Using TIDE to Debug ASF+SDF on Multiple Levels

Bas Cornelissen
December 2004

Master's thesis Computer Science

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Sectie: Programmatuur
Richting: Computing, System Architecture and Programming
Afstudeerdocent: Prof. dr. Paul Klint
Begeleider: Drs. Jurgen Vinju

Acknowledgements

Of all the people that have made it possible for me to work on this thesis, I would like to thank Paul Klint and Jurgen Vinju in particular. Even though I proved to be a slow starter (as usual), they remained patient and never got tired of me or my questions - at least, they never gave me that impression :-) Jurgen has been both a source of inspiration and a guru to me, and I wish him the best of luck in finalizing his PhD thesis. Other thanks go out to Hayco de Jong for his personal lectures on C and on the ToolBus in particular, and to Pieter Olivier for thinking along with Jurgen and me. It has been an honor to work with you guys!

Contents

Chapter 1

Introduction

In this thesis, we will study generic debugging in the ASF+SDF Meta-Environment. This is done by using *TIDE*, the *ToolBus Integrated Debugging Environment*, designed five years ago by Olivier [1]. Being a ToolBus-based framework, TIDE is especially applicable in the ASF+SDF Meta-Environment. Although it has been partially documented and applied in the past, we feel that further research may reveal interesting results for debugging ASF+SDF and debugging in general.

In this chapter, the reader is introduced to ASF+SDF, the ToolBus and the ASF+SDF Meta-Environment, which is the main application area of TIDE. We then formulate the problem statements of this thesis, explaining what it is we are trying to accomplish. Finally, we will name some related work on generic debugging.

1.1 Introduction to the ASF+SDF Environment

1.1.1 ASF+SDF

ASF+SDF [2, 3, 4, 5] is a specification formalism used for the definition of languages. An ASF+SDF specification consists of two parts: SDF, which stands for *Syntax Definition Formalism*, and ASF, *Algebraic Specification Formalism.*

A specification's SDF-part defines the syntax of the language. It allows for the definition of both the lexical and the context-free syntax.

The ASF-part describes the language's semantics by providing of a set of equations. These equations are rewrite rules that are meant for reducing a syntactically correct term to its normal form, in accordance with the syntax definition in the SDF-part.

1.1.2 The ToolBus

The ToolBus [6] is a software coordination architecture written in C. It is a middleware that is suitable for component-based environments. A scripting language based on process algebra is used to let tools communicate with each other.

Associated with each tool is an *adapter*, which forms the intermediary between the tool and its ToolBus *process*. Tools can only communicate with each other through these processes (figure 1.1).

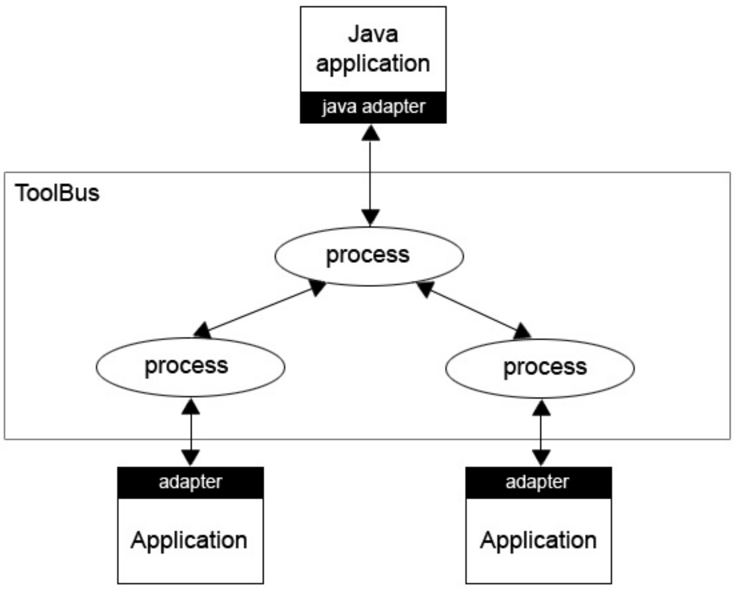


Figure 1.1: The ToolBus.

1.1.3 The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [7, 8] is an interactive programming environment for the development of ASF+SDF specifications. It uses the ToolBus as a coordination architecture and features a user interface that displays the module hierarchy of the module being developed. A separate tab for TIDE's debugging environment has recently been added.

The user may open existing modules or create new ones, and the editors offer support for the syntax checking of both ASF and SDF files. An ASF+SDF specification's correctness can be verified by providing an input term in the term editor. In this editor, one may perform a syntax check on the term and have the term reduced to its normal form by use of the specification's rewrite rules (figure 1.2). Another option is to reduce the term with debugging support, which invokes the TIDE interface and allows the user to debug the specification at hand.

1.2 Problem statement

Writing a faultless ASF+SDF specification is not easy. While we are dealing with formal language, it is essentially a programming language in which mistakes are easily made. This clarifies that there is a substantial need for debugging support in the Meta-Environment.

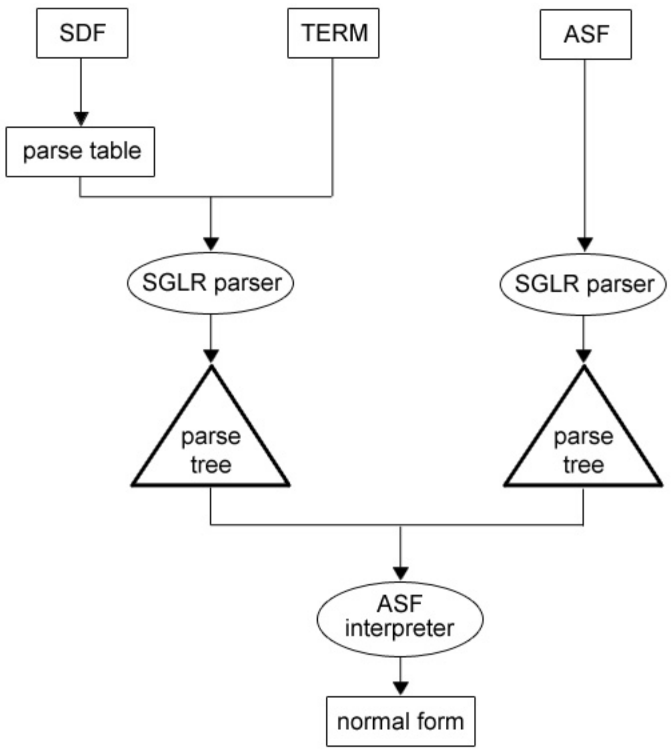


Figure 1.2: Reducing an input term to its normal form.

In this thesis, we explore the ways in which both ASF+SDF and languages specified in ASF+SDF are debugged. It is important to distinguish between these two, as debugging can be done on more than one abstraction level. To this end we will use a debugging framework called TIDE [1]. This is a ToolBus-integrated framework that is capable of debugging both ASF+SDF and, theoretically, any other language. We will touch upon the following problems:

- **Documentation**
  The TIDE framework lacks a thorough *technical documentation.* Although a number aspects are clarified in [1], there is need for a complete specification of the framework's design and implementation. Additionally, it would be good to have some sort of tutorial that contains guidelines for debugging ASF+SDF specifications by use of this framework.

- **Generating language-specific debuggers**
  Currently, the framework is merely capable of debugging ASF+SDF itself, and some prototype debuggers for C and Java exist. However, it has been stated that languages specified in ASF+SDF are also eligible for having debugging support. We will explore this matter and attempt to obtain a specific debugger for such a language, and if we succeed in doing so, provide guidelines for creating new debug adapters.

- **Debugging at multiple levels**

Debugging one single process within an application is more or less straightforward. It would be more interesting to investigate to what extent we can debug *multiple* processes, especially if they are *dependent* on each other. TIDE has been designed to debug multiple processes in parallel, but if these processes rely on each other, things get more complicated. An attempt to simultaneously debug such processes with TIDE has not yet been made.

1.3 Related work

Related to the work in this thesis is the DDF project [9], in which a debugger framework is built for domain-specific languages. Using a mapping technique, existing domain-specific language grammars are extended with hooks for them to communicate with a debugger.

The reader is referred to the appropriate sections in [1] for other, similar efforts.

1.4 Overview

- Chapter 2 provides an extensive description of the TIDE framework, and sheds light upon both its design and its implementation.
- The following chapter provides a tutorial for using the framework to debug ASF+SDF in the Meta-Environment.
- In chapter 4 we explain how to extend an application, or a programming language, with debugging support.
- Chapter 5 discusses how to obtain a debugger from a language specification and provides a case study.
- Finally, we present our conclusions in chapter 6.

Chapter 2

The TIDE framework

2.1 Introduction

The need for debugging support in the ASF+SDF Meta-Environment inspired Olivier [1] to design a framework for generic debugging: *TIDE*, the *ToolBus Integrated Debugging Environment*. It was designed to be a framework for debugging *heterogeneous*, *distributed* systems while remaining *language independent*:

- It is applicable in distributed environments, in which multiple applications reside that require debugging.
- In addition to the support offered by most debuggers, in which support is often limited to debugging the communication between components, it can also debug these components themselves and the processes therein.
- Being language independent, it is capable of dealing with the various languages in which applications may be implemented.

By use of the ToolBus as a coordination architecture, TIDE utilizes language independent debugging events that are generated by components that require debugging.

This chapter offers a detailed description of both the concept and the current implementation of the framework. Before explaining the notion of events and *event rules* in section 2.3, we discuss the *debug processes* and *adapters* that are associated with the components being debugged. We then describe the *functions* that are implemented by TIDE (section 2.3.3), both existing and new. What follows in section 2.4 is a brief description of the graphical user interface. Finally, section 2.5 reveals some details on the implementation of the framework.

2.2 Adapters and processes

The TIDE framework is capable of debugging multiple processes simultaneously. This is explained in detail in the case study in chapter 5. In order to keep track of the distinct components being debugged and to be able to distinguish between these components, TIDE utilizes *debug adapters* and *debug processes*.

A tool that needs debugging may connect to TIDE, which will in turn create an adapter for this tool. Afterwards, one or more debug processes are created within this adapter. This concludes the creation of a debugger for this specific component, and execution with debugging support may commence. Note that TIDE's Java part (section 2.5.3) maintains counterparts of the active debug adapters and processes.

The following sections provide details on the ins and outs of debug adapters and debug processes.

2.2.1 Adapters

The ToolBus maintains the connections of the various applications and controls the communication between these applications. If such a component - having been extended with TIDE support - requires debugging, it may connect to TIDE through a *debug adapter*. A debug adapter translates the debugging primitives from the application to the TIDE interface. Every new application or language needs such an adapter to receive debugging support (figure 2.3).

As such an application applies for debugging support, TIDE creates a new debug adapter and creates a ToolBus connection between the adapter and the application. This is an adapter-specific connection that is used for the debugging of this application in particular. The adapter then allocates one or more new debug processes that will consequently register any custom functions for which the component has the necessary implementations (see section 2.3.3.

Certain applications, for instance if they are multithreaded, may require multiple debug processes. Though having never been tested with such applications, TIDE provides the necessary support by allowing more than one debug process to reside within the adapter simultaneously. Nevertheless, we will only consider applications featuring *one single process* in this thesis.

As the adapter and the process have been created, it is communicated to TIDE's graphical interface that a new adapter has been connected. The user may now start a debugging session, in which all ToolBus communications on one hand and the component's debugging events on the other are handled by the debug adapter.

2.2.2 Debug processes

Associated with each debuggable process in the component is a debug process in the component's debug adapter. Such a debug process contains actual information on the state in which the application currently resides: for example, the current point of execution. This is best illustrated by the corresponding code fragment:

```
typedef struct
{
  int         id;
  int         cid;
  char        *name;
  int         state;
  int         stack_level;
  int         start_level;
  TA_Location cpe;
  TA_Rule     *enabled_rules[MAX_PORT_TYPES];
```

```
    TA_Rule      rules[MAX_EVENT_RULES];
  } TA_Process;
```

Most fields are used by any type of debug process; the stack level and start level however are only applicable in case of a stack-based execution.

As the execution has finished, the corresponding debug process is destroyed.

2.3 Events, ports and rules

An adapter communicates events to TIDE for it to keep track of the execution of a program. Based on these events, TIDE may take action: for instance, if a breakpoint is encountered, execution must be halted and control must be returned to the user while awaiting further orders.

In order to bind such actions to certain events, the notion of *event rules* is introduced. An event rule is process-specific in that it belongs to one certain debuggable process. It consists of an *event port*, a *condition*, and one or more *actions*. The event port indicates at which point during the execution the associated rule should be *activated* (table 2.1). Upon activation of a rule, its condition is evaluated and if the condition holds true, the rule is *triggered* and its actions are consequently performed (figure 2.1). During execution, event

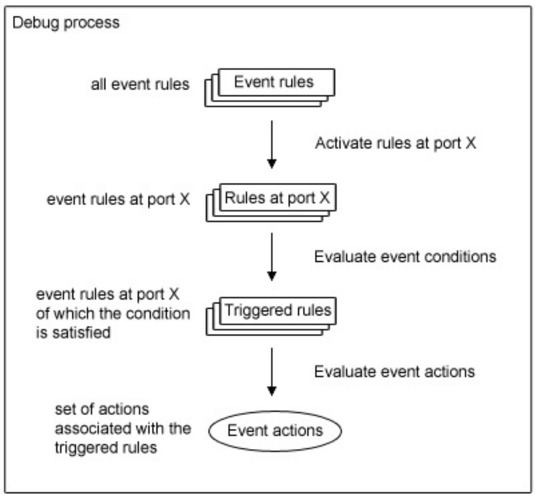


Figure 2.1: Event rules.

rules may be enabled, disabled, deleted or modified by the user or the application. Sections 2.3.1 through 2.3.3 provide detailed descriptions on each of the event rule's components, whereas some examples are given in section 2.3.4.

2.3.1 Event ports

Event ports are used to bind events to event rules. Which port is currently active in the debugging session depends on the application's debug adapter, which is capable of setting it to any of the values in table 2.1. When a certain port has become active, e.g. when execution has stopped, it is verified whether the current process contains any enabled rules that are bound to this port. If such rules exist, they are activated.

One can observe in the code fragment in section 2.2.2 that indeed a set of

event port	activated when
`started`	execution is continued
`stopped`	execution has halted
`step`	a statement is executed

Table 2.1: The event ports that are currently supported.

event rules is associated with each debug process. Each execution step results in its debug process being changed, and its *event rules* being operated on. For instance, as the execution's port changes, TIDE searches the process' array of enabled rules so as to activate the rules that are on the new port.

2.3.2 Event conditions

Upon activation of an event rule, the associated event condition is evaluated to determine whether the rule should be triggered. This condition is an expression that is evaluated by the process' debug adapter. An event rule may simply put *true* as the condition so the action is always performed on the corresponding port, for example highlighting the current point of execution when there is no activity. However, one may also use functions for conditions; more details on functions are described in the next section.

2.3.3 Functions and actions

TIDE offers built-in functions that may be used for conditions and actions. These are predefined expressions that are evaluated by a special function called `eval()`. Arguments may be passed on to these functions, as is shown in table 2.2. The list of currently implemented functions is far from complete. One can

function	return type	returns
`true`	boolean	`true`
`false`	boolean	`false`
`cpe`	location	current point of execution
`location(loc)`	boolean	whether `loc` is part of the current cpe
`stack-depth`	integer	current depth of the stack
`start-depth`	integer	stackdepth as was on the last call of `resume`

Table 2.2: Predefined built-in functions.

think of many more functions that come in handy when creating custom event rules, such as functions that determine equalities between integers. We have contributed in the expansion of the pre-defined set by adding more comparison operations on integers, one of which is exemplified in the next chapter. Since the user is required to know the exact syntax of these functions in order to apply them, they are explicitly listed in table 2.3.

In addition to these predefined functions, a tool may also register custom

function	return type	returns
`equal(t1,t2)`	boolean	whether `t1` is equal to `t2`
`higher(t1,t2)`	boolean	whether `t1` is higher than `t2`
`less(t1,t2)`	boolean	whether `t1` is less than `t2`
`higher-equal(t1,t2)`	boolean	whether `t1` is higher than or equal to `t2`
`less-equal(t1,t2)`	boolean	whether `t1` is less than or equal to `t2`

Table 2.3: Additional functions.

functions for its debug processes to use. These functions need to be implemented by the debug adapters themselves. An example of such a custom function is `source-var`, which is used for variable viewing during the debugging process. Since every application stores variables and their actual values in its own way, each component needs to provide a method for TIDE to collect these values.

Functions can be used for both conditions and actions. Examples are given in section 2.3.4.

2.3.4 Examples of event rules

We now provide some examples of event rules. These will clarify the use of event rules in various aspects of the debugging process.

We obviously want the *current point of execution* to be highlighted between every step. To achieve this, TIDE creates the following rule by default:

port	`stopped`
condition	`true`
action	`cpe`

This implies that whenever execution is halted, the condition `true` is evaluated, obviously returning true. The rule is consequently triggered and the action `cpe` is executed. This is a built-in function resulting in the evaluation of the current point of execution. By means of a ToolBus event, the result is then received by the GUI, in which the corresponding location is highlighted.

TIDE implements *breakpoints* by constructing a rule from the source location that was clicked on:

port	`step`
condition	`location(loc)`
action	`break`

As an instruction is about to be executed, the corresponding debug adapter evaluates whether the point of execution associated with *loc* has been reached, halting execution if this is the case. This is achieved by use of the built-in function `location()`, which determines whether its argument is part of the actual cpe.

A final example illustrates how the actual values of variables are retrieved. Recall from the previous section the custom function `source-var()`. By use of this function, TIDE creates a rule of the following form whenever the value of a variable is requested:

port	`stopped`
condition	`true`
action	`source-var(loc)`

As soon as this rule has been created for the requested variable, the value of this variable is continually displayed and retrieved every time the execution halts. The value is passed to the GUI and, as will become clear in section 2.4, it is displayed in a popup. Closing this popup will automatically delete the rule.

2.4 GUI

TIDE's graphical user interface is where the user controls the debugging processes of one or more applications. It has been integrated into the most recent release of the ASF+SDF Meta-Environment, residing under the *Debugging*-tab.

The left portion of the GUI displays the currently connected adapters and

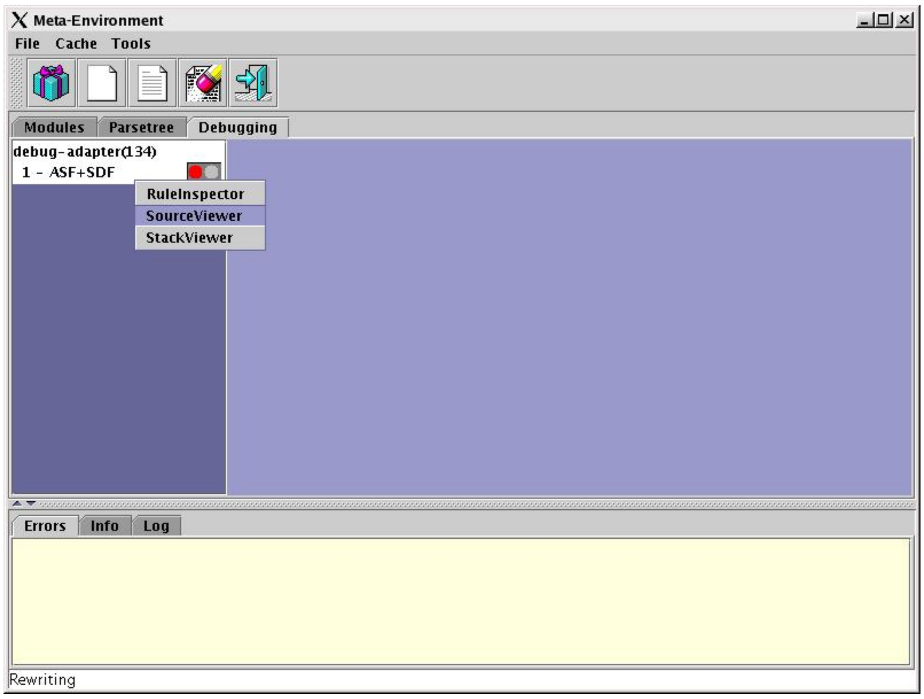


Figure 2.2: TIDE's user interface in the ASF+SDF Meta-Environment. Depicted is a recently connected adapter hosting a process called *ASF+SDF*.

processes that are currently connected, whereas the rightmost part contains the TIDE tools that are active for these processes. These utilities are process-specific and can be invoked by right-clicking on a processname and selecting the appropriate tool (figure 2.2). Currently implemented are three types of tools:

- **SourceViewer**
  Used for execution control such as stepping and setting breakpoints.

- **StackViewer**
  Allows the user to inspect and operate on the execution stack.
- **RuleInspector**
  Grants access to the currently active rules for the purpose of inspection or modification.

Pictures of the tools can be found in figures 3.1 through 3.4 in the next chapter.

Each tool is a producer of event rules and, at the same time, a consumer of the event actions' results. Some of the tools are useless when debugging certain applications; the stackviewer for instance is not applicable if the component being debugged has no stack-based execution model.

Whenever the user starts debugging an application, the Debugging-tab automatically jumps on top as soon as the debug adapter has connected. The reader is referred to the next chapter for guidelines on using the GUI and its tools, as they are best explained by way of an example debugging session.

2.5 Implementation

The implementation of TIDE comsists of ToolBus scripts, C libraries, Java libraries and a Java-based GUI. Communication between the debug adapters and the GUI is handled by the ToolBus. The GUI has recently been integrated into the ASF+SDF Meta-Environment for users to apply it whenever they consider it necessary.

Section 2.5.1 provides information on the ToolBus processes that are associated with TIDE. Follows in section 2.5.2 is a description of the most important C library, called `tide-adapter.c`. Finally, section 2.5.3 briefly discusses the class hierarchy and interaction in TIDE's Java part.

2.5.1 ToolBus interaction

TIDE utilizes ToolBus scripts for its processes to communicate with each other. Figure 2.3 illustrates the relevant processes and tools when debugging a single application. We shall now provide brief descriptions for each of these processes.

- **CONTROL**
  This process generally utilizes the ToolBus' java-adapter to invoke TIDE's graphical user interface[1]. The GUI then connects to the `TOOL-CONNECT`-process. `CONTROL` offers the possibility to embed TIDE in another GUI, or to run as a separate application.

- **TOOL-CONNECT**
  `TOOL-CONNECT` waits for a debug tool (generally TIDE's GUI) to connect, after which a new process `TOOL` is consequently created. A name of the form `debug-tool(cid)` is henceforth associated with this process.

[1]In the case of a custom debug tool (e.g. an alternate GUI), passive mode may be enabled for the `CONTROL`-process to accept incoming tool connections instead of invoking the standard GUI.

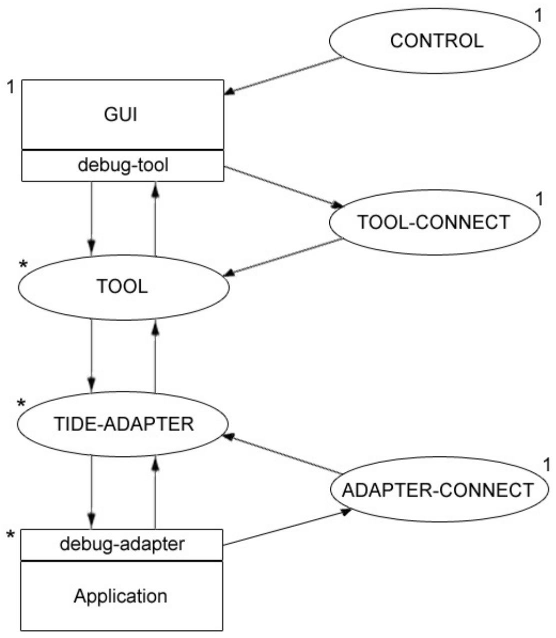


Figure 2.3: The relations between TIDE's ToolBus processes, the GUI, and the applications being debugged. It has been indicated whether multiple instances of the processes and tools can exist.

- **TOOL(debug-tool)**
  Upon creation of the `TOOL`-process that belongs to `debug-tool`, TIDE has been initialized and is ready to accept connections from applications that need debugging. Any messages and events that are destined for or originating from the GUI are handled by `TOOL`.

- **TIDE-ADAPTER-CONNECT**
  `TIDE-ADAPTER-CONNECT` awaits the connection of a debug adapter, after which it creates a new process called `TIDE-ADAPTER`. The `debug-adapter`'s identification is supplied as a parameter.

- **TIDE-ADAPTER(debug-adapter)**
  A new `TIDE-ADAPTER`-process first informs any `TOOL`-processes of its existence by sending an appropriate message. It then awaits further orders, which may come from its application or from the `TOOL`-process.

Summarizing the above, the ToolBus scripts implement a simple router between TIDE's interface, the applications being debugged, and the various debug tools and debug adapters.

As an example, consider the situation in which the GUI enables an event rule. The associated debug-tool posts a ToolBus event that is picked up by its `TOOL`-process. The relevant piece of communication in `TOOL` looks like

```
rec-event(T, enable-rule(Process?, Rule?)) .
snd-msg(enable-rule(Process, Rule)) .
```

```
rec-msg(rule-enabled(Process)) .
snd-ack-event(T, rule-enabled(Process, Rule))
```

in which `Process` is a tuple of a debug adapter and a process id within this adapter.

The corresponding messages in `TIDE-ADAPTER` are

```
rec-msg(enable-rule(proc(A, Pid?), Rule?)) .
snd-eval(A, enable-rule(Pid, Rule)) .
rec-value(A, rule-enabled) .
snd-msg(rule-enabled(proc(A, Pid)))
```

`TOOL` receives the job from the GUI and consequently sends it to the debug process that the rule belongs to (i.e. to its debug adapter). It then waits for an acknowledgement message from the corresponding `TIDE-ADAPTER`-process. As the TIDE extension of the application reports that the rule has been enabled, the GUI is notified by means of an ack-event.

2.5.2 Adapter library

Applications that are C-based have TIDE's adapter library linked with them as they are extended with debugging support. This library, `tide-adapter.c`, contains all of the functions that are necessary for the application to communicate with TIDE, i.e. to create a debug adapter and a debug process. (Programs written in Java may use the Java equivalent, which is the `DebugAdapter`-class discussed later on.) Auxiliary functions reside in `TA.c`, whereas ToolBus communication is handled by `debug-adapter.c`.

Section 4.3, which deals with TIDE extensions in applications, reveals some details on the adapter library by looking at it from an application's point of view.

2.5.3 The TIDE mainframe

TIDE's Java part consists of roughly 50 Java classes. The implementation conforms to the *Model-View-Controller* (MVC) paradigm: there exists a clear separation between the *model*, the *view* and the *controller*. We provide short descriptions for the most important classes and their relations below (see also figure 2.4).

- **TideMainFrame**
  TIDE's primary class is instantiated the moment TIDE is started. This is the result of the `CONTROL`-process invoking the java-adapter upon the `TideMainFrame`-class. `TideMainFrame` creates the main window and instantiates the `TideControl`-class.

- **TideControl**
  Among other things, this object creates a `TideControlBridge` and a `DebugTool`. It then plugs the `DebugTool` into the ToolBus, resulting in the creation of a new `debug-tool(cid)` that is to be associated with the `TOOL`-process mentioned in section 2.5.1.

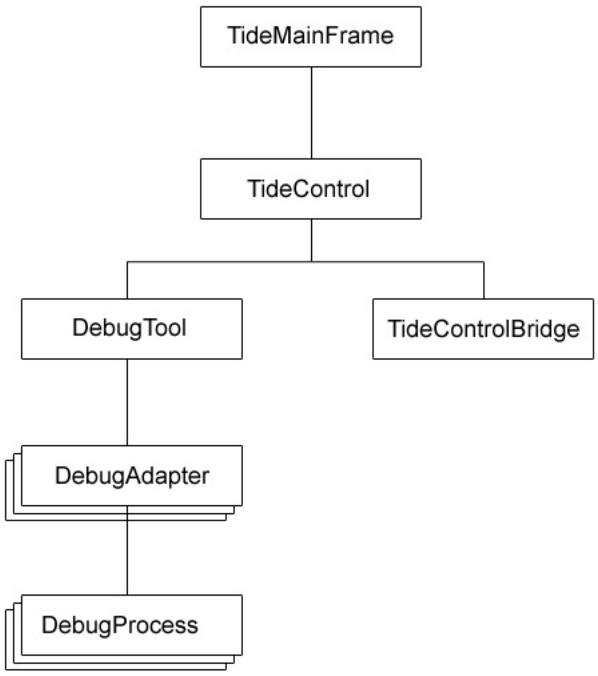


Figure 2.4: The foremost classes in TIDE's Java class hierarchy.

- **DebugTool**
  Together with the `DebugAdapter` described below, this class forms the *controller*-part of the MVC-model. The `DebugTool` is responsible for handling the connection of any new `DebugAdapter`, which it then puts in a list. It also listens for incoming events that its `TOOL`-process receives, such as the creation of a new event rule. This is then delegated to the appropriate `DebugAdapter`.

- **DebugAdapter**
  The `DebugAdapter`-class can be considered as the Java counterpart of the adapter library described in section 2.5.2. Serving as an intermediary between the `DebugTool` and the `DebugProcess` it is, by means of messages and events, continually synchronized with the corresponding debug adapter at application-level.
  As has been stated in section 2.2, each debug adapter maintains a list of debug processes for its application. It is also the case in TIDE's Java part that the adapter serves as an intermediary. When the `DebugTool` receives an event from the ToolBus, it is passed on to the appropriate `DebugAdapter`, which then sends it to the target `DebugProcess`.

- **DebugProcess**
  This the Java version of the debug process at application-level (section 2.2.2). In MVC terms, `DebugProcess` is the *model*: it contains the properties of the debug process it represents, such as the current point of execution and a set of enabled rules. These properties are retrieved or updated through

a chain of responsibility.

- **SourceViewer**, **StackViewer**, **RuleInspector**
  These are the graphical tools that receive input from the user and visualize the current state of the associated debug process. Together, they make for the *view*-part of the architecture. Each of the tools is instantiated by the `DebugProcess`-object that it belongs to.

Chapter 3

Using TIDE to debug ASF+SDF specifications

3.1 Introduction

Having discussed the TIDE framework in chapter 2, we now present what can be considered as a tutorial for using TIDE in the ASF+SDF Meta-Environment. The following sections cover most of the functionality that TIDE offers with respect to debugging on the ASF+SDF level. The reader is referred to chapter 4 for debugging on different layers.

Given the fact that the ASF interpreter was extended with TIDE support when TIDE was developed, and since TIDE has now been integrated into the Meta-Environment, not much needs to be done by the user to start debugging ASF+SDF specifications. By means of illustrations and examples, this chapter explains the features that are currently at the users disposal. These features include execution control, stack operations, viewing the values of variables, and operations on rules. We shall start off by describing how rule rewriting works in ASF+SDF, and in which respect TIDE may offer support in debugging this rewriting process.

3.1.1 ASF execution mechanism

The ASF interpreter is responsible for rewriting an input term by use of an ASF+SDF specification. Before reducing a term, it has been verified whether the term is syntactically correct.

The next phase consists of rewriting the term to its normal form, which is done by use of the rewrite rules that are contained in the specification at hand. The reduction of terms occurs in an *innermost* fashion: considering the term as a tree, the leaves (subterms) are reduced first. As a subterm is being reduced, each of the rewrite rules is tried until a match has been found. The matching rule is then applied, and reduction continues until all terms are in normal form.

In the case of a *conditional* rewrite rule, the conditions are evaluated to determine whether the term being reduced is eligible for reduction by this rule. This leads to terms being put on the interpreter's *execution stack*. The stack can actually grow pretty large if the conditions contain function calls that in themselves require an innermost reduction.

The next section provides an example of a term reduction.

3.1.2 TIDE support in the ASF interpreter

TIDE offers debugging support to the ASF intepreter on the level of rewriting. In the process of reducing a term, the interpreter generates a debugging event whenever it is about to take the next rewriting step. This is the case

- when the left-hand side of the rewrite rule is considered.
- when each of the conditions is evaluated.
- when the right-hand side of the rule is considered.
- when each of the terms in the right-hand side is considered, in an innermost fashion, thus leading to the transformation of the term.

It is important for the reader to know this sequence, as the notion of breakpoints (discussed in section 3.2.2) requires the user to know exactly *in which part of the rule* he or she must put a breakpoint, if execution is to halt at the desired point.

As an example, suppose we have a generic *conditional* rewrite rule of the form

```
[tag] variable == true
================
lhs = rhs
```

and assume that the condition is in fact satisfied. As the ASF interpreter applies this rule while using debugging support, it generates a debugging step whenever it is about to evaluate a new part the rule - thereby shifting TIDE's focus to this new part. The order of evaluation would be:

$$\begin{array}{l}\texttt{variable == true}\\ \texttt{================}\\ \boxed{\texttt{lhs}}\texttt{ = rhs}\end{array}$$

$$\begin{array}{l}\boxed{\texttt{variable == true}}\\ \texttt{================}\\ \texttt{lhs = rhs}\end{array}$$

$$\begin{array}{l}\boxed{\texttt{variable}}\texttt{ == true}\\ \texttt{================}\\ \texttt{lhs = rhs}\end{array}$$

$$\begin{array}{l}\texttt{variable == }\boxed{\texttt{true}}\\ \texttt{================}\\ \texttt{lhs = rhs}\end{array}$$

$$\begin{array}{l}\texttt{variable }\boxed{\texttt{==}}\texttt{ true}\\ \texttt{================}\\ \texttt{lhs = rhs}\end{array}$$

$$\begin{array}{l}\boxed{\texttt{variable == true}}\\ \texttt{================}\\ \texttt{lhs = rhs}\end{array}$$

$$\begin{array}{l}\texttt{variable == true}\\ \texttt{================}\\ \texttt{lhs = }\boxed{\texttt{rhs}}\end{array}$$

What is important is that the last step would not have taken place if the condition were not satisfied. If this were the case, placing a breakpoint at the right-hand side would not have halted execution, implying that the evaluation of this rule would have gone unnoticed.

3.1.3 Getting started

We now discuss a couple of matters before moving onto the debugging part.

Connecting to TIDE

Enabling debugging support in the process of reducing a term is easy. Instead of choosing *reduce* in the editor's *Actions*-menu, one selects *reduce* from the *Debug*-menu. As soon as all pre-execution steps are finished, the *Debugging*-tab becomes active. In the top-left portion of this tab, one can see that a new debug adapter has connected, associated with which is a process named `ASF+SDF`. Debugging can now start at the user's discretion.

Tools

Currently implemented in the TIDE interface are three types of tools. These tools include a sourceviewer, a stackviewer and a rule inspector, all of which can be used when debugging ASF+SDF specifications. Each of these utilities may be invoked by right-clicking on the process of choice and selecting the appropriate tool from the popup menu. The tools provide various features, of which the details are explained in sections 3.2 through 3.5.

3.2 Execution control

An indispensible feature in any self-respecting debugger is rendering the user in control of the execution of the program that is being debugged. This is generally accomplished by stepping through instructions and by use of breakpoints. TIDE supports this functionality and presents the user with three kinds of stepping, and enables the user to set breakpoints in the specification being debugged. Stepping and breakpoint interaction are both available in the sourceviewer.

3.2.1 Stepping

With regard to stepping through the execution path of a program, TIDE distinguishes three types of steps: Single stepping (*step into*), stepping over function calls (*step over*) and stepping up (*step up*). These functions are part of TIDE's graphical interface and a button is associated with each of the steptypes. The functionality is described in the following sections.

Step into

The simplest stepping method is *step into*, which resides under the leftmost of the step buttons in the sourceviewer. Pressing *step into* basically sets the process state to 'running', thereby continuing the execution until either the next step or a breakpoint is encountered. Stepping is implemented by the following rule (section 2.3).

port	`step`
condition	`true`
action	`break`

which reads: *in case a statement is executed, always halt execution.* As execution is halted, the current point of execution is highlighted.

Step over

When an instruction containing a function call is about to be executed, pressing *step over* results in the execution of that function, during which no stepping or intervention is possible. This can be useful in case the user is not interested in stepping through the individual instructions of the function that is being skipped. Typical functions that one may want to step over are library functions, such as `get-location()`.

What happens during a *step over* is that the initial stack level is determined first. Execution is then resumed and as soon as the current stackdepth has become equal to or smaller than the initial depth, the function has apparently finished and consequently a break renders the user back in control. This is accomplished by use of the following rule:

port	`step`
condition	`higher-equal(start-depth, stack-depth)`
action	`break`

As the condition of this rule is evaluated, the built-in function `higher-equal()` is invoked, which takes two arguments and returns true in case the former number equals or exceeds the latter. The arguments in this case are the built-ins `start-depth` and `stack-depth`, which evaluate the process' initial and current stackdepth, respectively.

As has been pointed out in section 3.1.1, ASF has a stack-based execution model, thus enabling it to use the provided stacklevel-based steppings.

Step up

In addition to the former two types of stepping, another variation called *step up* has been constructed that continues execution until the current stackdepth has grown smaller than the initial stackdepth (i.e. the level at which this button was pressed). This feature can be a helping hand when *step into* has accidentally been pressed and the user wants to return to the former stack level.

Among other, similar functions, `less()` has been implemented to support this functionality. The rule therefore reads

port	`step`
condition	`less(stack-depth, start-depth)`
action	`break`

3.2.2 Breakpoints

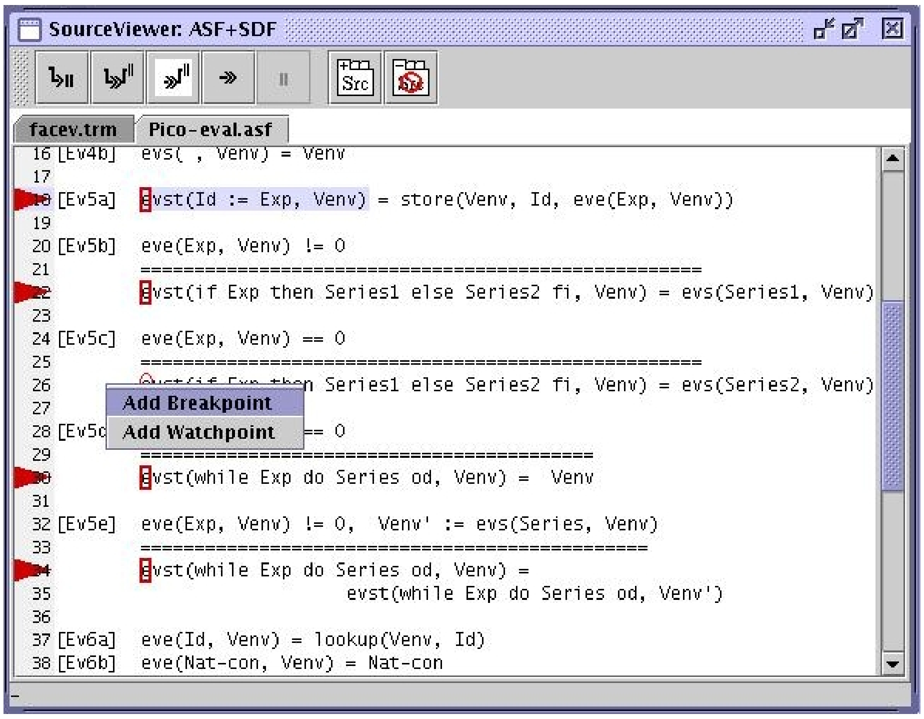
Another way of controlling the execution of a program is by use of breakpoints. A breakpoint is a user-defined fragment in the source code that causes the execution to halt when the current point of execution has reached it. That way the user can have the program execute until a certain point from whence he or she wishes to control the execution manually. This is typically a piece of code that requires close inspection or some debug utility. Section 3.1.2 provides information on the placement of breakpoints in rewrite rules.

The sourceviewer provides the ability to set breakpoints in any piece of source code. This is done by simply right-clicking on the desired spot and selecting *add breakpoint.* An event rule of the following form is consequently created:

port	`step`
condition	`location(loc)`
action	`break`

Figure 3.1 illustrates the use of breakpoints.

Once the desired breakpoints have been set, it is a matter of pushing the *run*-button to let TIDE resume execution until a breakpoint has been encountered. Removal of a breakpoint is done by right-clicking on it and choosing *delete rule*; clicking *edit rule* highlights this breakpoint in the rule inspector for the user to



```
[Ev4b]   evs( , Venv) = Venv

[Ev5a]   evst(Id := Exp, Venv) = store(Venv, Id, eve(Exp, Venv))

[Ev5b]   eve(Exp, Venv) != 0
         ===================================================
         evst(if Exp then Series1 else Series2 fi, Venv) = evs(Series1, Venv)

[Ev5c]   eve(Exp, Venv) == 0
         ===================================================
         evst(if Exp then Series1 else Series2 fi, Venv) = evs(Series2, Venv)

[Ev5c                  == 0
         ==========================================
         evst(while Exp do Series od, Venv) =  Venv

[Ev5e]   eve(Exp, Venv) != 0,  Venv' := evs(Series, Venv)
         ===============================================
         evst(while Exp do Series od, Venv) =
                           evst(while Exp do Series od, Venv')

[Ev6a]   eve(Id, Venv) = lookup(Venv, Id)
[Ev6b]   eve(Nat-con, Venv) = Nat-con
```

Figure 3.1: Inserting breakpoints in the sourceviewer.

alter it. The rule inspector is also the best option in case multiple breakpoints need to be removed at once.

3.3 Stack viewing and unwinding

As the execution of an input term is started, more and more instructions are placed on the stack. Associated with each of these instructions is a *stackframe* which contains the frame's name, its stackdepth, and its location in the corresponding source file. When the stackviewer is spawned, a list containing all current stackframes is generated. Selecting a stackframe provides its fields for the user to inspect.

Figure 3.2 shows an example in which the stackframe that corresponds with a rewrite rule called `[Ev5d]` has been highlighted. This rule reads

```
[Ev5d]  eve(Exp, Venv) == 0
        ==========================================
        evst(while Exp do Series od, Venv) =  Venv
```

and we observe that the stackviewer displays the values of the variables involved, e.g. `Exp`'s value is `input-1`, which is the condition for this while-loop. If we take a step further, the ASF interpreter will shift its focus to this rule's condition so as to evaluate it in an innermost fashion. This results in the creation of a new stackframe called `*innermost*` (figure 3.3). Selecting this new frame, we see that it is now evaluating the leftmost argument of the `eve()` function that

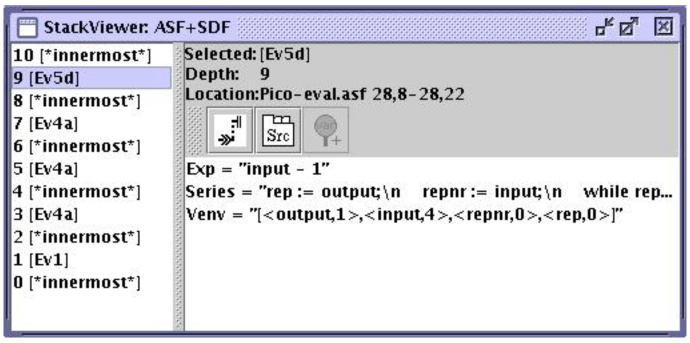

```
StackViewer: ASF+SDF
10 [*innermost*]      Selected: [Ev5d]
9 [Ev5d]              Depth:    9
8 [*innermost*]       Location:Pico-eval.asf 28,8-28,22
7 [Ev4a]
6 [*innermost*]
5 [Ev4a]              Exp = "input - 1"
4 [*innermost*]       Series = "rep := output;\n    repnr := input;\n    while rep...
3 [Ev4a]              Venv = "[<output,1>,<input,4>,<repnr,0>,<rep,0>]"
2 [*innermost*]
1 [Ev1]
0 [*innermost*]
```

Figure 3.2: Stack viewing. The stackframe associated with a rewrite rule has been selected.

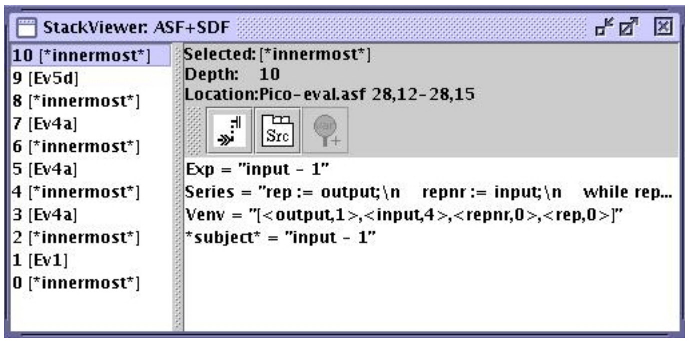

```
StackViewer: ASF+SDF
10 [*innermost*]      Selected: [*innermost*]
9 [Ev5d]              Depth:    10
8 [*innermost*]       Location:Pico-eval.asf 28,12-28,15
7 [Ev4a]
6 [*innermost*]
5 [Ev4a]              Exp = "input - 1"
4 [*innermost*]       Series = "rep := output;\n    repnr := input;\n    while rep...
3 [Ev4a]              Venv = "[<output,1>,<input,4>,<repnr,0>,<rep,0>]"
2 [*innermost*]       *subject* = "input - 1"
1 [Ev1]
0 [*innermost*]
```

Figure 3.3: Selection of a so-called *innermost* frame.

is being called in the rule's condition. This is indicated by `*subject*`: the left-most argument is `Exp`, which corresponds with `input-1` in the input program.

An extremely useful feature is the ability to *unwind* the stack up to a certain stackframe. If a stackframe is selected and this button is pressed, execution will automatically continue until the chosen stackframe has reached the top of the stack, i.e. the current stackdepth has become equal to the selected depth. (This is similar to stepping up, except that the target stacklevel is user-defined.)

The associated rule reads

port	`step`
condition	`higher-equal(frame-depth, stack-level)`
action	`break`

in which `frame-depth` is the depth of the chosen stackframe. Looking at this from an ASF+SDF perspective, we are able to automatically continue execution until we have returned at the desired *rewrite rule*.

Note that any resident breakpoints in the source are potentially responsible for a premature halting of a stack unwinding operation.

3.4 Monitoring variables

Another of the sourceviewer's features is viewing the actual values of variables in the corresponding source. This is done by simply doubleclicking on an alleged variable. If there exists no variable associated with this location, an error dialog is displayed. Otherwise, a popup appears that displays the actual value of the variable.

The event rule for this feature reads

port	`stopped`
condition	`true`
action	`source-var(loc)`

in which `loc` is the location as is on the user's screen, and `source-var()` the registered function that retrieves the value of the variable at that position.

Instead of removing the popup, the user may drag the popup to a suitable location enabling him or her to monitor this variable; the value is automatically updated every step.

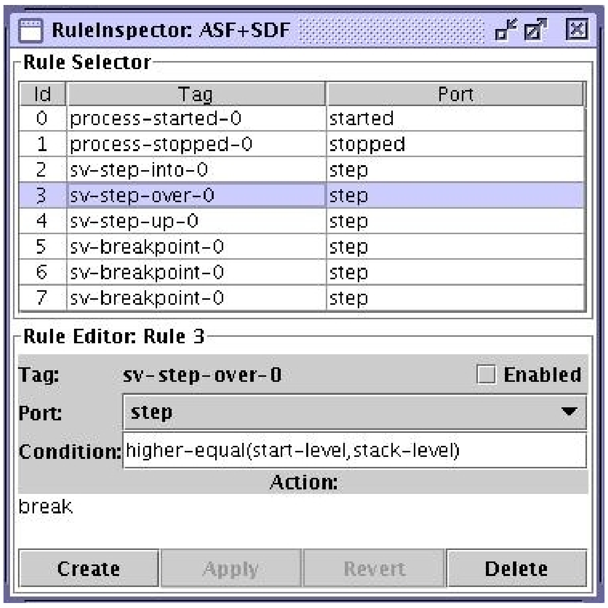
3.5 Rule viewing and manipulation

As has been described in section 2.3, the way in which the program being debugged executes largely depends on the rules that are currently active in the associated TIDE process(es). Most of the rules are created, modified, deleted or triggered automatically: these include the standard rules that either resume or halt execution at ports `started` and `stopped`, respectively. Another example is the *step into* rule that is created as soon as a sourceviewer is spawned.

Other rules are (indirectly) constructed at request, e.g. rules associated with breakpoints.

By use of the rule inspector on the other hand, the user can have a *direct* influence on the rules that are considered during the current debugging process. Invoking a rule inspector for a certain debug process presents the user with a list of rules that currently exist in that process (figure 3.4).

For each of the rules the tag, the port and the condition are displayed. They can be modified, deleted or enabled, and new rules may be created by supplying the necessary fields. Refer to sections 2.3.1 and 2.3.3 for a list of currently supported ports and built-in functions.



Id	Tag	Port
0	process-started-0	started
1	process-stopped-0	stopped
2	sv-step-into-0	step
3	sv-step-over-0	step
4	sv-step-up-0	step
5	sv-breakpoint-0	step
6	sv-breakpoint-0	step
7	sv-breakpoint-0	step

Tag: sv-step-over-0 ☐ Enabled

Port: step

Condition: higher-equal(start-level,stack-level)

Action:

break

Figure 3.4: The rule inspector.

Chapter 4

Extending interpreters with debugging support

4.1 Introduction

The previous chapter has illustrated the use of TIDE in debugging ASF+SDF specifications on the ASF+SDF level. Debugging applications on this level has been made possible by the fact that the ASF interpreter is extended with TIDE support. We will now show that such applications may also be debugged on the *application level* by adding debugging support to their implementations.

The recipe for extending an interpreter or runtime system with debugging support basically consists of three steps:

- Verifying whether the requirements are met.
- Integrating TIDE's adapter library into the interpreter's architecture and establishing a communication layer.
- Placement of debugging steps near relevant events within the interpreter.

We start off by reviewing these ingredients in sections 4.2 through 4.4. The guidelines we present apply to the situation in which the application being debugged is *written in C*. In case the application is written in some other language, TIDE's adapter library must be ported to that language.

4.2 Requirements

TIDE offers both basic and advanced features to debuggable applications. The advanced functionality can only be used in case the application can provide the necessary implementation. As Olivier mentions in [1] the TIDE framework *does* however require a set of *basic features*. Debugging support for a component is not possible in case the component lacks these properties. The basic features include:

- Support for the notion of a current point of execution in each process being debugged.

- Being able to resume and halt execution at the debugger's discretion.
- Single stepping functionality.
- Breakpoint support.

Additional debugging features are offered only if the application being debugged allows for it. An example of this is stepping up (section 3.2.1), which requires the application to rely on a stack-based execution.

4.3 Connecting and initializing

In case the requirements mentioned in the previous section are met for a certain application, it may be considered for extension with debugging support. Thus, a *language specific debugger* may be created. This section deals with the steps that are necessary for a C-based application to establish a connection with the debugger and to initialize its debug adapter.

4.3.1 Getting started

To extend a component with debugging support, we need to write additional code to make communication with TIDE possible. Considering this from the application's point of view, we must:

- **Include the adapter library.**
- **Add function calls to create a new adapter and a new process.** This means putting `TA_connect(int port)` from where on debugging should commence. By use of this function, the connection id (*cid*) of the new debug adapter is received. Afterwards, a call to `TA_createProcess(int cid, char *name)` is performed to obtain a process id (*pid*).
- **Implement a stepfunction.** This function is to be invoked whenever an instruction is executed that calls for a step, i.e. in case execution is supposed to halt. It receives the current point of execution as an argument so the process' cpe may be updated for the sourceviewer to display. Furthermore, the function should contain an eventloop in which it waits for the user to undertake action. The step function must end as the user chooses to resume execution.
- **Register built-in functions for resuming and halting execution.** Registering built-in functions is achieved through use of the function `TA_registerFunction(int pid, AFun name, TA_Function func)`, that takes as arguments the pid, the properties of the desired built-in function and the name of the local function that implements it. These properties include the name and the arity.
- If desirable, **provide implementations for additional debugging features** and register the corresponding built-in functions. It is by means of these functions that custom actions are implemented.

For instance, consider the following code that is used by the ASF interpreter to establish a connection with the debugger.

```
int pid;

void Tide_connect()
{
  int cid;
  char *name = "ASF+SDF";

  cid = TA_connect(TIDE_PORT);
  pid = TA_createProcess(cid, name);

  TA_registerFunction(pid,ATmakeAFun("resume",0, ATfalse),eval_resume);
  TA_registerFunction(pid,ATmakeAFun("break", 0, ATfalse),eval_break);

...
}
```

Appendix A reveals a large portion of an application's TIDE extension.

4.3.2 Implementing features

If an application wishes to receive more debugging support than it gets by default, it may register additional functions in its debug adapter. By means of registering such an additional function, TIDE has a handle by which it can invoke the function (i.e. execute the action). The component being debugged must provide *implementations* for these function handles, as the debugger itself has no way of knowing how to perform the requested feature in the application's specific architecture. The functions can then be called by means of an action in an event rule.

As an example, consider the function called `list-sources` in the ASF interpreter's TIDE extension. It is used by the sourceviewer, presenting the user with a list of sourcefiles that are currently active. `list-sources` is a function that takes no arguments and invokes `eval_list_sources()`, which is the interpreters implementation for this function. Upon reception of an evaluation request of type `list-sources`, the interpreter executes the associated function and returns the list of source files.

4.4 Placement of debugging steps

As the application in question has been extended with debugging support, the ability to debug it is within reach. Hoewever, before we can make anything useful out of debugging it, we need to insert calls to the application's `step`-function into its architecture.

During execution, a call to the application's *step*-function results in the execution being stopped in case of single stepping. Obviously the placement of these steps should be done at strategic places, that is, at points at which the user would actually *want* the execution to halt. For instance, in the case of the ASF interpreter, calls to `TIDE_STEP` have been placed in the various stages of the rewriting process (see section 3.1.2) for the user to verify whether terms are reduced correctly. This is illustrated by the following code fragment, which is part of the interpreter's subroutine that deals with rewriting terms.

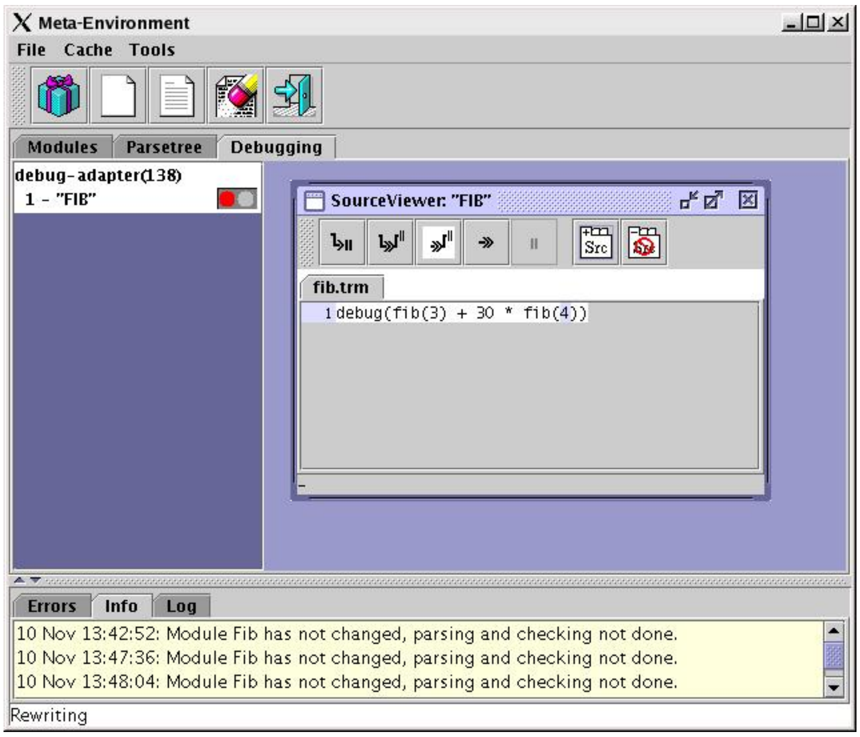
```
  TIDE_STEP(lhs, env, depth);
  lhstrm = rewriteInnermost(lhs, env, depth+1, NO_TRAVERSAL);

  if (!lhstrm) {
    return fail_env;
  }

  TIDE_STEP(rhs, env, depth);
  rhstrm = rewriteInnermost(rhs, env, depth+1, NO_TRAVERSAL);

  if (!rhstrm) {
    return fail_env;
  }

  currentRule = cur;
  TIDE_STEP((PT_Tree) cond, env, depth);
```

As soon as debugging steps have been inserted at relevant points in the application's architecture, we may proceed with debugging.

4.5 Summary

With the appropriate support in place, the construction of a language specific debugger is complete. The interpreter at hand can now connect to TIDE and send debug events that TIDE can comprehend. Upon execution of the interpreter, the support being offered is automatically available. The user may intervene the execution process, for instance by invoking a stackviewer or by placement of breakpoints (figure 4.1). The program finishes upon execution of the last instruction.

Chapter 5 presents the user with a case study, in which a specific debugger is instantiated for an example language.



```
debug(fib(3) + 30 * fib(4))
```

```
10 Nov 13:42:52: Module Fib has not changed, parsing and checking not done.
10 Nov 13:47:36: Module Fib has not changed, parsing and checking not done.
10 Nov 13:48:04: Module Fib has not changed, parsing and checking not done.
```

Figure 4.1: Language specific debugging.

Chapter 5

Obtaining a debugger from an ASF+SDF specification

5.1 Introduction

It has become clear in chapter 4 that extending an application with proper debugging support is relatively simple. In order to create a language specific debugging environment, it suffices to extend the language's specification with calls to functions in the TIDE adapter library and implement the desired debugging features. With such a debugger at our disposal, in addition to TIDE's ability to debug ASF+SDF, we have the means to take debugging a step further.

In this chapter, we will focus on two types of debugging:

1. Debugging a language specified in ASF+SDF.
2. Debugging a program that is executed by an interpreter that is also being debugged at the same time.

We start off by introducing the reader to the material used in this chapter, and then touch upon some issues that will rise in our attempt to accomplish the above. By means of an example language called *Pico*, we will then show in section 5.2 how to implement language-specific debugging support and perform an example debugging session. Afterwards, we will then attempt to debug this language on multiple levels. We conclude this chapter by discussing the results, e.g. the degree to which we have been successful, and which aspects require further effort.

5.1.1 Debugging programs written in ASF+SDF specified languages

An ASF+SDF language specification consists of the language's syntax in SDF and its semantics in ASF. In order to debug programs written in such a language, we require it to generate debugging events during its execution. The easiest way to accomplish this is by means of calls to built-in functions in the specification's rewrite rules.

Furthermore, we need a debugger that can handle these debugging events.

This debugger can be either *language-specific* or *generic*: this depends on whether we want to reuse the support for the debugging of other languages. More on this is explained later on.

5.1.2 Multilevel debugging

So far, we have only considered debugging one single process in this thesis. However, given the fact that TIDE has been designed to deal with multiple debug adapters and debug processes, one could also consider debugging multiple process at the same time.

We distinguish between three cases:

- The processes being debugged belong to different applications and are independent of each other (figure 5.1).
- The processes being debugged belong to one single application and are possibly dependent on each other (figure 5.2).
- The processes being debugged belong to different applications and are dependent on each other (figure 5.3).

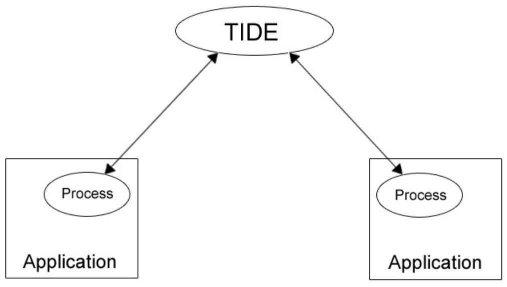


Figure 5.1: Debugging two processes that are completely independent of each other.

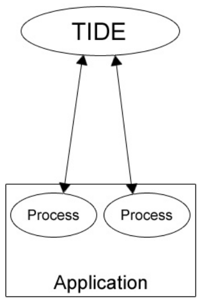


Figure 5.2: Debugging two processes in one application.

The first case is not very interesting in the context of this thesis, and we do not consider it necessary to prove that this actually works with TIDE.

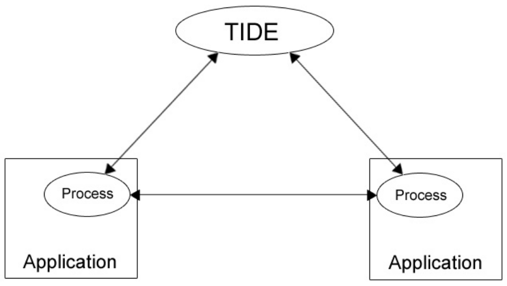


Figure 5.3: Debugging two processes that are dependent on each other.

The latter two cases on the other hand are situations that seem quite attractive for testing. Debugging two dependent components *simultaneously* may yield interesting results: Imagine running an application that consists of two parts, both of which are prone to contain mistakes - for example, an interpreter running an interpreter that runs a program. Debugging this process on two abstraction levels *at the same time* would greatly improve the effectiveness of a debugging session (figure 5.4).

In debugging two or more applications that are dependent on each other,

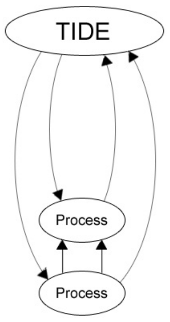


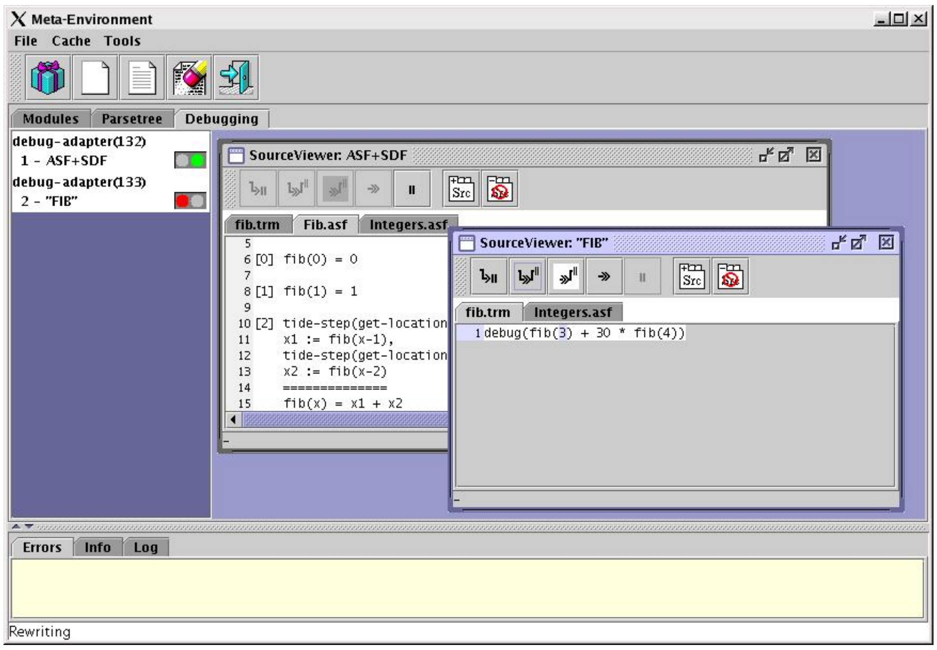
Figure 5.4: Debugging a single application on two abstraction levels.

the current point of execution shifts from one level to the other, following the exact path of execution in this program. The user is constantly in control of both debug processes and may utilize the debugging features that are offered on each of the layers (figure 5.5). Clearly, all of the components being debugged must be extended with TIDE support to accomplish this.

5.1.3 Issues

Several challenges await us when implementing a debugger for languages specified in ASF+SDF. We now describe the most notable issues.

- As was mentioned earlier, we have two options with regard to the type of debugger: it can either be language-specific or generic. In the running example we present later, we will choose the latter solution. This means



```
[0] fib(0) = 0

[1] fib(1) = 1

[2] tide-step(get-location
    x1 := fib(x-1),
    tide-step(get-location
    x2 := fib(x-2)
    ==============
    fib(x) = x1 + x2
```

```
debug(fib(3) + 30 * fib(4))
```

Figure 5.5: An example session in which debugging is done at both the specification and the language level.

that the generated events and the built-in functions will be generic in nature and, to a certain extent, capable of debugging any language specified in ASF+SDF. Whether or not we are making the right choice will become clear in the case study.

- The notion of multilevel debugging can not be realized in TIDE in its current form. This is because of the fact that although multiple processes are supported, processes belonging to *multiple applications* are not. As TIDE is asked to handle debugging events from more than one process and to relate these to the correct debug adapters, race conditions will inevitably turn up. Since our goal is to debug an ASF+SDF specification on both the language level and the level of ASF+SDF, we are essentially dealing with two applications at the same time and must therefore come up with a solution.

5.1.4 Introduction to Pico

To explain the concepts in this chapter, we will employ a running example by use of the toy language *Pico*. Pico is a simple programming language specified in ASF+SDF. Its implementation consists of multiple modules that specify the types, the value environments, syntax and so on (figure 5.6).

A Pico program starts by declaring its variables, which are subsequently put in a *variable environment*. The next phase consists of Pascal-like code that operates on these variables and adjusts the variable environment accordingly. Well-known constructions such as *if-then-else* and *while*-loops may be used to

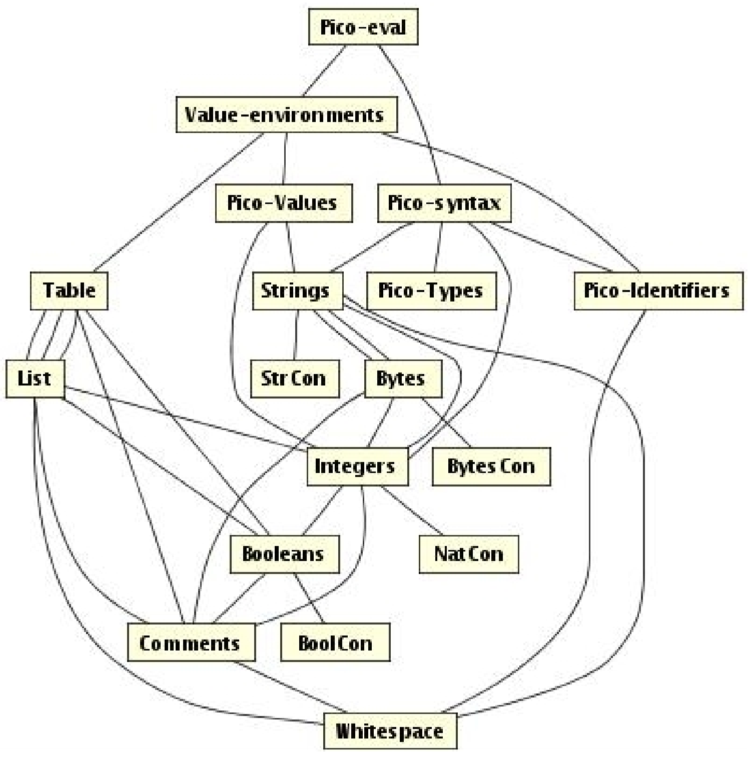


Figure 5.6: Pico's module hierarchy.

obtain the desired behavior.

Follows is an example Pico program that computes the factorial of 4:

```
begin
  declare
    input  : natural,
    output : natural,
    repnr  : natural,
    rep    : natural;

  input := 4;
  output := 1;

  while input - 1 do
    rep := output;
    repnr := input;
    while repnr - 1 do
      output := output + rep;
      repnr := repnr - 1
    od;
    input := input - 1
  od
  end
```

`Pico-eval` is a module that evaluates a Pico program and returns the resulting values of all variables. It contains the syntax definitions and semantics specifications necessary to reduce a Pico program to a variable environment.

As an example, consider the rewrite rule in `Pico-eval.asf` that is applied in case the condition in an if-then-else construction fails:

```
  eve(Exp, Venv) == 0
  ==================================================
  evst(if Exp then Series1 else Series2 fi, Venv) =
                                    evs(Series2, Venv)
```

As the condition `Exp` in the current environment `Venv` evaluates to 0, evaluation of the else-statements `Series2` ensues.

Having `Pico-eval` reduce the example Pico program mentioned earlier results in a variable environment that contains

```
[<input,1>, <repnr,2>, <output,24>, <rep,12>]
```

5.2 Implementation

5.2.1 Support being offered

In this case study, we have restricted the debugging support being offered to some of the most elementary features:

- Single stepping through Pico instructions.
- Placement of breakpoints in a Pico program.
- Inspection of event rules during a Pico debugging session.

This means that in comparison to the debugging support for ASF+SDF, several feature have been dropped. Unfortunately, this is inevitable due to genericity - the built-in library discussed in the next section has been designed to essentially support any ASF+SDF-specified language.

The dropped features include:

- Stepping over and stepping up is obviously not possible, as the execution of a Pico program is not stack-based.
- The stackviewer is not applicable for the same reason.
- No implementation is offered for viewing the actual values of Pico variables, as this is not done easily and because we do not consider it necessary for our demonstration purposes.

The latter feature would require the `tide-step`-function to take additional parameters, e.g. a value environment. Implementing a generic solution for this, however, is not trivial because languages may each store their variables in a different way. The alternative would be to make the built-in library language-specific, implying that it would merely be suitable for debugging Pico programs.

Nevertheless, it is our belief that the support being offered is sufficient to illustrate the basics of language specific debugging.

5.2.2 Built-in library

In order to effectively debug a Pico program, debugging steps are placed at strategic positions in Pico's semantics specification. To implement these debugging steps, and for both connecting and disconnecting, we have written a generic module called `Tide` that provides the necessary functions:

```
context-free syntax
  "tide-connect"(StrCon,Location) -> Boolean
                                       {built-in("tide-connect")}
  "tide-disconnect"(StrCon)       -> Boolean
                                       {built-in("tide-disconnect")}
  "tide-step"(Location)           -> Boolean
                                       {built-in("tide-step")}
```

These are to call newly created *built-in functions*, that relate to a built-in library called `builtin-tide.c`[1]. This is a generic library, meant for any language specification to employ through the use of the built-in functions. TIDE's adapter library (described in section 2.5.2) has been included in the built-in library for the language to communicate with the debugger.

5.2.3 Insertion of debugging events

We now need to extend Pico's specification with calls to the built-in `tide-step`-function mentioned in the previous section. In addition, we must have the Pico program connect to the debugger in an early stadium, and allow it to disconnect when the execution of the program has finished.

The latter is relatively simple in the Pico evaluator, as in this module there exists a global function `evp()` that encapsulates a Pico program. Its syntax definition reads

```
  "evp"(PROGRAM) -> VENV
```

whereas its algebraic specification looks like

```
  evp(begin Decls Series end) = evs(Series, evd(Decls))
```

This reads: upon encountering a Pico program, evaluate its declarations and then evaluate its statements.

Converting the rewrite rule so as to enable connection with the debugger results in

```
  tide-connect("PICO",get-location(Decls)) == true,
  tide-step(get-location(Decls)) == true,
  Venv := evs(Series, evd(Decls)),
  tide-disconnect("PICO") == true
  ================================================
  evp(begin Decls Series end) = Venv
```

[1]Built-in functions are constructed in the Meta-Environment's *asc-support* package, which is linked with the ASF interpreter.

which means that in order to rewrite a Pico program to a new value environment it will connect to TIDE, invoke a debugging step upon the location of the declarations, rewrite the program, and disconnect.

The placement of debugging steps into other relevant rewrite rules is done in a similar fashion. Taking the rule associated with an if-then-else construction (mentioned in the introduction) as an example, we put

```
tide-step(get-location(Exp)) == true,
eve(Exp, Venv) != 0
=====================================================
evst(if Exp then Series1 else Series2 fi, Venv) =
                                   evs(Series1, Venv)
```

The same is done with all rewrite rules that are associated with while-loops and statements. Appendix B provides the entire specification of the modified Pico evaluator.

5.2.4 Handling multiple applications

The issue concerning TIDE's incapability to handle multiple applications has been solved by building a *wrapper* around TIDE's connection manager. By means of this wrapper, it can not only distinguish between processes but also between applications. We have accomplished this by having TIDE establish *separate ToolBus connections* for each application that requires debugging.

In addition to the wrapper, an adjustment in the ToolBus architecture was necessary to keep messages from being delivered to the wrong debug adapter. We identified what can be considered a shortcoming in the way the ToolBus handles queued events: As the first in a series of events was posted, the ToolBus memorized the destination file descriptor and, consequently, sent all of the queued events to the very same connection. Obviously this posed a problem, as a series of events may well contain messages that are destined for different debug adapters. It has been solved by extending the ToolBus' `EventQueue` datastructure with a field that contains the corresponding file descriptor.

Having dealed with these issues, the debug events generated by one application can no longer interfere with the communications of the other (figure 5.3), since the applications only listen at their own sockets. Multiple applications can now be part of one single UNIX process, which is the basis for multilevel debugging.

Further details on the implementation of the built-in library can be found in appendix A.

5.3 Debugging a Pico program

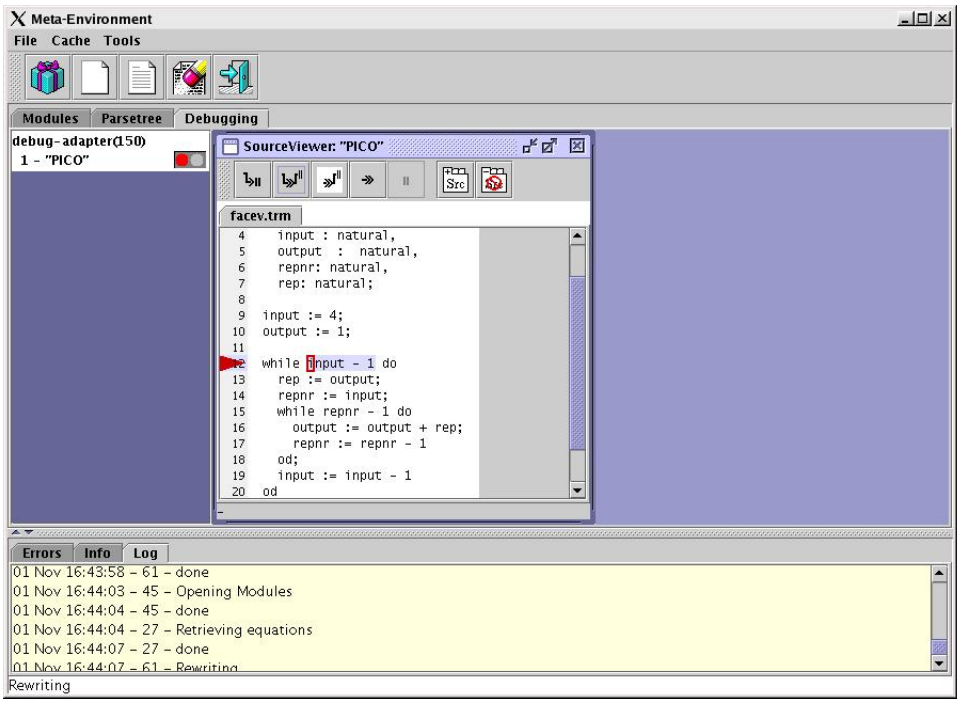
With the debugging steps and the underlying implementations in place, we are ready to start a Pico debugging session. Having loaded the modified Pico evaluation module, and taking the example program of section 5.1.4, we start reducing by choosing *Actions* and *Reduce*. (Choosing Reduce from the *Debug*-menu is left to the next section.)

As the first debugging step is encountered, reduction of the program halts.

Indeed this occurs in the initial phase, that is, upon evaluation of the declarations. If at this point we take a single step, the current point of execution shifts to the first statement. This is according to our expectations, as the specification of `Pico-eval` has been modified such that it invokes a debugging step at every type of statement.

We now step further until we come across the outer *while .. do .. od* construction. Pressing *step* once more reveals that it is now evaluating its condition. As the condition initially evaluates to true, the statements within the while construction are next.

Having verified whether the statements within the outer while-loop are all evaluated in accordance with our expectations, we may consider the placement of a breakpoint. By putting a breakpoint at the loop's condition for example, we can monitor the evaluation of the condition and afterwards press *run* to make the evaluator automatically step through the statements within the iteration (figure 5.7). Indeed, by putting a breakpoint in the condition and automati-



```
  input : natural,
  output  :  natural,
  repnr: natural,
  rep: natural;

input := 4;
output := 1;

while input - 1 do
  rep := output;
  repnr := input;
  while repnr - 1 do
    output := output + rep;
    repnr := repnr - 1
  od;
  input := input - 1
od
```

```
01 Nov 16:43:58 - 61 - done
01 Nov 16:44:03 - 45 - Opening Modules
01 Nov 16:44:04 - 45 - done
01 Nov 16:44:04 - 27 - Retrieving equations
01 Nov 16:44:07 - 27 - done
01 Nov 16:44:07 - 61 - Rewriting
Rewriting
```

Figure 5.7: An example Pico debugging session. A breakpoint has been placed at the condition of the outer while-loop.

cally executing each iteration for a total of four times, the reduction of the Pico program finishes as soon as the condition fails.

5.4 Debugging ASF+SDF and Pico simultaneously

Now that we have a language specific debugger at our disposal, in addition to TIDE's ability to debug ASF+SDF, we have the means to conduct a debugging session on multiple layers. To start debugging a Pico program and its ASF+SDF specification simultaneously, we open our example Pico program and choose *Reduce* from the *Debug*-menu.

The ASF interpreter is the first to connect to TIDE: in the GUI we can see on the left that an adapter and its debug process have connected. We can now invoke a sourceviewer for the ASF+SDF process and step until Pico's initial rewrite rule is encountered: the rule that reduces a Pico program to the evaluation of its declarations and its statements. Since this rule has a call to `tide-connect()` in its condition, we can see that after a few more steps a new adapter connects with a debug process named *Pico*. However, since we have not yet reached a debugging step for the Pico debugger, we cannot directly start debugging on the Pico level.

After some more steps, we have come across a debugging step in Pico's specification. From here on the ASF interpreter has no longer control over the execution (the light next to the *ASF+SDF*-process turns green) and we may open a sourceviewer for the *Pico*-process. As was the case in the previous section, the program's declarations are highlighted in the Pico sourceviewer. Upon taking another step, the execution's control is returned to the ASF interpreter.

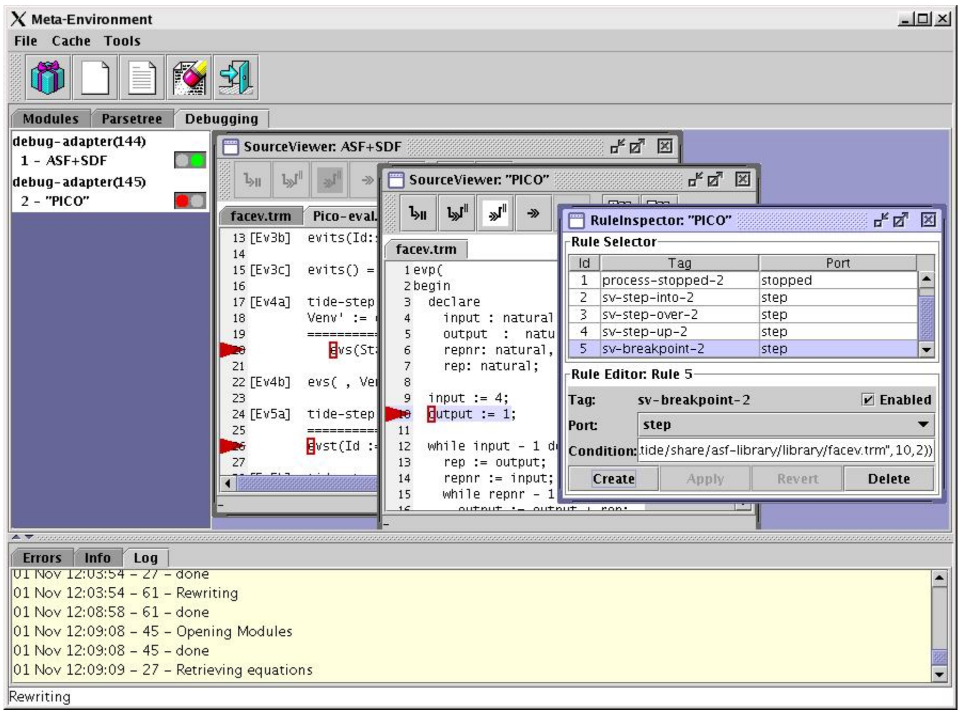
One can observe that the control continually switches between the two debug processes (figure 5.8). If at a certain point we decide that debugging one of the two processes is no longer necessary, it is simply a matter of pressing *run* in the sourceviewer of that process. From here on, that process' state is set to *running*, which implies that the other process can continue without being interfered by debugging steps on other levels - as long as no breakpoints are set on those levels. Clicking on *run* in the other process as well results in the normal execution of the input term.

It is important to realize that every tool at the user's disposal is process specific. Changing the event rules for a certain debug process does not influence the set of rules that is maintained in the other; the same goes for breakpoints.

5.5 Discussion

From our case study we can draw several conclusions.

- The processes being debugged are strictly separated by TIDE. The way one process is debugged does not influence the execution path of the other. Although one can think of situations in which the contrary would be helpful, for example setting breakpoints in one process that result in the halting of the other process' execution, we feel that a strict separation is wise because it guarantees a normal execution of the processes involved.
  Implementing the support for multilevel debugging turned out to be quite delicate. Race conditions indeed occured at first, but the wrapper we described in 5.2.2 solved this issue. Additionally, several adjustments in the ToolBus' architecture were necessary. Having done this, the processes



```
[Ev3b]  evits(Id:
        
[Ev3c]  evits() =
        
[Ev4a]  tide-step
        Venv' :=
        =========
           evs(St
        
[Ev4b]  evs( , Ve
        
[Ev5a]  tide-step
        =========
        evst(Id :
```

```
evp(
begin
  declare
    input : natural
    output  :  natu
    repnr: natural,
    rep: natural;

  input := 4;
  output := 1;

  while input - 1 d
    rep := output;
    repnr := input;
    while repnr - 1
```

Id	Tag	Port
1	process-stopped-2	stopped
2	sv-step-into-2	step
3	sv-step-over-2	step
4	sv-step-up-2	step
5	sv-breakpoint-2	step

Tag: sv-breakpoint-2 ☑ Enabled
Port: step
Condition: tide/share/asf-library/library/facev.trm",10,2))

```
01 Nov 12:03:54 – 27 – done
01 Nov 12:03:54 – 61 – Rewriting
01 Nov 12:08:58 – 61 – done
01 Nov 12:09:08 – 45 – Opening Modules
01 Nov 12:09:08 – 45 – done
01 Nov 12:09:09 – 27 – Retrieving equations
Rewriting
```

Figure 5.8: Simultaneously debugging a Pico program and its ASF+SDF specification.

involved in the session we conducted were strictly separated by means of their own private TIDE connections, resulting in our debugging session showing the desired behavior.

- We can state that, judging by the amount of code that was needed, it was relatively easy to extend Pico with debugging support. As we required only the basic support in this case study, the built-in library we created remained fairly simple. Insertion of steps into Pico's specification was a relatively straightforward task.
  It is important to realize that things are not always this simple. Building debugging support into an ASF+SDF specified language is easy because a generic module may be used to invoke the TIDE functions, by means of which a language specification is easily transformed even by users with limited ASF+SDF knowledge. However, one can also think of other types of applications, e.g. applications in which the placement of debugging steps is not intuitive. A great deal of knowledge of an application's architecture is required if calls to the step-function are to be placed at suitable positions.

- Concerning the built-in library, the question is how to provide a generic interface, e.g. for value environments. One solution is to have a value environment that every language specification should use. It is obvious that by remaining generic, the amount of domain-specific support diminishes.

On the other hand, implementing more advanced features tends to be complicated: a thorough understanding of Pico's specification is then required. If we want to implement the feature that displays the actual value of a variable, we need to know all about the underlying datastructures in which the variables are stored. The debugging support for the ASF interpreter does contain this feature, and indeed its implementation comprises over 70 lines of code. Still, this is negligible compared to the effort it takes to implement a language specific debugger from scratch.

Chapter 6

Conclusions

In this final chapter we shall reflect upon our work in this project. It will be summarized to what extent we have managed to tackle the problems stated in the introduction, and with regard to which aspects we have been less successful. We conclude by pointing out which areas require further research.

6.1 Contributions

- **Documentation**
  In this thesis, we have documented the TIDE framework in detail. In chapter 2, we touched upon most of the framework's features and provided an understanding of the underlying implementation, whereas the tutorial in chapter 3 illustrated the use of TIDE's features in the context of ASF+SDF debugging. We also added more built-in functions and implemented a button for stepping up.

- **Generating language-specific debuggers**
  We then broadened our perspective and moved onto extending the framework's usefulness. We both explored and implemented debugging support for a language specified in ASF+SDF. By means of this case study, we illustrated that any ASF+SDF specified language is subject to debugging by extending it with the appropriate support. We observed that the effort needed to implement the debugging support was dependent on the desired features, and concluded that compared to building a language-specific debugger from scratch, the effort was negligible.

- **Debugging at multiple levels**
  Finally, we took it a step further by discussing the case in which multiple processes within one application are debugged at the same time. After a few issues were solved, some of which required adjustments in TIDE's own implementation, TIDE was able to handle multiple applications. The notion of multilevel debugging in TIDE was introduced and we have shown that TIDE is indeed capable of simultaneously debugging a language at two abstraction layers.

6.2 Future work

As it stands, the user does not receive full support in debugging ASF+SDF specifications when using the `Tide`-module we presented. It was pointed out in 5.2 that looking up variables, for example, is not easily done in a generic manner because it depends on the program's architecture how the variables are stored. Whether this can be accomplished may be an interesting research question for the future.

Other points of interest include applying TIDE in different environments. Although we have only used the framework for debugging ASF+SDF and languages specified therein, it should also work for debugging programs that are written in different languages, e.g. C++ and Java. New debug adapters need to be written for these languages to render them eligible for debugging, and the adapter library discussed in 2.5.2 needs to be ported to these other languages.

Another interesting application area for TIDE would be a distributed environment. This should obviously work out since we have shown that the framework is capable of handling multiple processes, both dependent and independent.

Bibliography

[1] P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* PhD thesis, Faculty of Science, University of Amsterdam, 2000.

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

[3] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping, An Algebraic Specification Approach.* World Scientific Publishing Co. Pte. Ltd., 1996.

[4] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The Syntax Definition Formalism SDF, Reference Manual.* University of Amsterdam, Programming Research Group, 1992.

[5] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.

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

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

[8] M.G.J. van den Brand et al. The ASF+SDF meta-environment: A component-based language development environment. In *Computational Complexity*, pages 365–370, 2001.

[9] Domain specific language debugger framework (DDF) project, http://www.cis.uab.edu/wuh/ddf/.

Appendix A

TIDE built-in

In this appendix we provide the relevant parts of the source code of `builtin-tide.c`, the library that implements the TIDE built-ins discussed in section 5.2.

```
int pid;

static PT_Tree tide_connect(PT_Tree name, PT_Tree init_location)
{
  int cid;
  char* str = PT_yieldTree(name);

  cid = TA_connect(8999);
  pid = TA_createProcess(cid, str);

  TA_registerFunction(pid, ATmakeAFun("resume", 0, ATfalse),
                                                   eval_resume);
  TA_registerFunction(pid, ATmakeAFun("break",  0, ATfalse),
                                                   eval_break);

  TA_atCPE(pid, tree_to_ta_location(init_location), 0);

  return (PT_Tree)CO_makeBooleanConstant(CO_makeBoolConTrue());
}

static PT_Tree tide_disconnect()
{
  if (TA_isConnected()) {
    TA_disconnect(ATtrue, pid);
  }

  return (PT_Tree)CO_makeBooleanConstant(CO_makeBoolConTrue());
}

static PT_Tree tide_step(PT_Tree tLoc)
{
  if (TA_isConnected()) {
    int old_state = TA_getProcessState(pid);

    old_state = TA_getProcessState(pid);

    TA_atCPE(pid, tree_to_ta_location(tLoc), 0);
    TA_activateRules(pid, TA_makePortStep());

    if (TA_getProcessState(pid) == STATE_STOPPED) {
      if (old_state != STATE_STOPPED) {
        TA_activateRules(pid, TA_makePortStopped());
      }

      while (TA_getProcessState(pid) == STATE_STOPPED) {
        TA_handleAny(pid);
      }
      if (TA_isConnected()) {
        TA_activateRules(pid, TA_makePortStarted());
      }
    }
  }
  return (PT_Tree)CO_makeBooleanConstant(CO_makeBoolConTrue());
}

static TA_Expr eval_resume(int pid, AFun fun, TA_ExprList args)
{
  TA_setProcessState(pid, STATE_RUNNING);

  return ATparse("true");
}

static TA_Expr eval_break(int pid, AFun fun, TA_ExprList args)
{
  TA_setProcessState(pid, STATE_STOPPED);

  return ATparse("true");
}

static TA_Location tree_to_ta_location(PT_Tree tree)
{
  PERR_Location pLoc;
  ERR_Location loc;

  pLoc = PERR_LocationFromTerm(PT_TreeToTerm(tree));
  loc = PERR_lowerLocation(pLoc);

  return TA_LocationFromTerm(ERR_LocationToTerm(loc));
}
```

Appendix B

Pico specification with TIDE support

The entire algebraic specification of `Pico-eval` as described in section 5.1.4 can be found below. It has been enriched with calls to TIDE functions to make Pico specific debugging possible.

```
equations
[Ev1]   tide-connect("PICO",get-location(Decls)) == true,
        tide-step(get-location(Decls)) == true,
        Venv := evs(Series, evd(Decls)),
        tide-disconnect("PICO") == true
        ==================================================
        evp(begin Decls Series end) = Venv

[Ev2]   evd(declare Id-type*;) = evits(Id-type*)

[Ev3a]  evits(Id:natural, Id-type*) = store(evits(Id-type*), Id, 0)

[Ev3b]  evits(Id:string, Id-type*) = store(evits(Id-type*), Id, "")

[Ev3c]  evits() = []

[Ev4a]  tide-step(get-location(Stat)) == true,
        Venv' := evst(Stat, Venv), Venv'' := evs(Stat*, Venv')
        ======================================================
           evs(Stat ; Stat*, Venv) =  Venv''

[Ev4b]  evs( , Venv) = Venv

[Ev5a]  tide-step(get-location(Exp)) == true
        =======================================================
        evst(Id := Exp, Venv) = store(Venv, Id, eve(Exp, Venv))

[Ev5b]  tide-step(get-location(Exp)) == true,
        eve(Exp, Venv) != 0
        =====================================================
        evst(if Exp then Series1 else Series2 fi, Venv) =
                                           evs(Series1, Venv)

[Ev5c]  tide-step(get-location(Exp)) == true,
        eve(Exp, Venv) == 0
        =====================================================
        evst(if Exp then Series1 else Series2 fi, Venv) =
                                           evs(Series2, Venv)

[Ev5d]  tide-step(get-location(Exp)) == true,
        eve(Exp, Venv) == 0
        ==========================================
        evst(while Exp do Series od, Venv) =  Venv

[Ev5e]  tide-step(get-location(Exp)) == true,
        eve(Exp, Venv) != 0,  Venv' := evs(Series, Venv)
        =====================================================
        evst(while Exp do Series od, Venv) =
                          evst(while Exp do Series od, Venv')

[Ev6a]  eve(Id, Venv) = lookup(Venv, Id)

...

[Ev6d]  tide-step(get-location(Exp1)) == true,
        Nat1 := eve(Exp1, Venv),
        Nat2 := eve(Exp2, Venv)
        ======================================
        eve(Exp1 + Exp2, Venv) = Nat1 + Nat2

[Ev6e]  tide-step(get-location(Exp1)) == true,
        Nat1 := eve(Exp1, Venv),
        Nat2 := eve(Exp2, Venv)
        ======================================
        eve(Exp1 - Exp2, Venv) =  Nat1 -/ Nat2

[Ev6f]  tide-step(get-location(Exp1)) == true,
        Str1 := eve(Exp1, Venv),
        Str2 := eve(Exp2, Venv)
        ======================================
        eve(Exp1 || Exp2, Venv) = Str1 || Str2

[default-Ev6]  tide-step(get-location(Exp)) == true
               ==================================
               eve(Exp,Venv) = nil-value
```