entering a new file name. The `Compile Equations...` entry allows you to invoke the ASF+SDF-compiler to generate C code.

The `Generate ASF+SDF API...` entry allows you to derive from an SDF module a new module which contains functionality to compare and manipulate the items defined in the SDF definition.

Refactor `Refactor` opens another menu (see Figure 19) in order to refactor the specification.

The `Copy...` entry in the refactor menu allows you to make a copy of a module.



Figure 19: Refactor menu

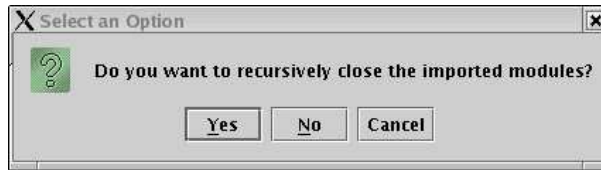


Figure 20: Select an Option window

The `Delete...` entry allows you to delete a module, not only in the import pane but also on disk! The import sections of the modules importing the deleted module are updated as well.

The `Rename...` allows you to rename a module and the renaming is also performed in the import sections of the modules importing the renamed module.

The `Add Import...` allows you to create a new import relation between 2 modules.

The `Remove Import...` allows you to cancel an import relation between 2 modules.

Close... This action removes a module from the specification. This action has only effect of the module is not imported by other modules. If the module imports other modules the user can decide to recursively close all imported modules as well. This can be done via `Select an Option` window (see Figure 20).

Reopen This entry allows you to revert this specific module from disk.

4 Editing Specifications

The editors used to create and modify specifications and terms are based on Emacs, so some familiarity with this editor is assumed. In this guided tour we restrict ourselves to the Emacs editors. But it is also possible to use GVim.

The various pull-down menus `Actions`, `Move`, and `Upgrade` have been added to the standard user-interface of Emacs, depending on the editor type:

- The `Action` menu, for syntax and equation editors, contains syntax specific buttons (Figure 21) and equation specific buttons (Figure 22), respectively.

The syntax editor specific buttons are:

- `Parse` for applying the SDF parser to the complete text buffer.
- `Check` for activating the sdf-checker in order to check the well-formedness of the SDF specification.
- `Edit Equations` to invoke the corresponding equation editor.
- `Edit Term` to invoke a term editor.
- `Run Tests` to run the corresponding unit tests defined for this module.

The equation editor specific buttons are:

- `Parse` for parsing the complete text buffer using the underlying SDF definition.
- `Check` for activating the asf-checker in order to check the well-formedness of the ASF specification, for instance, it is checked whether there are any uninstantiated variables in the right-hand side of an equation.
- `View Tree` for displaying the parse tree of the focus in the parse tree panel (Figure 25).
- `View Full Tree` for displaying the parse tree of the focus in the parse tree panel (Figure 26). In this case all individual characters are also shown. This feature is extremely useful when trying to find an ambiguity on the lexical level.

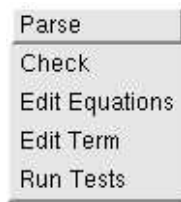


Figure 21: Actions menu (syntax editor)



Figure 22: Actions menu (equations editor)

- `Edit Syntax` to invoke the corresponding syntax editor.
- `Edit Term` to invoke a term editor.
- `Run Tests` to run the corresponding unit tests defined for this module.
- The `Move` pull-down menu contains four buttons (Figure 23) for structured traversal of the syntax tree of the text in the editor. Using the entries `Left`, `Right`, `Up`, `Down` the user can navigate in the tree.
- Both the syntax and equation editor have an extra pull-down menu `Upgrade`. The `Upgrade` menu for the syntax editor invokes a tool which transforms the SDF module. This transformation involves changing the tuple syntax, (see User Manual), quoting of unquoted literals, and changing symbol declarations into context-free start-symbols. The `Upgrade` menu for the equation editor involves the introduction of the new operators `==`, `!=`, `:=`, and `!:`, in the conditions of the equations, see the User Manual for more details.
- The term editors have the `Actions` menu which contains buttons (Figure 24).
 - `Parse` parses the term in the editor given the corresponding SDF definition.
 - `Reduce` that applies the evaluator to the text in the editor given the corresponding equations.
 - `View Tree` displays the parse tree structure of the focus in the `parsetree` pane.
 - `View Full Tree` displays the parse tree structure with more details of the focus in the `parsetree` pane.



Figure 23: Move menu (all editors)



Figure 24: Actions menu (term editor)

- ShowOrigin
- Dump ParseTable Saves the corresponding parse table to disk.
- Edit Syntax to invoke the corresponding syntax editor.
- Edit Equations to invoke the corresponding equation editor.

4.1 Editing the Syntax Part of a Module

An editor for editing the syntax part of a module can be activated by pressing the `Edit Syntax` button of the pop-menu in the Import Pane (Section 3.4.1), in the Module Pane (Section 3.4.2), or via an equation or term editor. An example is shown in Figure 32.

Initially the text is not highlighted, but the text has already been parsed. Click at an arbitrary place you will see that part of the text will be highlighted, this is what we will call the *focus* and the message “Focus symbol: <SORT>” appears in the status line at the bottom of the main window, where <SORT> will be the non-terminal/sort of the focus.

Via the entry `Parse` in the `Actions` menu of the editor, the parser can be activated. The parser is finished when the status line in the main window displays `Idle` again.

Note: when parsing a large term it may take some time for the editor to be active again. If the parse was successful, the bottom line in the Emacs window displays the message `Focus sort: None`. If the term contains an error, the cursor is located at the position where the error was detected and the bottom line in the Emacs window displays the message `Parse error near cursor`. When pushing the parse button the text will be saved first and then parsed.

4.2 Editing the Equations Section of a Module

An editor for editing the equations section of a module is activated via the button `Edit Equations` in the pop-menu in the Import Pane (Section 3.4.1), in the Module Pane (Section 3.4.2), or via a syntax or term editor. An example is shown in Figure 33.

The entry `Parse` in the `Actions` menu of the editor activates the parser for the equations. It is possible that in order to parse the equations, a parse table must first be generated. This is visible through the status message `Generating parsetable <ModuleName>`. When pushing the parse button the text will be saved first and then parsed.

4.3 Editing Terms

An editor for editing a term over a module is activated via the button `Edit Term` in the pop-menu in the Import Pane (Section 3.4.1), in the Module Pane (Section 3.4.2), or via a syntax or equation editor. An example is shown here in Figure 34.

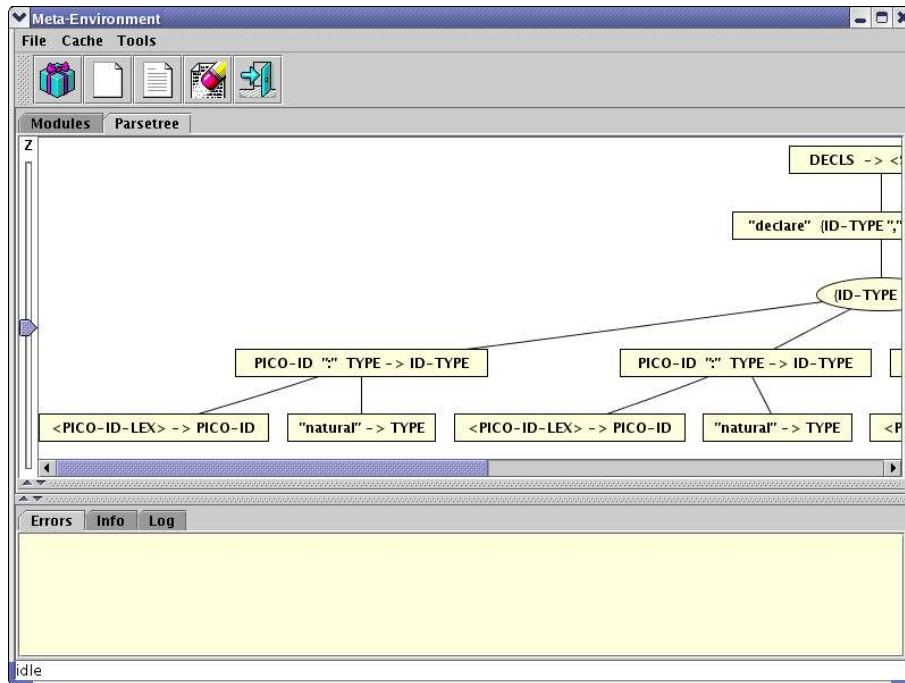


Figure 25: Main window of ASF+SDF Meta-Environment with the parsetree panel activated

The entry `Parse` in the `Actions` menu of the editor activate the parser for this term. It is possible that in order to parse the term, a parse table must be generated. This is visible through the status message `Generating parsetable <ModuleName>`. The parse action again involves saving of the text.

The entry `Reduce` in the `Actions` menu of the editor activates the evaluator¹. The term is reduced given the specified equations (if any). In order to reduce the term it may be necessary to parse the equations of various modules and to initialize the evaluator with this set of equations. Note that, clicking on the reduce button does not always imply saving and parsing the text.

When you have created a term (using a term editor, see Section 4.3), you can reduce it (by selecting the `Reduce` entry from the `Actions` menu of the term editor). As a result, rewrite rules will be applied until a normal form is reached (a term for which no applicable rule can be found). This normal form is the result of the execution and is displayed in a new term window.

The entry `View Tree` and `View Full Tree` in the `Actions` menu of the term editor displays the parse tree of the focus, if any. The parse tree will be shown in the `parsetree` panel, see Figure 25 or `parsetree` panel, see Figure 26.

5 Message Tabs

There are three different message tabs:

1. Errors
2. Info
3. Log

¹We will also use interpreter, rewriter, or reducer instead of evaluator, and equivalently we use interpreting, rewriting, reducing, or evaluating of a term, respectively

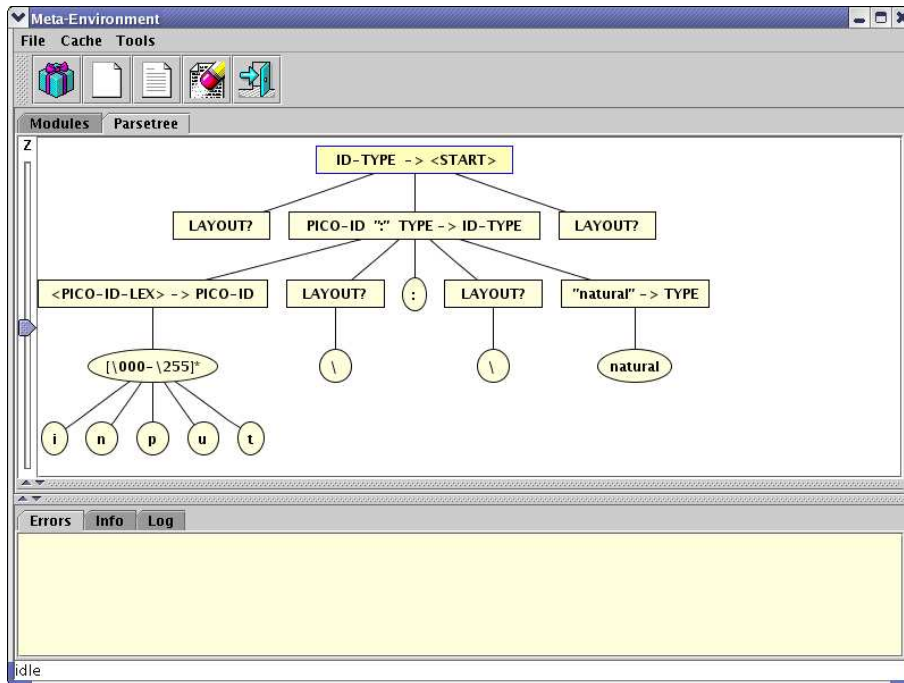


Figure 26: Main window of ASF+SDF Meta-Environment with the parsetree panel activated showing a more detailed version of the parsetree

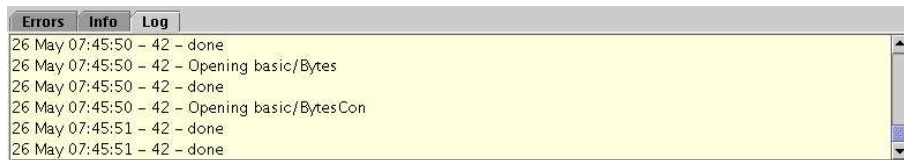


Figure 27: The Log message tab of ASF+SDF Meta-Environment after loading the module Pico-syntax

5.1 Log Message Tab

The Log message tab, see Figure 27 logs all status information decorated with a time stamp. Via the eraser button in the user interface or the `Clear Info` and `Log` button in the `Tools` pull-down menu this tab can be cleared.

5.2 Info Message Tab

The Info message tab, see Figure 28 shows more general information related to the specification being processed. Via the eraser button in the user interface or the `Clear Info` and `Log` button in the `Tools` pull-down menu this tab can be cleared.

5.3 Error Message Tab

The warnings and errors are displayed in the `Error` message tab, see Figure 29. The messages in this tab are clickable, clicking on the message will invoke the corresponding editor and the cursor will be position on the source of the error message. The messages shown here can only be removed by fixing the cause of the warning or error.

There are various categories of messages that will be displayed via `Error` tab:

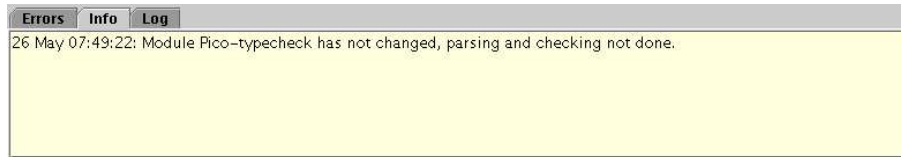


Figure 28: The Info message tab of ASF+SDF Meta-Environment after dumping the equations

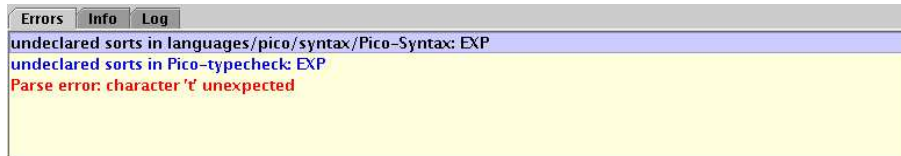


Figure 29: The Error message tab of ASF+SDF Meta-Environment

1. Parse errors.
2. SDF type check warnings.
3. SDF type check errors.
4. ASF type check errors.

We will enumerate the warning/error messages. The exact cause and how to fix such a warning or error is discussed in the reference manual.

5.4 Parse Errors

There are three different types of parse errors:

1. A syntax error, which is reported by pinpointing the exact location in the file and the message `Parse error near cursor` in case of an editor or in the message pane an error message similar to `Parse error: character '<c>' unexpected`. This means that the parser detected a syntax error in the text to be parsed and can not proceed its parsing process. Clicking on the error in the `Errors`-pane moves the cursor to the exact error location and launches if needed the editor. A variant of the syntax error message is: `Parse error: eof unexpected`.
2. A cycle, in case of an editor the cursor is positioned at the position where the first cycle is detected in the input and the message is `Cycle: <list_of_production_rules>` is printed. A cycle is reported whenever the parser detects a non terminating chain of reductions. All production rules on the cycle are shown as `<list_of_production_rules>`.
3. An ambiguity, again in case of an editor the cursor is positioned at the position where the first ambiguity is detected in the input and the message `Ambiguity: <list_of_production_rules>` is printed. An ambiguity is reported whenever the parser was able to recognized a (part of) the input sentence in different ways. The `<list_of_production_rules>` shows all production rules that are involved in the ambiguity.

5.5 Type check warnings for plain SDF

The warnings and error for SDF are separated into 4 sections. First we will discuss the type check warnings and errors (see Section 5.6) for plain SDF. This variant of SDF is independent of ASF. Later we will discuss

the warnings (see Section 5.7) and errors (see Section 5.8) for SDF used in combination with ASF. In this case we need to be more strict and every SDF construct is supported by ASF.

Warnings do not break the specification, but it is advisable to fix them anyway. Often they point out some not well-formed part in the specification.

- undeclared sorts
- double declared sort
- double declared start-symbol
- illegal attribute: {bracket, left, right, assoc, non-assoc}
- used in priorities but undefined
- inconsistent rhs in priorities
- unknown constructor used in priorities
- sort CHAR used in production rule
- deprecated tuple notation
- deprecated unquoted symbol notation
- deprecated non-plain sort definition
- aliased symbol already declared

5.6 Type check errors for plain SDF

- module not available
- start-symbols in <ModuleName> not defined in any right-hand
- literal in right-hand-side not allowed
- only sort allowed in right-hand-side of lexical-function
- double used label
- constructor has already been used

5.7 Type check warnings for ASF+SDF

- exported variables section
- kernel syntax construction
- production renamings not supported
- not supported symbol

5.8 Type check errors for ASF+SDF

- traversal attributes in non-prefix function
- illegal traversal attribute
- missing bottom-up or top-down attribute
- missing break or continue attribute
- missing trafo and/or accu attribute
- accu should return accumulated type
- trafo should return traversed type
- accutrafo should return tuple of correct types
- inconsistent arguments of traversal productions
- inconsistent traversal attributes
- asf equation sort must not be used
- charclasses not allowed in context-free syntax

5.9 Type check warnings for ASF

- Lexical probably intended to be a variable
- Deprecated condition syntax "="
- constructor not expected as outermost function symbol of left hand side

5.10 Type check errors for ASF

- equations contain ambiguities
- uninstantiated variable occurrence
- negative condition introduces variable(s)
- uninstantiated variables in both sides of condition
- uninstantiated variables in equality condition
- right-hand side of matching condition introduces variables
- matching condition does not introduce new variables
- strange condition encountered
- Left hand side is contained in a list
- no variables may be introduced in left hand side of test

6 Libraries

The ASF+SDF library modules are very convenient when developing a specification. It provides a number of predefined basic data structures and grammars. There are 4 different library categories:

- basic
- containers
- langauges
- utilities

The basic library provides basic data structures such as Booleans, Bytes, Comments, Integers, and Strings. Furthermore, it provides modules to access information stored in the underlying parse trees, such as position information.

The containers library gives a number of parameterized data structures, such as balanced trees, lists, sets, and tables.

The languages library allows the reuse of a number of grammars. This part of the library will be extended in the near future.

The utilies library provides functionality which can be very helpful when developing sophisticated ASF+SDF specifications.

7 Guided Tour

To help you get acquainted with the ASF+SDF Meta-Environment the system contains two example specifications. The first one is a very simple specification: `Booleans`, and the second is the specification of the syntax, typechecker, and dynamic semantics of the small programming language Pico.

This Guided Tour is meant to guide you through these specifications, and show you the main features of the ASF+SDF Meta-Environment. Only global information is given about these features but references are made to parts of the user-manual where detailed information can be found.

7.1 Before you start the Guided Tour

When configuring the Meta-Environment a directory was giving where the Meta-Environment will be installed, e.g., `<path>/asfsdf-meta`. You will then find the files needed for this Guided Tour in the directory `<path>/share/asfsdf-meta/demo/pico`. It is advisable to make your personal copy of this directory. In this Guided Tour we will use `pico` to refer to your own copy of the directory. Furthermore, all the examples given in this manual can be found in `<path>/asfsdf-meta/share/demo/user-manual-examples`.

For each module in a specification there exists a file `module.sdf` which contains the syntax of `module` and there may be a file `module.asf` which contains the equations (semantics) of `module`. The directory `pico` contains:

- Files for the Pico-specification.
- Three examples of Pico-programs: `big.pico`, `fac.pico`, `small.pico`.
- Terms for typechecking and evaluating these Pico-programs.

7.2 Beginning the Guided Tour

- Go to your personal copy of the directory `pico`.
- Type the command `meta`. The main window of the Meta-Environment will appear as shown in Figure 1.

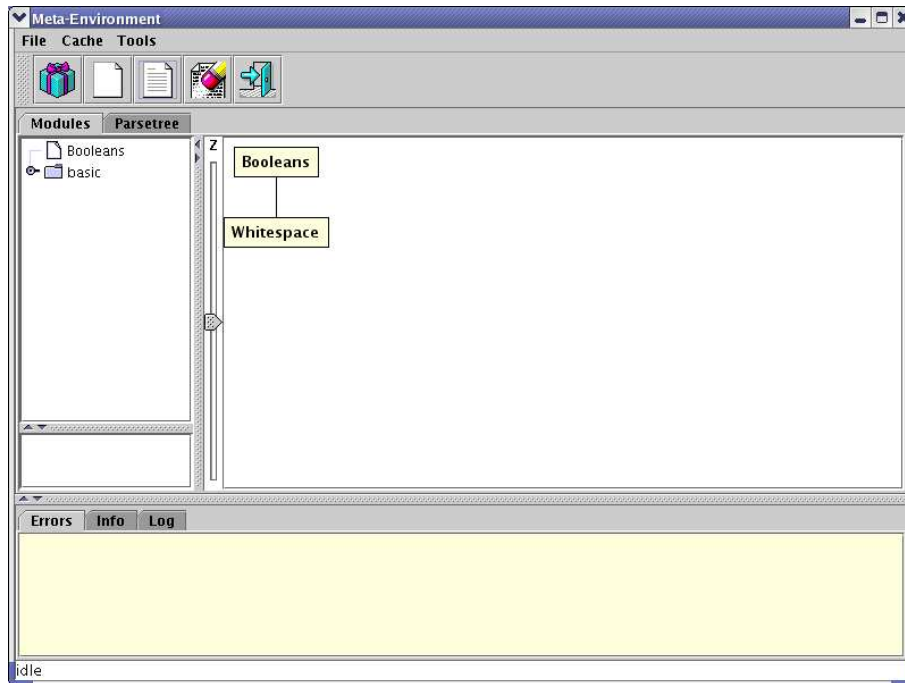


Figure 30: Main window after opening `Booleans`

- Add the module `Booleans` by selecting the `File` menu, and choosing the `Open Module` button. In a dialog window, the system asks you to give the name of the module to be opened. It presents a list of all files with extension `sdf`. Click once on `Booleans.sdf` and then push the `Open Module` button. This will load the module `Booleans` (both its syntax and equations!) into the system.
- Verify that module `Booleans` appears as a rectangle in the import pane as well as in the module pane of the main window as shown in Figure 30.

7.3 The Module `Booleans`

One of the simplest specifications possible, and therefore frequently used as an example, is the datatype of the Boolean values. It defines the constants `true` and `false` and the functions `and` and `or` (written in infix notation using the left-associative operators `&` and `|`, respectively) and `not` (written in prefix notation using the function symbol `not`). The specification is shown in Figure 31.

7.3.1 The Module Editor for `Booleans`

- Select module `Booleans` from the module pane (the vertical list of module names that now contains `Booleans` and `basic/Whitespace`) by clicking on it once.
- Push the button `Edit Syntax` in the button pane at the right-hand side of the main window. An editor will appear containing the syntax part of the `Booleans` specification: the SDF section. This editor is a version of the standard text editor Emacs extended with the menus `Actions` and `Move`. The result is shown in Figure 32.
- Push the button `Edit Equations`. This will open a new instance of Emacs containing the semantic part of the `Booleans` specification: a list of conditional equations. Note that the syntax of the equations is determined by the syntax defined in the SDF section. The result is shown in Figure 33.

```
module Booleans

imports basic/Whitespace
exports
  context-free start-symbols Bool

  sorts Bool

  context-free syntax
    "true"          -> Bool
    "false"         -> Bool
    Bool "|" Bool   -> Bool {left}
    Bool "&" Bool    -> Bool {left}
    "not" "(" Bool ")" -> Bool
    "(" Bool ")"    -> Bool {bracket}

  context-free priorities
    Bool "&" Bool -> Bool >
    Bool "|" Bool -> Bool

  hiddens
    variables
      "Bool"[0-9]* -> Bool

  equations

  [B1] true | Bool = true
  [B2] false | Bool = Bool

  [B3] true & Bool = Bool
  [B4] false & Bool = false

  [B5] not(false) = true
  [B6] not(true) = false
```

Figure 31: Specification of module Booleans

```

emacs@localhost.localdomain
File Edit Options Buffers Tools Actions Move Upgrade Help
module Booleans
imports basic/Whitespace
exports
  context-free start-symbols Bool

sorts Bool

context-free syntax
  "true"          -> Bool
  "false"         -> Bool
  Bool "|" Bool   -> Bool {left}
  Bool "&" Bool    -> Bool {left}
  "not" "(" Bool ")" -> Bool
  "(" Bool ")"    -> Bool {bracket}

context-free priorities
  Bool "&" Bool -> Bool >
  Bool "|" Bool -> Bool

hiddens
variables
  "Bool"[0-9]+ -> Bool
  Booleans.sdf (Fundamental CVS:1.3.4.2)--L9--Top
Focus symbol: Grammar

```

Figure 32: Editor for the syntax of Booleans

```

emacs@localhost.localdomain <2>
File Edit Options Buffers Tools Actions Move Upgrade Help
equations
  [B1] true | Bool = true
  [B2] false | Bool = Bool
  [B3] true & Bool = Bool
  [B4] false & Bool = false
  [B5] not(false) = true
  [B6] not(true) = false
  Booleans.asf (Fundamental CVS-1.1)--L3--All
Focus symbol: ASF-ConditionalEquation

```

Figure 33: Editor for the equations of Booleans

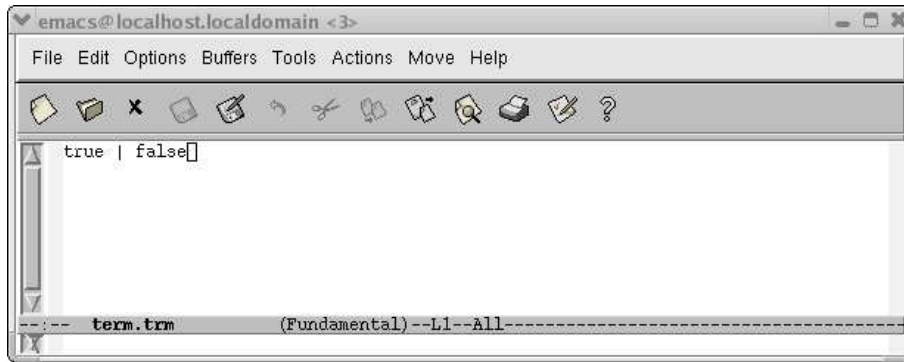


Figure 34: Term editor for Booleans after entering ‘true & false’

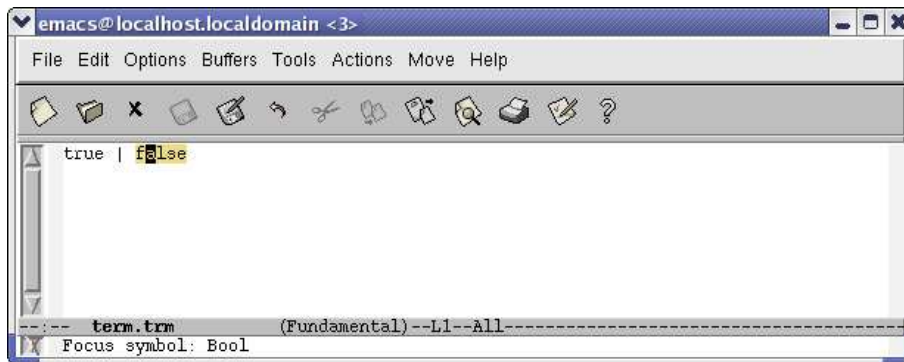


Figure 35: Term editor for Bool-example after clicking on ‘false’

7.3.2 A Term Editor for Booleans

- Open a term-editor over module `Booleans` by first selecting module `Booleans` in the module pane, and then pushing the `New Term ...` button. A standard dialog window pops up. Enter any new filename, for instance, ‘`term.trm`’.
- Type the term ‘`true & false`’ in this editor. The result is shown in Figure 34.
- From menu `Actions` click the `Parse` button. The text in the focus is now being parsed.
- Click on one of the characters of the word ‘`false`’, this will move the cursor (a single character-sized rectangle). You have selected ‘`false`’ as new focus and the blue background appears. This is shown in Figure 35.
- Click on the *and* operator ‘`&`’. The whole expression is now selected as focus.

The movements of the focus are *syntax-directed*: when you click on any character in the text, the smallest syntactic unit enclosing that character will be selected and becomes the focus.

- Reduce the term in the term-editor by clicking the `Reduce` button in the `Actions` menu of the editor. The result will appear in a new term editor window (Figure 36).

Error-messages

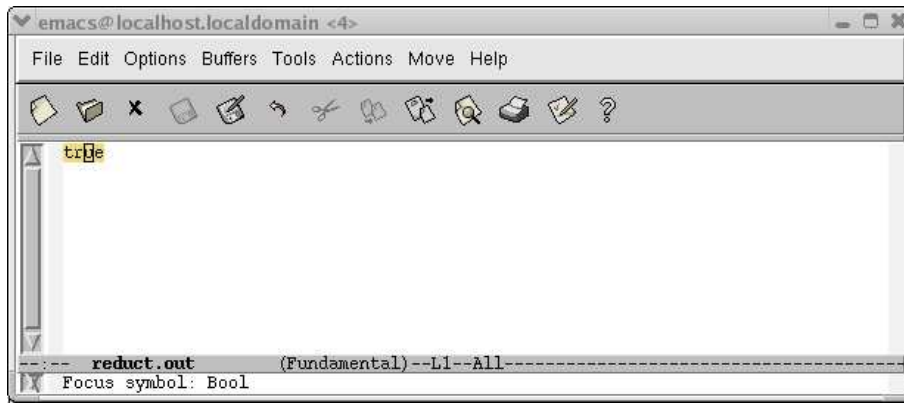


Figure 36: New term editor with normal form of 'true & false'

- Edit the term 'true & false' such that the new term will be syntactically incorrect. For instance, type 'true & wrong'. Force a parse of the term by selecting the Parse button of the Actions menu.

In the status line at the bottom of the edit window a message appears 'Parse error near cursor' and the cursor will be positioned in the word 'wrong'.

Associativity, Priorities and Brackets

- Erase the term in your term-editor and type a new term 'true & false & true'.
- Parse the term using the Parse button.
- Try to find out how this term has been parsed by clicking on different parts of the term and studying the resulting focus.

The left attribute in the SDF definition indicates that the '&' operator is left associative. The term will thus be parsed as '(true & false) & true'. Clicking on the left or right & yields a focus that corresponds with this parse.

- Erase the term in your term-window and type a new term 'true | false & true'.
- Parse the term using the Parse button.
- Try to find out how this term has been parsed by clicking on various parts of the term and studying the resulting focus.

The context-free priorities definitions in the SDF definition state that the '&' operator binds stronger than the '|' operator.

- Erase the term in the term-editor and type a new term 'true & false'. Click on 'false', so that the focus is around 'false' only. Then add '| true' after 'false', so that the resulting term is 'true & false | true'.
- Parse the term.
- Click on the '&' symbol. Is this what you wanted? Probably not.

To resolve a priority conflict '(' and ')', which are defined as brackets in the SDF definition are put around the term 'false | true'. Thus 'true & (false | true)' is more likely to express what you intended.

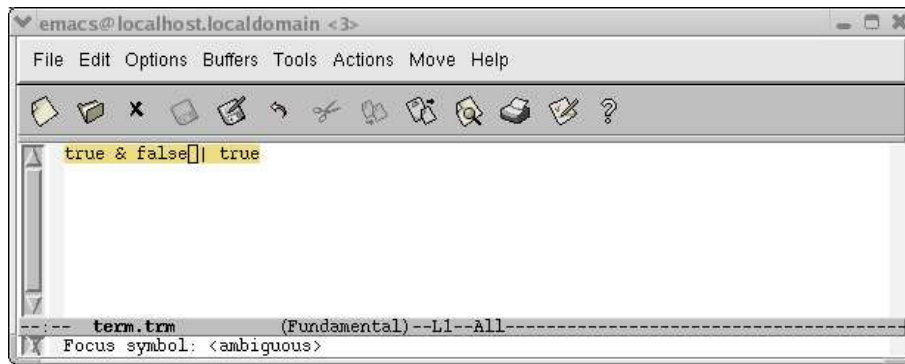


Figure 37: Term editor with an ambiguous boolean term

7.3.3 Modifying Booleans

The ASF+SDF Meta-Environment is an *incremental environment generator*. After each edit operation on a module, its *implementation* (i.e., scanner, parser and term rewriting system) is updated immediately.

The editing of both the syntax section and the equations section of a module is syntax-directed like the editing of terms in a term editor.

Modifying the Equations The equations section of a module begins with the keyword `equations` and is saved in files ending on `.asf`.

- Click in the equation section to investigate the focus behavior.
- Change the equations, for instance replace in equation [B1] the last part `= true` by `= false`.
- Study the effect on the reduction of terms in the term-editor.

Modifying the Syntax The syntax part of a module starts with the keyword `module` and is saved in files ending on `.sdf`. Modifying the syntax causes the generated scanner and parser to be adapted. After each edit operation in the SDF section that is followed by a parse of the SDF section, the focus in both the equations section and the term editor is extended to completely contain the text in these editors.

Modifying the context-free syntax:

- Change the syntax of the defined functions. E.g. replace `'not'` by `'negation'`.
- Try to re-parse the equations.

Modifying the priorities:

- Remove the priority declaration.
- Type the term `'true & false | true'` in the term-editor (or anything similar according to your current syntax). Parse this term. In the error pane of the user interface a message will be printed which indicates that the parse contained 1 ambiguity. Clicking in the term editor window (see Figure 37) will result in a focussed symbol `<ambiguous>`. Viewing the corresponding parse tree via `View Full Tree` in the menu `Actions` will give the parsetree panel (see Figure 38).
- Add the priority declarations again.

The effect of removing the `LAYOUT` definition.

- Remove import of `basic/Whitespace`.
- Try parsing equations of `Booleans`.

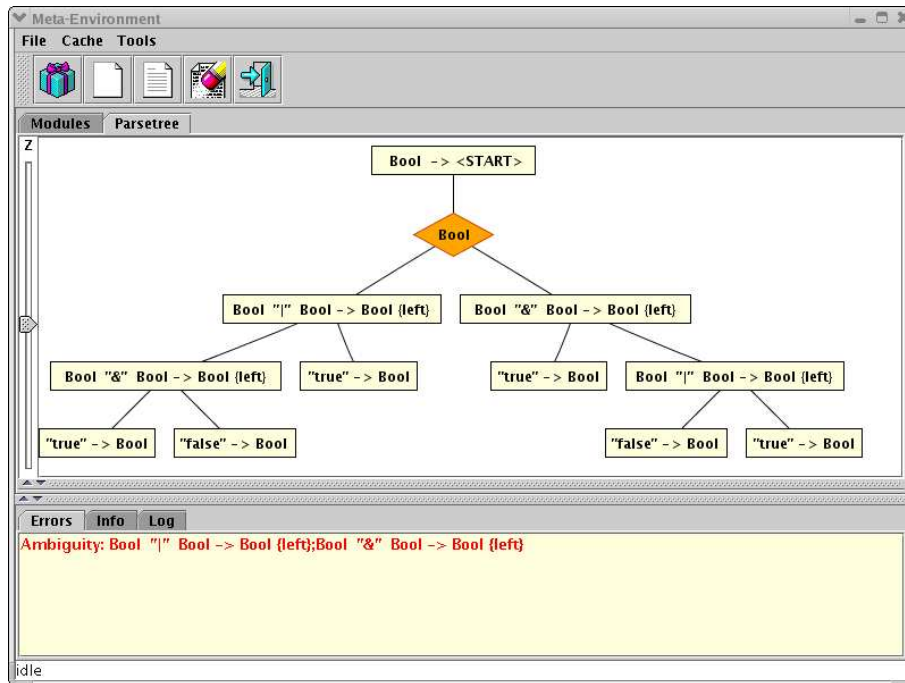


Figure 38: Main window with an ambiguous tree in the parse tree panel

Frequently occurring errors Omitting the LAYOUT definition is one of most common errors made when writing a new specification; always make sure your syntax definitions define at least spaces and newlines to be LAYOUT. In fact, if you want to be sure use whenever possible the predefined module `Whitespace` from the library. Try to do this for as many modules as possible.

- End the editing of your term and the module `Booleans` by selecting the `Exit Emacs` from the `File` menu of the editor.
- Exit the system by pushing the `Quit` entry in the `File` menu of the main window of the ASF+SDF Meta-Environment.

7.4 The Pico Specification

More features of the ASF+SDF Meta-Environment can be studied by looking at the Pico specification. Pico is a toy language used for demonstration purposes. We turn our attention to the complete Pico language.

- Leave the ASF+SDF Meta-Environment. This is done by selecting `Quit` from the `File` menu (Section 3.1).
- Restart the ASF+SDF Meta-Environment by entering `meta` (Section 2) at the command line.
- Add the module `Pico-syntax` by selecting the `File` menu, and selecting the `Open Library Module...` button. In the dialog window that appears, click on `languages` and push the `Open` button. In the subdirectory, choose `pico` via double clicking or selecting and pushing `Open`. Select the single subdirectory `syntax` and finally choose the file `Pico-Syntax.sdf`.
- As you can see in both the import pane and the module pane (see Figure 39), not only `Pico-syntax` has been added, but also all modules that are directly or transitively imported by `Pico-Syntax`.

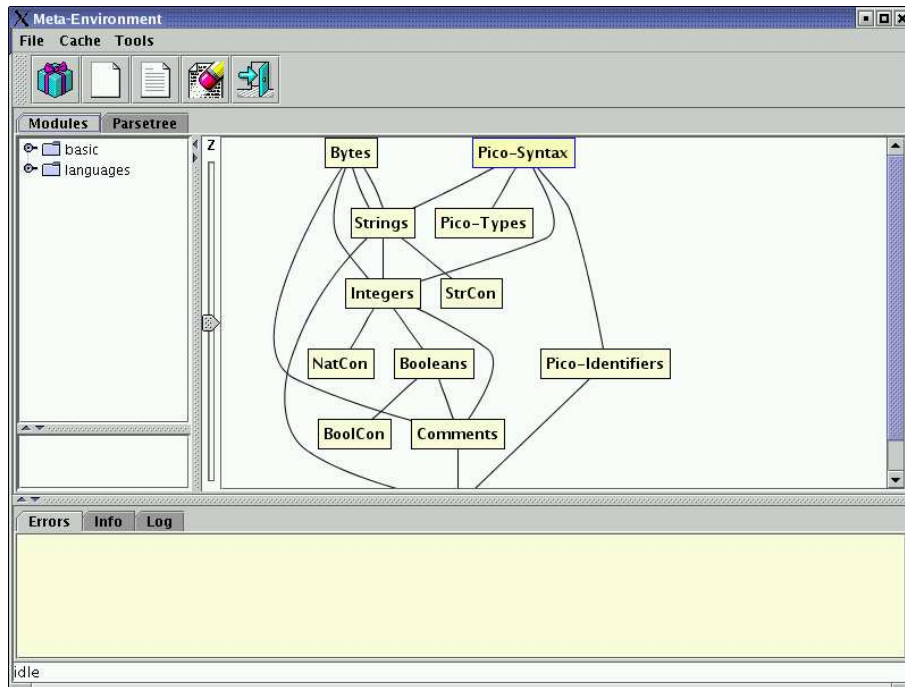


Figure 39: Main window after opening Pico-syntax

7.4.1 The Module Editor for Pico-Syntax

- Open an editor for the syntax of Pico-Syntax (using the `Edit Syntax` button).

A Pico program consists of the word ‘begin’, a declaration section, a series of zero or more statements, and the word ‘end’. The declaration section consists of the word ‘declare’, a list of zero or more tuples ‘*identifier* : *type*’ and a semi-colon ‘;’. Types are ‘string’ and ‘natural’. There are three kinds of statements: assignments, if-then-else statements and while-loops. The Pico language has also expressions for adding and subtracting natural numbers and for concatenating strings.

Notes:

- In the context-free section the list constructs ‘{ID-TYPE " , " }*’ and ‘{STATEMENT " ; " }*’ are used.
- The Pico-syntax module contains no equations.

7.4.2 A Term Editor for Pico-syntax

- Open a term-editor for the Pico-program ‘`fac.pico`’: select Pico-syntax in the module pane and push the `term` button in the button pane. A dialog window pops up and type ‘`fac.pico`’ as name of the term.
- Press the `Parse` button in the `Actions` menu of the editor. As a result, `fac.pico` is parsed.
- Press the `Reduce` button in the `Actions` menu of the editor.

This has the following effects:

- The term in the editor is parsed.



Figure 40: Pico specific pull-down menu `Pico`

- All the equations that are valid for this editor are parsed and compiled into a rewrite system. In this case that means the equations of the imported modules `basic/Bytes`, `basic/Booleans`, `basic/Integers`, and `basic/Strings`. This may take some time.
- The term in the editor is reduced. As no equation can be applied to reduce this term, the term itself is returned in the shell window from which the Meta-Environment has been started.

Reducing a term for the second time is notably faster: the equations have been processed already. If you are curious what is going, have a look at the status field at the bottom of the main window. It reveals the steps that are necessary to arrive at a specification which can be interpreted.

- Verify this by pushing the `Reduce` button once more.

Next to the pull-down menu `Move` in the term editor for `fac.pico` you will see a pull-down menu `Pico`(see Figure 40).

Push the `TypeCheck` button. This has the following effects:

- The `Pico-typecheck` specification is loaded in the environment.
- The term in `fac.pico` is automatically extended with the function `tcp`. The function `tcp` (for type check program), applies the typing rules for the Pico language to its single argument: a complete Pico program. The result is `true` or `false`, depending on whether the Pico program is properly typed.
- All equations of `Pico-typecheck` and its imported modules are being compiled.
- The term `fac.pico` with the function `tcp` is reduced using the equations of `Pico-typecheck`.
- The same effect can be achieved by opening a term editor over the module `Pico-typecheck` and choose for instance the term `fac.ptc`, and push the `Reduce` button in this launched term editor.

Typechecking a term for the second time is notably faster, the modules have been added already and the equations have been compiled.

- Verify this, by pushing `TypeCheck` button in `fac.pico` once more.
- Make some modifications to '`fac.pico`' in the term-editor. Typecheck the modified program.
- Open term-editors with other pico programs ('`small.pico`', '`big.pico`') or create your own program. Typecheck these programs.

The evaluation of Pico programs is achieved in a similar fashion, by pushing the `Evaluate` button. The evaluation rules are defined in the module '`Pico-eval`'. Applications of the evaluation function '`evp`' can be found in '`small.pev`', '`fac.pev`', and '`big.pev`'.

- Repeat the steps described above for typechecking, now for the evaluation of Pico programs.

The same can be done once again for code generation, via the button `Compile`.

7.4.3 More Exercises to Study the Pico Specification

- Study other modules in the specification. The modules `Pico-typecheck` and `Pico-eval` are explained in the next sections.
- Add a repeat statement `'repeat {STATEMENT ";" }* until EXP'` to `Pico-syntax`, add typecheck equations to `Pico-typecheck`, and eval-equations to `Pico-eval`, for this new statement.
- Add your own module to the specification.
- Make your own specification. Create a new directory for each specification.

7.4.4 Module Pico-typecheck

- Open an editor for the syntax of `Pico-typecheck`.

The function `'tcp'` is defined for typechecking Pico-programs. Variants of this function exist for typechecking various parts of a Pico program. The typechecking of the declarations yields a type-environment: a table of identifiers and their types. This type-environment, and the `'lookup'` function is specified in the module `Type-environments`. The typechecking of statements uses a type-environment and yields a Boolean value.

- Open an editor for the equations of `Pico-typecheck`.

The equations define how a Pico-program is typechecked. Equation `[Tc1]` says that the typechecking of a program is `'true'` if the typechecking of the Series in the type-environments, `'tcd(Decls)'`, is `'true'`.

Equations `[Tc2]`, `[Tc3a]`, `[Tc3b]`, `[Tc4a]`, `[Tc4b]` specify how a type-environment is constructed, when the declarations are typechecked.

Equations `[Tc5a]` and `[Tc5b]` specify the typechecking of a, possibly empty, list of statements. Equations `[Tc6a]` through `[default-Tc6]` specify how the three kinds of Statements are typechecked using the information from the type-environment.

The rest of the equations deal with the typechecking of expressions.

7.4.5 Module Pico-eval

- Open an editor for the syntax of `Pico-eval`.

The functions `'evp'` and variants are defined for describing the dynamic semantics of Pico. The result of evaluation is a value-environment: a table of identifiers and values with the final values of the declared identifiers. (Note that Pico does not have an output-statement.)

- Open an editor for the equations of `Pico-eval`.

The equations define how a program is evaluated. Equation `[Ev1]` says that the evaluation of a program is the evaluation of the Series in the value-environments, `'evs(Decls)'`.

Equations `[Ev2]` through `[Ev3c]` specify how a value-environment is constructed, when the declarations are evaluated. Identifiers of type `'natural'` get value `'0'`, Identifiers of type `'string'` get value `'"'` (the empty-string).

Equations `[Ev4a]` and `[Ev4b]` specify the evaluation of a, possibly empty, list of statements. Equations `[Ev5a]` through `[Ev5e]` specify how the three kinds of statements are evaluated using the information from the value-environment. Evaluating statements means updating the value-environment.

The rest of the equations deal with the evaluation of expressions. Evaluating an expression results in a value.

7.4.6 Module Pico-compile

- Open an editor for the syntax of `Pico-compile`.

The functions `trp` and variants are defined for compiling Pico into a stack machine based assembler. The result of evaluation is a list of assembler instructions.

- Open an editor for the equations of `Pico-compile`.

The equations define how a program is compiled. Equation `[Tr1]` says that the compilation of a program is the compilation of the `Decls` concatenated with the instructions resulted from the compilation of the `Series`.

Equations `[Tr2]` through `[Tr3c]` specify how the translation of the declarations is performed. Identifiers `Id` of type `'natural'` result in the instruction `dclnat Id`, Identifiers of type `'string'` result in the instruction `dclstr Id`.

Equations `[Tr4a]` and `[Tr4b]` specify the compilation of a, possibly empty, list of statements. Equations `[Tr5a]` through `[Tr5e]` specify how the three kinds of statements are translated. During the translation process a variable `Label` is used which contains the last used label and which is updated via the function `nextlabel`. The labels are used to direct the flow of control in case of the conditional and while loop.

The rest of the equations deal with the translation of expressions.

References

- [1] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.
- [2] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.
- [3] M. G. J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new ASF+SDF meta-environment. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer, 1997.
- [4] M.G.J. van den Brand, A. van Deursen, Jan Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, LNCS, pages 365–370. Springer, 2001.
- [5] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van den Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of LNCS. Springer-Verlag, 1996.
- [6] M.G.J. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: The asf+sdf compiler. July 2000. See: CoRR E-print Server cs.PL/0007008.
- [7] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.
- [8] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.
- [9] M.G.J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual*.
- [10] M.G.J. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Compiler Construction (CC '99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

- [11] M.G.J. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. To appear.
- [12] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with type-safe traversal functions. In B. Gramlich and S. Lucas, editors, *Second International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
- [13] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 2003. to appear.
- [14] M.G.J. van den Brand, A.S. Klusener, L. Moonen, and J.J. Vinju. Generalized parsing and term rewriting - semantics directed disambiguation. In B. Bryant and J. Saraiva, editors, *Third Workshop on Language Descriptions Tools and Applications (LDTA 2003)*, *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.
- [15] M.G.J. van den Brand and C. Ringeissen. ASF+SDF parsing tools applied to ELAN. In *Third International Workshop on Rewriting Logic and Applications*, ENTCS, 2000.
- [16] M.G.J. van den Brand and J. Scheerder. Development of Parsing Tools for CASL using Generic Language Technology. In D. Bert, C. Choppy, and P. Mosses, editors, *Workshop on Algebraic Development Techniques (WADT'99)*, volume 1827 of *LNCS*. Springer-Verlag, 2000.
- [17] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In N. Horspool, editor, *Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2002.
- [18] M.G.J. van den Brand and J. Vinju. Rewriting with layout. In *Workshop on Rule-based Programming (PLI2000)*, 2000.
- [19] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.
- [20] H. de Jong and P. Olivier. *ATerm Library User Manual*.
- [21] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
- [22] J. Heering. Application software, domain-specific languages, and language design assistants. May 2000. see: CoRR E-print Server cs.PL/0005002.
- [23] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
- [24] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In *ESPRIT '85: Status Report of Continuing Work*, pages 467–477. North-Holland, 1986. Part I.
- [25] J. Heering and P. Klint. Towards monolingual programming environments. *ACM Transactions on Programming Languages and Systems*, 7(2):183–213, 1985.
- [26] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, March 2000. also: ACM CoRR E-print Server xxx.lanl.gov/abs/cs.PL/9911001.
- [27] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.
- [28] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.
- [29] P. Klint. *A Guide to ToolBus Programming*. Included in ToolBus distribution.

- [30] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
- [31] P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, University of Amsterdam, 2000.
- [32] J. Rekers. A parser generator for finitely ambiguous context-free grammars. In *Conference Proceedings of Computing Science in the Netherlands, CSN'87*, pages 69–86, Amsterdam, 1987. SION.
- [33] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
- [34] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Notices*, 26(5):59–66, 1991.
- [35] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.