A Framework for Debugging
Olivier, P.A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
4

Debugging Heterogeneous Distributed Applications

Our experience gained from the work described in the previous chapter convinced us that it is possible to design a generic language independent debugging framework based on the TOOLBUS coordination architecture.

In this chapter, we present a debugging framework for debugging heterogeneous distributed systems. In such systems, a variety of languages can be used for the implementation of individual components, so our debugging framework must be able to deal with these languages. Instead of reinventing the wheel in this area, we try to reuse existing debugging support for these languages as much as possible.

The majority of the existing debuggers for these languages work by abstracting the behavior of the program being debugged into events, and visualizing these events. We utilize these sequential debuggers to generate language-independent debugging events related to the sequential execution of the components in the distributed system. The underlying coordination architecture (in our case the TOOLBUS) is used to generate debugging events dealing with the interaction between components. These sequential and process interaction related debugging events are then processed by a separate distributed system that implements the high-level language-independent debugging functionality.

4.1 Introduction

Debugging is the process of locating and fixing errors (bugs) in software systems. A debugger is a software tool that can help understand a system being debugged by visualizing different aspects of its execution.

Although multilingual sequential debuggers have been around for quite some time [Bea83, SP91], many distributed debugging tools are based on support from a single experimental operating system or language environment [For89].

In the area of heterogeneous distributed systems, things get even worse. Debugging support is often limited to tracing the communication between components, and the debugging of the
components themselves is left to traditional tools for debugging sequential programs. Although most modern language implementations have some kind of debugging support built in, they all have their own interface and command set. Having to work with a number of debuggers at the same time can make it very difficult, if not impossible, to debug such heterogeneous systems.

This calls for an interface between distributed and sequential debuggers, in order to combine the two fields [CBM90] and reuse existing implementations. We have designed such an interface, and subsequently built a powerful, multilingual debugger for distributed systems.

Most debuggers work by gathering primitive events, filtering or clustering them, and presenting the results to the user. Details on how to gather events, which filtering or clustering algorithms to use, and when and how the results are presented differ in each case. Unfortunately, when it comes to event gathering, every system seems to reinvent the wheel. Solutions ranging from hardware assisted compiler instrumented code [AY91], to manually inserted event generation calls [BW83] can be found in the literature. In this chapter we show that by reusing these low level event gathering implementations, we can leverage the existing plethora of debugger implementations into a single framework for debugging heterogeneous distributed applications.

We have combined the power of the TOOLBUS coordination architecture with existing low level debugging interfaces for sequential programming languages. The resulting framework consists of language-dependent debugging interfaces, based on the native debugging support found in most existing language environments, coupled with language-independent debugging components. By adding event reporting for the coordination architecture we use, the resulting set of primitive events is rich enough to build a solid distributed debugging environment.

To show the feasibility of our approach, we have constructed a debugger for distributed applications. The debugger offers a uniform graphical user interface for inspecting the communication behavior of the system being debugged, and for tracing the source level execution of the components of the system. Our debugger is both extensible in the set of languages it can handle, as well as in the debugging functionality it has to offer, because of the debugging framework it is based on. The design and implementation of this debugger is discussed in Section 4.5.

In Section 4.2 we introduce our basic framework. In Section 4.3 we present the notion of event rules that play a central role in our framework, and in Section 4.4 we show how these event rules can be used to implement some of the functionality that is typically found in traditional sequential debuggers.

4.2 A TOOLBUS Framework for Debugging Distributed Applications

The majority of distributed systems consist of a number of sequential components running in parallel. We present a debugging framework for distributed systems which uses the events generated by native debuggers for these sequential components. These native debugger events
are translated into language-independent debugging events. The events generated by the TOOLBUS monitor protocol (see Section 2.7) are used to generate events about the interaction between processes. The combination of language-independent debugging events and process interaction events are used to synthesize high-level debugging facilities not found in the underlying native debuggers. Examples of these facilities are:

- A uniform graphical user interface for debugging all tools, independent of the language they are written in.
- Tracing and animating execution at the source code level, independent of the language the tool is written in.
- Full conditional breakpoint and watchpoint support, based on the expressions discussed in Section 4.3.3, even if the underlying native debugger does not support conditional breakpoints.
- Breakpoints and watchpoints that can be set on any type of event, not just on specific locations in the source code. For example, watch the value of a variable whenever a certain message is sent.

We base our framework on the assumption that the distributed system being debugged is also based on the TOOLBUS coordination architecture. However, our techniques can easily be generalized to other distributed architectures as the ones discussed in Section 2.8.

Let us first recall the basic TOOLBUS architecture from Chapter 2. Figure 4.1 shows how in the standard TOOLBUS architecture a viewer tool can be connected to debug TOOLBUS scripts.

The first step towards this new architecture is to introduce a second TOOLBUS based distributed system which actually implements the debugger as shown in Figure 4.2. The TOOLBUS used in the system being debugged will be called the application bus, and the one used in the debugging system will be called the debugging bus.
In Section 2.7 we explained that the viewer tool visualized debugging events to provide the user with visual feedback on the execution of the application bus. In our new architecture, the TOOLBUS viewer tool is replaced by a tool that acts as a gateway between the application bus and the debugging bus. This gateway is used to forward the debugging events received from the application bus directly to the debugging bus. The actual debugging functionality is implemented in the debugging bus using a number of cooperating tools, for instance, a source code browser and a process viewer. This architecture makes it possible to break up the complex implementation of the debugger in a number of more manageable components.

But we can take this architecture one step further. To do this, we must realize that most of the languages used to implement the components that are connected to the application bus have some kind of native debugger support. The debugging interfaces of languages like Java [GJS96], Python [WvRA96], Tcl/Tk [Ous94], and C (using for instance gdb) all offer the possibility to implement a sophisticated debugger on top of a low level debug interface.

Figure 4.3 shows our final architecture, in which the individual components of the application bus are also connected to the debugging bus using a debug-adapter. This debug-adapter uses the low level native debug interface to generate debug events related to the actual execu-
tion of these components.

It is important to note that components can contain multiple lightweight processes (sometimes called *threads*), for instance in the case of a Java component. As we will show later on, each debug-adaptor is responsible for processing the low level debug events of all debuggable processes in the component it is part of.

![Diagram](image)

**Figure 4.3: Full debug event gathering**

### 4.3 Event Rules

Most debuggers gather primitive events and use these to inform the user of what is going on in a program. Our framework reuses these native debugging events to do the actual event gathering. The primary task of the debug adapters introduced in Section 4.2 is to unify and filter these events in order to make our framework language-independent.

We do this unification and filtering of low level debugging events by using *event rules*. An event rule consists of an *event port*, an *event condition*, and a list of *event actions* (Figure 4.4). The event port indicates at which points during the execution of a program an event rule is *activated*. When an event rule is activated, its event condition is evaluated to see if
the event rule should be *triggered*. Triggering the event rule consists of executing its event actions. The event actions can generate unified debugging events handled by the application bus, or they can influence the execution of the component that triggered them, for instance by halting execution (*breakpoints*), or changing the contents of a variable.

Event rules are process specific. It is the task of the debug-adapter to maintain a set of event rules for each process in the component that is being debugged. Note that most components contain only one process. Only multithreaded/multiprocess applications (for instance Java applications or the TOOLBUS) can contain multiple processes.

### 4.3.1 Required Support

In order to be useful in our framework, the components being debugged must support some basic features. Surprisingly, the set of features required to be able to implement a minimal debug adapter is quite small. Only four features are absolutely necessary:

- The component can consist of multiple processes, but each process must support the notion of a *current point of execution (cpe)* in order to be able to debug that process. It must be possible to relate this *cpe* to a location in the source code of the component.

- It must be possible to start and stop the execution of each process in the component.

- It must be possible to single step through the execution of a process. The granularity of such a “single step” must be intuitive at the source code level. Single stepping one assembly instruction is not all that useful when working at the source code level.

- There must be support for breakpoints.

Other features are not absolutely necessary, but are only required to enable certain debugger functionality. For instance, the debugger can only offer the “step over” functionality...
discussed in the next section when the component being debugged has a stack-based execution model, and the debug-adapter can access the current stack depth. Another example is the possibility to inspect the value of variables. This will only work when the debug-adapter of the component being debugged has some means of retrieving the value of a variable.

Stated differently, our framework offers a form of *graceful degradation*. The fewer the features for the debug-adapter to work with, the fewer debug functionality is offered to the user of the debugger. But the whole mechanism only breaks down when the debug-adapter does not implement the *cpe* or execution control functionality discussed above.

### 4.3.2 Event Ports

The *event port* is used to bind a set of native debugging events to an event rule. Table 4.1 shows a list of event ports we are currently considering. This list can be extended or restricted to suit the needs of a particular implementation.\(^1\)

\(^1\)The term *event port* is based on the standard Prolog 4-port tracer [CM87]. The *entry* and *exit* ports are directly based on their Prolog counterparts.
4 / DEBUGGING HETEROGENEOUS DISTRIBUTED APPLICATIONS

<table>
<thead>
<tr>
<th>Port</th>
<th>Activated when</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>a function/predicate is called.</td>
</tr>
<tr>
<td>exit</td>
<td>exit from function/predicate.</td>
</tr>
<tr>
<td>stopped</td>
<td>execution stops.</td>
</tr>
<tr>
<td>started</td>
<td>execution continues.</td>
</tr>
<tr>
<td>location(loc)</td>
<td>the specific location in the source code indicated by loc is reached.</td>
</tr>
<tr>
<td>var-access(var)</td>
<td>the contents of a variable is accessed.</td>
</tr>
<tr>
<td>var-change(var)</td>
<td>the value of a variable changes.</td>
</tr>
<tr>
<td>exception</td>
<td>an exception or error occurs.</td>
</tr>
<tr>
<td>step</td>
<td>a statement is executed.</td>
</tr>
<tr>
<td>send</td>
<td>a message is sent.</td>
</tr>
<tr>
<td>receive</td>
<td>a message is received.</td>
</tr>
</tbody>
</table>

Table 4.1: Event ports

4.3.3 Event Conditions

Event conditions are used to filter uninteresting events locally. This prevents the debugging bus from having to handle excessive amounts of events, and protects the user from an excessive amount of information that he or she did not ask for. For instance, by implementing conditional breakpoints even when the underlying native debugger only supports unconditional breakpoints.

The event condition is an expression that is evaluated when the event rule is activated. The event rule is only triggered when this expression evaluates to true. The debug adapter decides which terms are considered to be equivalent to true and which are not.

In our framework, there are two places where we use expressions:

- In the condition of event rules.
- In the arguments of actions of event rules.

Unfortunately, the syntax and semantics of expressions are different in every programming language. To make things even worse, not every debugger supports the evaluation of these native expressions.

Our framework offers the user mixed expressions in order to overcome the inadequacy of some debuggers in this area. Our expression language consists of prefix functions that are evaluated whenever their value is needed.

These expressions are called mixed, because there is a special function eval, which takes a single string argument. This string is passed to the native debugging interface for evaluation. The native debugger determines how this string is interpreted, but most debuggers will allow the user to use the syntax of the language being debugged. Although the result type of most
### Event Rules / 4.3

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Debug functionality that depends on this function (see Section 4.5)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>state</strong></td>
<td>Retrieves the execution state of the current process.</td>
<td>Process status viewing</td>
</tr>
<tr>
<td><strong>cpe</strong></td>
<td>Retrieve the current point of execution.</td>
<td>Highlighting cpe in the source code</td>
</tr>
<tr>
<td><strong>var</strong></td>
<td>Retrieves the value of a variable.</td>
<td>Conditional breakpoints and variable viewing</td>
</tr>
<tr>
<td><strong>msg</strong></td>
<td>Retrieve the last message sent or received.</td>
<td>Communication viewing</td>
</tr>
<tr>
<td><strong>stack-depth</strong></td>
<td>Retrieve the current depth of the stack.</td>
<td>step-over</td>
</tr>
<tr>
<td><strong>start-depth</strong></td>
<td>Retrieve the depth of the stack at the time the last resume function was executed.</td>
<td>step-over</td>
</tr>
<tr>
<td><strong>true</strong></td>
<td>Always returns true</td>
<td>Basic support</td>
</tr>
<tr>
<td><strong>false</strong></td>
<td>Always returns false</td>
<td>Basic support</td>
</tr>
<tr>
<td><strong>equal(t1,t2)</strong></td>
<td>Returns true when t1 and t2 are equal, false otherwise.</td>
<td>Basic support</td>
</tr>
<tr>
<td><strong>higher-equal(t1,t2)</strong></td>
<td>Returns true when t1 is a number that is higher or equal than t2.</td>
<td>Basic support</td>
</tr>
</tbody>
</table>

Table 4.2: Functions for use in event conditions and event actions

functions is predefined, the result of `eval` is left to the debug adapter. For instance, if it can determine that the result is of type `integer`, an integer is returned. It can always return a string representing the result when there is no other sensible TOOLBUS term equivalent. Fortunately, most TOOLBUS adapters already provide a generic way to translate internal data objects into TOOLBUS terms.

A number of other functions can be used in expressions, including functions to retrieve the current state of execution of the process being debugged or to retrieve the value of a variable. Mathematical and logical operators are also included to perform simple calculations and comparisons. Tables 4.2 and 4.3 give an overview of the functions that are required to implement all the debug functionality discussed in this chapter. This is not a closed set of functions, but it will expand as more and more functionality is added to an implementation. Not all debug-adapters have to support all of these functions. Some features are just not available when the responsible debug adapter does not support the correct functions.

The event port and event condition can be used by the debug adapter to configure the un-
4 / DEBUGGING HETEROGENEOUS DISTRIBUTED APPLICATIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Debug functionality that depends on this function (see Section 4.5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>Stop the execution of the current process.</td>
<td>Explicit execution control, breakpoints</td>
</tr>
<tr>
<td>resume</td>
<td>Resume execution of the current process.</td>
<td>Explicit execution control</td>
</tr>
</tbody>
</table>

Table 4.3: Functions for use only in event actions

derlying native debugger. In this way, events can be filtered in an early stage, using hardware or operating system support when possible. A typical example of this kind of optimization is the use of low level conditional breakpoints to avoid activation of rules when the condition would fail anyway.

### 4.3.4 Event Actions

When an event rule is triggered, its event actions are evaluated. When the evaluation results in a value other than true, a generic debugging event is generated that can be processed by the debugging bus.

The evaluation of an event action can also cause a change in the (execution) state of the process that triggered the event rule. It can for instance halt the execution. In this case, the event rule that triggered the action is functionally equivalent to a breakpoint in traditional debugging terminology.

All expressions described in Section 4.3.3 can also be used in event actions. In event actions some special actions can also be used that change the (execution) state of a process, as shown in Table 4.3.

### 4.4 Example Event Rules

In this section, we will show how some well known debugger features can be implemented using event rules. Our debugger implementation (Section 4.5) implements these features using the event rules described in this section.

#### 4.4.1 Single Stepping Execution

One of the most important features of any debugger is the ability to single step through the execution of a segment of code.

Ensuring execution stops after executing a single step is done using the event rule:


After each step, the current location of the program counter is usually visualized by printing or highlighting the appropriate piece of source code. Which event rules are used to perform this highlighting is shown in Section 4.4.3.

4.4.2 Stepping over Function Calls

A variation of the step command discussed above, is the step over command. When the user issues this command, a single instruction is executed, just as with the step command. If this instruction happens to be a function call, the entire call is executed before control is returned to the user.

To implement this, we use the same event rule as with the step command, but instead of leaving out a condition, we introduce a condition that uses the depth of the stack to determine whether to halt execution or not:

\[\text{Port: step}\]
\[\text{Condition: higher-equal(start-depth, stack-depth)}\]
\[\text{Action: break}\]

The built-in function start-depth returns the depth of the stack when execution was last started, i.e., at the time the last resume action was executed. The function stack-depth returns the current depth of the stack. Whenever the current depth of the stack equals or exceeds the depth at which execution was started, we know that we are no longer executing a function call and so we can halt execution.

Note that some debug-adapters optimize this event rule by translating it into an explicit 'step over' call in the native debugger. The native debugger in turn can implement this by placing a breakpoint on the next line in the current function. This way native breakpoint support can be used to execute called functions at full speed instead of activating the event rule after every instruction of that function.

4.4.3 Instruction Highlighting

Most debuggers can visualize the execution of a program by highlighting instructions as they are executed. This instruction highlighting can be found in two flavors:

- Highlight on halt.
- Execution animation.
Both flavors can be implemented using event rules that utilize the function \texttt{cpe} that returns the current point of execution of a process using coordinates related to the original source code.

To highlight the current instruction whenever the execution is halted, we use the event rule:

\begin{verbatim}
Port:        stopped
Condition:   true
Action:      cpe
\end{verbatim}

By changing the port from \texttt{stopped} to \texttt{step}, the event rule is triggered with the execution of every instruction. This enables the debugger to animate the execution of the process:

\begin{verbatim}
Port:        step
Condition:   true
Action:      cpe
\end{verbatim}

### 4.4.4 Watching Variables

To watch the current value of a set of variables whenever the execution of a process stops, we use the function \texttt{var(name)} that retrieves the current value of a variable. By creating this event rule:

\begin{verbatim}
Port:        stopped
Condition:   true
Action:      var(variable-name)
\end{verbatim}

The value of the variable is sent to the debugging bus so it can be displayed whenever execution stops.

We could also change the port from \texttt{stopped} to \texttt{var-change} to animate the value of the variable during program execution:

\begin{verbatim}
Port:        var-change(variable-name)
Condition:   true
Action:      var(variable-name)
\end{verbatim}

### 4.4.5 Breakpoints

Almost every debugger enables the user to set a breakpoint at a specific location in the source code. When the execution of the program reaches this location, the execution is halted. In our framework, this kind of breakpoint is called 'simple breakpoint'. A simple breakpoint is implemented by creating the event rule:
Where \( loc \) is the desired breakpoint location in the source code given in source coordinates (file name and line number). By simply adding conditions to the event rule, we can implement conditional breakpoints as well.

### 4.5 Implementation

So far, we introduced a framework for language independent distributed debugging. In this section, we will present an actual debugger we have implemented based on this framework. This implementation shows how the framework can be used as a basis for a generic debugging system that can support a variety of languages and debugging tools. We will give an overview of the features present in this implementation, and give an impression of the user interface of the various debugging tools.

We have named our debugger TIDE: the Toolbus Integrated Debugging Environment. It offers impressive advantages over ‘standard’ debugging environments:

- TIDE can debug distributed systems based on the TOOLBUS.
- TIDE operates in any operating environment that is supported by the TOOLBUS. This includes most modern unix operating systems.
- When the distributed system to be debugged is based on the TOOLBUS, TIDE can display a graphical representation of the processes and communication in the system.
- Even in non-TOOLBUS based distributed systems or with stand-alone programs, TIDE can trace sequential components at the source level, as long as the component is implemented using one of the language environments supported by TIDE.
- Support for a new source language only requires a new debug adapter to interface with the TIDE system. Because these adapters are built on top of the native debugging interface of the language implementation in question, it is typically very small (300-1000 lines of source code). We have implemented debugging adapters for the following languages:
  - C
  - Java
  - Tcl/Tk
  - ASF+SDF\(^2\)

\(^2\)In Chapter 8 we present TIDE debugging support for ASF+SDF.
Because of the loose coupling between the debug adapters and the debugging tools, it is also relatively easy to add new debugging tools.

The debugger TOOLBUS is the center of the TIDE system. Connected to it are two kinds of tools: debugging adapters and debugging tools. The adapters generate debugging events related to the execution of the program being debugged. The debugging tools control the configuration of the debugging adapters, and visualize the debugging events.

### 4.5.1 User interface

![Image of TIDE user interface](image)

Figure 4.6: Debugging a distributed system using TIDE

Figure 4.6 shows a screenshot of a debugging session. The system being debugged is the *calculator example* presented in Section 2.5. On the left side a list is shown of all debug adapters that are connected to the TIDE system. For each debug adapter, a tree of processes managed by the adapter is shown. Most debug adapters manage only a single process. Exceptions are debug adapters for multithreaded or parallel languages like Java and the TOOLBUS itself. In these cases, each thread or process is handled separately.
Next to this list of debug adapters, the process viewer is visible. This debug tool shows an overview of the TOOLBUS system being debugged, complete with processes, tools, and communication between them using arrows.

Below the process viewer, an instance of the source viewer shows the source code of one of the tools being debugged, in this case a tool written in Tcl/Tk. The current point of execution in this tool is highlighted.

To the right of the process viewer, another instance of the source viewer is used to display the source code of one of the TOOLBUS processes. Again the current point of execution is highlighted. In this window, the value of the variable N is displayed by right-clicking on it. When right-clicking on a variable, the source viewer retrieves the current value of that variable from the debug adapter, and displays it using a small popup window. This approach has the advantage that the user of the debugger does not have to make a mental ‘context switch’ to another window to view the value of a variable: the information is readily available at the location where it is needed.

Figure 4.7 shows the TIDE user interface when debugging a stand-alone program. In this case the program is written in C. A source viewer window shows the source code of the program being debugged. The current point of execution is again highlighted, together with
a breakpoint that has been set. Breakpoints are set by double-clicking on the target source line.

To offer the user of TIDE maximum flexibility, we have made it possible to directly edit the event rules present in the debug adapter. In order to do this, the rule editor is used, which is shown above the source viewer window in Figure 4.7. In this case the rule that corresponds to the breakpoint highlighted in the source viewer is selected. This direct control over the content of a rule can, for instance, be used to turn an unconditional breakpoint into a conditional one.

### 4.6 Related work

Reusability of debugger implementations has (at least) two dimensions: portability across multiple platforms and support for multiple languages. Most work on portable debuggers focussed on one of these two dimensions.

Work along the other dimension, achieving some degree of machine independence, mostly aims at supporting only one implementation language across multiple platforms. Probably the best example of this is the work done on ldb [RH92] and more recently cdb [HR96, Han99b], two largely machine independent debuggers for gcc [FH95] a retargetable C compiler.

Cdb indeed nearly achieves complete independence from architectures and operating systems, by loading a small amount of code with the target C program, and by having the compiler emit a machine-independent symbol table.

In some sense, TIDE and cdb could complement each others strengths, as TIDE relies heavily on ‘foreign’ low level debug implementations. These implementations are often completely machine dependent. Writing a debug adapter based on cdb would provide instant TIDE support across all platforms that cdb runs on.

The extensible graphical debugger Deet [HK97], which is based on cdb, is very closely related to our work. Although it can only be used to debug C programs, it does show how to use gdb as a vehicle to gather low level debugging events. The use of debugging ‘nubs’ by both cdb and Deet closely resembles the role of debug adapters in our work.

Another example of a related approach is described in [Sos95], where the Dynascope program directing tool is described. Dynascope is language independent because it offers a procedural interface for debuggers at an abstraction level below that of high-level languages. Debuggers that want to make use of this interface and want to offer high-level language support are forced to implement the mapping between the low abstraction level offered by Dynascope, and the high-level language.

Mainstream debugging technology either totally ignores the issue of reusability (for instance the Microsoft Visual C++ debugger), or supports a small set of different languages on a specific platform (SUN’s dbx is a good example).
The only mainstream debugger we know of that supports multiple languages on multiple platforms is the GNU debugger gdb [SPSS00]. The implementation of gdb is also quite monolithic, making the addition of new features very difficult. As a result, the basic set of features offered by gdb has not changed much over the last couple of years.

The effort invested in gdb over the years to get it running on a large number of unix platforms did prompt us to build a debug-adapter on top of it. This debug-adapter yields TIDE support for all the platforms that gdb runs on.

4.7 Conclusions

We have introduced a framework for distributed debugging, by systematically building on well known sequential debugging techniques and implementations. The resulting distributed debugger prototype is unique, both in its simple design and in its flexibility towards the support of new source languages.

One of the strengths of the TIDE approach is that it introduces an extremely clear separation between the platform dependent pieces (debug-adapter) and the platform independent parts. In Table 4.4 the size of the different components of TIDE are shown. The small size of the debug-adapter compared to the platform independent parts is a clear indication of the amount of reuse that can be obtained with the TIDE approach.

The performance of TIDE is comparable to that of traditional sequential debuggers, because primitive events are filtered locally when there is no need for them. Only interesting events are processed in a distributed fashion.

Because of the modularity and simplicity of our framework, we believe it to be a solid base for future experiments. Before starting any new experiments however, we need to add support for more source languages, especially for languages based on paradigms other than the imperative or functional paradigm, for instance logical languages. Our work on debugging of ASF+SDF specifications (see Chapter 8) shows that support for backtracking is straightforward. Extensions in the other direction, more debugging functionality, are also needed.

It will be interesting to find out which existing debugging features can be implemented in a generic setting as presented in this chapter.

We also would like to extend the framework to allow event abstraction by grouping basic events into combined “abstract events”. By visualizing these abstract events [Kun95], we could offer more help in understanding the distributed system being debugged at the application level rather than at the implementation level.
<table>
<thead>
<tr>
<th>Component</th>
<th>Language</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control center</td>
<td>Java</td>
<td>1643</td>
</tr>
<tr>
<td>Java adapter library</td>
<td>Java</td>
<td>1274</td>
</tr>
<tr>
<td>C adapter library</td>
<td>C</td>
<td>913</td>
</tr>
<tr>
<td>Debug tool library</td>
<td>Java</td>
<td>1704</td>
</tr>
<tr>
<td>Total basic TIDE infrastructure</td>
<td></td>
<td>4534</td>
</tr>
<tr>
<td>Process-list tool</td>
<td>Java</td>
<td>415</td>
</tr>
<tr>
<td>Source viewer</td>
<td>Java</td>
<td>920</td>
</tr>
<tr>
<td>Process viewer</td>
<td>Java</td>
<td>894</td>
</tr>
<tr>
<td>Rule tracer</td>
<td>Java</td>
<td>677</td>
</tr>
<tr>
<td>Rule viewer</td>
<td>Java</td>
<td>472</td>
</tr>
<tr>
<td>Animation viewer†</td>
<td>Java</td>
<td>3457</td>
</tr>
<tr>
<td>Total debug tools</td>
<td></td>
<td>6844</td>
</tr>
<tr>
<td>GDB debug adapter</td>
<td>Java</td>
<td>940</td>
</tr>
<tr>
<td>Java debug adapter</td>
<td>Java</td>
<td>1363</td>
</tr>
<tr>
<td>ToolBus debug adapter</td>
<td>Java</td>
<td>906</td>
</tr>
<tr>
<td>Tcl/Tk debug adapter</td>
<td>C</td>
<td>599</td>
</tr>
<tr>
<td>ASF+SDF debug adapter</td>
<td>C</td>
<td>185</td>
</tr>
<tr>
<td>Total debug adapters</td>
<td></td>
<td>4448</td>
</tr>
<tr>
<td>Total code size of TIDE</td>
<td></td>
<td>15817</td>
</tr>
</tbody>
</table>

†Implemented by H.A. de Jong in the context of his masters thesis [dJ99].

Table 4.4: Size of TIDE components