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.
2 TOOLBUS

The TOOLBUS [BK98] is a software coordination architecture that utilizes a scripting language based on process algebra [BW90] to describe the communication between software tools. A TOOLBUS script describes a number of processes that can communicate with each other and with tools existing outside the TOOLBUS. In the current TOOLBUS implementation, every tool is implemented as a single operating system process. A language dependent adapter that translates between the internal TOOLBUS data format and the data format used by the individual tools makes it possible to write every tool in the language best suited for the task(s) it has to perform.

Most of the work in this thesis is based on the TOOLBUS software coordination architecture. In some sense it is the _leitmotif_ for our work, so it seems only appropriate that we start this first part by introducing the TOOLBUS.

One of the most striking developments in the software industry over the last 10 years is the shift towards _component based software development_. Modern software systems are often designed using a number of loosely coupled components. These components are then combined using a _component or coordination_ architecture. Typical advantages of a component based approach are increased software reuse and easier maintenance.

Some of the well known component architectures used in industry today include OMG’s CORBA [Cor99], Microsoft’s DCOM and SUN’s JavaBeans [Ham99]. However, these systems are fairly low level, and can also be described as _wiring standards_. More high level coordination architectures that are actually used in an industrial setting are very hard to find, but the Java based InfoBus from SUN comes close. A good discussion on the use of component technology in general and CORBA and DCOM in particular can be found in [Szy97].

The TOOLBUS architecture is characterized by its formal basis in the form of process algebra [BW90], and the use of a generic data exchange format makes this architecture truly language independent.

A TOOLBUS script describes a number of processes inside the TOOLBUS that can communicate with each other and with tools existing outside the TOOLBUS (Figure 2.1).

It is important to note that TOOLBUS tools can only communicate with processes in the TOOLBUS. They are not allowed to communicate directly with each other without interven-
2 / TOOLBUS

Figure 2.1: The TOOLBUS software application architecture

tion of the TOOLBUS. This ensures that the TOOLBUS script has complete control over the communication between tools.

Each tool is connected to the TOOLBUS using an adapter. An adapter is some interface code that is responsible for the connection with the TOOLBUS, and converts between “native” data formats used inside a tool and the common data exchange format used by the TOOLBUS. TOOLBUS adapters are language specific. This means each language (implementation) needs its own adapter to connect to the TOOLBUS. At this point we have about a dozen adapters for a wide range of languages varying from Tcl/Tk to COBOL.

TOOLBUS processes are described as process expressions which are built using primitive TOOLBUS actions and process composition operators. The following sections give an overview of the most important TOOLBUS primitives and operators. A complete overview can be found in Appendices A, B, and C.

2.1 Communication between TOOLBUS Processes

There are two mechanisms available for processes in the TOOLBUS to communicate with each other, message passing and selective broadcasting. A process can synchronously send a message using the snd-msg primitive which must be received by another process using the rec-msg primitive. Both of these primitives take a variable number of arguments. Data is transferred between sending and receiving process using matching. The argument terms of the snd-msg and rec-msg primitives are matched against each other. These argument terms can contain result variables, indicated by a trailing question mark (?). When such a variable matches with a subterm, that subterm is assigned to the variable. For example, if communication occurs between:

snd-msg(text("Hello world!"))
and

\[ \text{rec-msg(text(T?))} \]

The value "Hello world!" is assigned to T.

A process can send a note using `snd-note` to all processes that have subscribed, using `subscribe`, to that particular note type. The receiving processes read notes asynchronously using `rec-note`, at low priority. Transmitting notes amounts to asynchronous selective broadcasting. Data transfer is again accomplished using matching.

### 2.2 Communication between TOOLBUS and Tools

A TOOLBUS process can initiate communication with a tool by sending a message to a tool using `snd-do`, or `snd-eval` when an answer is expected. A process can receive the answer to a `snd-eval` request using the `rec-value` action.

A tool can initiate communication by sending an `event` to the TOOLBUS. A TOOLBUS process receives this event using the `rec-event` primitive and must acknowledge the event using the `snd-ack-event` primitive.

The execution and termination of the tools attached to the TOOLBUS as well as their connection/disconnection can be controlled explicitly. The `execute` action starts a new tool, while the `rec-connect` action waits for a tool to connect itself to the TOOLBUS. The `snd-disconnect` action actively disconnects a tool, while the `rec-disconnect` action waits until a tool disconnects itself from the TOOLBUS.

### 2.3 Process Composition

More complex processes can be created using process composition operators for choice (+ operator), sequential composition (\( . \) operator), parallel composition (\( | | \) operator), iteration (binary \( * \) operator) and guarded (conditional) execution (the `if-then-\( f i \)` operator). The test part of the `if-then-\( f i \)` uses expressions as discussed in the next section. The `process creation` primitive `create` can be used to create new process instances.

Process algebra semantics of the `+` operator (and consequently the \( | | \) operator because its semantics are expressed in terms of `+` and `.`) demand a non-deterministic choice between the alternative process expressions. In the TOOLBUS this is modeled by random selection of alternatives when both alternatives are valid.

### 2.4 Types and Variables

The TOOLBUS uses a common datatype called `ATerm` to represent data. `ATerm` stands for Annotated Term. The annotation mechanism is not visible from within TOOLBUS scripts,
and will be discussed in Chapter 6 in more detail. Only ATerms can be exchanged between tools and the TOOLBUS.

A limited set of operations is available from within TOOLBUS scripts to analyze and transform these terms.

All terms within the TOOLBUS are typed. The TOOLBUS defines a number of basic types for booleans, integers, reals, strings, and binary strings. Complex types can be formed using a list constructor or function application. The type term is a supertype of all other types.

The let-in-endlet construction makes it possible to declare variables. Assignment to variables is possible via the := operator. On the right-hand side of this operator, expressions can be given using a number of prefix functions operating on values over the basic types. Typical functions include the (in)equality check (equal and not-equal), functions operating on booleans like and, or and not, and functions operating on lists like first and next. Appendix C contains a complete overview of all functions available in expressions.

2.4.1 A Producer/Consumer Example

In Figure 2.2 an example is presented that models the relationship between a producer and multiple consumers. The producer produces numbers that are printed by a pair of consumer processes. Note that this example uses some TOOLBUS primitives that have not been discussed yet, namely the printf action that can be used to write output to the screen, and the delta action that represents deadlock. Figure 2.2 also shows the output of the TOOLBUS script. product is a user-defined function symbol used to match corresponding snd-msg and rec-msg actions. Note that because the delta action is never executed, it can effectively be used in combination with the iteration operator to implement an endless loop.

2.5 A Calculator Example

In this example a TOOLBUS script is presented that connects two tools, a calculator tool that calculates expressions and a user interface tool that asks the user for an expression and presents its value as result.

This TOOLBUS script contains two processes, a USER-INTERFACE process (Figure 2.3) that handles user interface events and a CALC process (Figure 2.4) communicating with the calculator tool.

In addition to these processes, two tools are introduced by tool declarations. The string following command = will be executed as a command by the underlying operating system to create an instance of the tool. A tool declaration also introduces a new type, that can later be used to declare tool identifier variables of that type.

The USER-INTERFACE process (Figure 2.3) uses three variables. The first one, UI, is a tool identifier of type ui. The second variable E contains an expression to be calculated, the third variable V contains the calculated result.

The USER-INTERFACE process first starts the user interface tool. The variable UI is a
The producer produces consecutive numbered products.

% PROCESS PRODUCER IS
let
  N : int
in
  N := 1 .
  ( snd-msg(product(N)) .
      N := add(N,1)
  ) * delta
endlet

The consumer consumes any product it can get hold of.

% PROCESS CONSUMER(ID : term) IS
let
  N : int
in
  ( rec-msg(product(N?)) .
      printf("CONSUMER(%t) consumes product %d\n", ID, N)
  ) * delta
endlet

toolbus(PRODUCER, CONSUMER(1), CONSUMER(2))

----- Output -----
CONSUMER(1) consumes product 1
CONSUMER(2) consumes product 2
CONSUMER(2) consumes product 3
CONSUMER(1) consumes product 4
CONSUMER(2) consumes product 5
...

Figure 2.2: Producer/consumer example

result occurrence of UI, because it is followed by a question mark (?). When a variable is used as a result variable a value is assigned to it, in contrast with a value occurrence of a variable (without a following ?), when the current value of the variable is substituted. In this case, the tool identifier for the new instance of the user interface tool (ui) is assigned to UI.

After starting the ui tool, the USER-INTERFACE process enters a loop waiting for expr events from the newly created user interface tool. Such an event is generated when the user enters an expression and wants to evaluate it. At this point the user interface tool generates an
tool ui is { command = "wish-adapter -script ui-calc.tcl" }

process USER-INTERFACE is
let
  UI : ui,
  E : str,
  V : term
in
  execute(ui, UI) .
  ( rec-event(UI, expr(E)) .
    snd-msg(calc, expr(E)) .
    rec-msg(calc, expr(E, V)) .
    snd-ack-event(UI, expr(E, V))
  ) *
  rec-event(UI, quit) .
  snd-ack-event(UI, quit) .
  shutdown("Goodbye!")
endlet

Figure 2.3: The USER-INTERFACE process

expr event, for instance expr("3+4"). The expression "3+4" is assigned to the variable E, and sent to the CALC process for evaluation using the snd-msg action. The result is received in the rec-msg action and returned to the user interface tool using snd-ack-event.

The loop continues until the user interface tool generates a quit event, for instance when the user presses the Quit button.

Now we turn our attention to the CALC process (Figure 2.4). It starts the calculator tool and waits for calculation requests. It sends them to the calculator tool and sends the result back.

The last construct of every TOOLBUS script is the TOOLBUS configuration that starts a number of processes in parallel. In this case the processes USER-INTERFACE and CALC are created and execution begins:

toolbus(USER-INTERFACE,CALC)
tool calc is { command = "calc" }

process CALC is
let
  Calc : calc,
  E : str,
  V : term
in
  execute(calc, Calc?).
  ( rec-msg(calc, expr(E?)).
    snd-eval(Calc, expr(E)).
    rec-value(Calc, V?).
    snd-msg(calc, expr(E, V))
  ) * delta
endlet

Figure 2.4: The CALC process

2.6 The Connection between TOOLBUS and Tools

In order to connect tools written in an arbitrary programming language, an adapter is needed for that language. An adapter is a piece of software that is used to establish connections with the TOOLBUS, and to convert data between the format used internally by the language implementation and the TOOLBUS ATerm format.

Adapters have been developed for a number of languages, including C, Java, Perl, Tcl/Tk, Python, SwiProlog, ASF+SDF, and even COBOL. From the TOOLBUS point of view, there is no semantic difference between tools written in any of these languages, making it the ideal vehicle to construct complex distributed systems using tools written in several languages without suffering from the usual interoperability problems.

Using special purpose adapters often makes it possible to reuse programs off-the-shelf, without even recompiling them. Some examples of the kind of tools we can now use as TOOLBUS tools are the interactive plotting program gnuplot, the text editor emacs, and the animation tool samba.

2.7 Debugging TOOLBUS Applications

An important feature of the TOOLBUS is the built-in monitor protocol, which is an extension of the regular communication protocol between TOOLBUS and tools. A tool can send two
different message to the TOOLBUS: a snd-value message as a reply to an earlier snd-eval received from the TOOLBUS, and a snd-event to signal an event that originated in the tool itself.

A regular tool can receive three different messages from the TOOLBUS, a rec-eval message to perform some operation that returns some value back to the TOOLBUS (using snd-value), the rec-do message when no result is expected back, and the rec-ack-event as a indication that a snd-event has been processed.

A monitor tool introduces three special tool types: the logger, viewer, and controller. A logger tool can be used to log process and tool activity, the viewer tool is a logger tool that can also control the execution speed of TOOLBUS processes, for instance to allow single stepping. The controller tool is a viewer tool that is also capable of changing the internal state of processes.

The current TOOLBUS distribution contains a viewer tool written in the user interface scripting language Tcl/Tk [Ous94]. Using the monitor protocol, the viewer tool is notified of state changes in the TOOLBUS. For instance, whenever a new TOOLBUS process is created, the viewer tool is informed of the name and process-id of the new process.

The standard viewer tool implements a debugging interface consisting of a source window and a TOOLBUS window. Figure 2.5 shows the viewer while stepping through the calculator example discussed in Section 2.5.

In the source window, the TOOLBUS script being executed is displayed, and actions that are executed are highlighted. The user can run or step through the execution of processes, and can set breakpoints by double clicking on a specific source line. In the TOOLBUS window a picture is drawn of the current processes and tools connected to the TOOLBUS. In this picture, arrows are drawn indicating communication taking place between tools and processes. Double clicking on a process in the TOOLBUS window opens a new window containing the variables of the selected process and their current values.

Although the debugging approach described in this section has proved very useful, it has two major drawbacks:

1. The main debugging component, the viewer, is a monolithic piece of Tcl/Tk code and is therefore hard to extend.

2. Only the external communication behavior of the tools can be observed. The tools connected to the TOOLBUS are treated as black boxes. There is no way to inspect the local state of tools.

We will show that using the generic debugging techniques described in this thesis, we can eliminate these problems.
In this section, we will compare the TOOLBUS to various component architectures and coordination architectures. While the term component architectures originates from the software industry, the term coordination architecture originates from the academic world. Although both types of architectures focus on composition of software components to build distributed systems, their foundations are different. As can be expected, component architectures focus on economic advantages of using components: increased productivity of software engineers and programmers, better project control and easier software reuse. Coordination architectures tend to focus on more ‘academic advantages’ like theoretical foundations of the architectures, expressiveness of formalisms and the theoretical possibility to prove certain properties like correctness, compositionality and real time characteristics. Because of these different viewpoints, components in component architectures are often quite large, while components in
coordination architectures tend to be more fine grained.

We will discuss a number of component architectures (2.8.1 - 2.8.3), and two coordination architectures (2.8.4 - 2.8.5). The TOOLBUS can be seen as a mixture of the two. As a coordination architecture it originated from the academic world and it has a clear theoretical foundation in the form of process algebra. When we look at the TOOLBUS as as component architecture we find that its components are not very fine grained and software engineering benefits like component reuse and programmer productivity are important issues.

2.8.1 CORBA

CORBA has been developed by the Object Management Group (OMG). This consortium set out to solve the problem of interaction between object-oriented systems implemented in different languages and running on different platforms. The solution they came up with was the Common Object Request Broker Architecture (CORBA).

The target of CORBA was to make it possible to connect a wide variety of languages, implementations, and platforms. This ambitious goal has one major downside. Individual CORBA implementations cannot talk to each other on an efficient binary level, but must communicate using high-level protocols.

CORBA essentially offers a form of portable remote method calls. As such, it offers a much cleaner model than traditional techniques based on remote procedure calls or even lower level abstractions like direct socket communication. A number of object service specifications (CORBA services) try to lift this level of abstraction even higher, by standardizing support for high-level services that are important for enterprise level applications. Included are services that provide a security mechanism, object persistence, a transaction mechanism, support for change management, concurrency, and an event notification service. Meta-information can be specified using the CORBA Interface Description Language (IDL).

Note that CORBA does not offer any sophisticated memory management support. This means that without any standardized way to solve these problems, ad hoc solutions will be used in practice to avoid memory leaks.

2.8.2 DCOM

DCOM is the distributed version of the Microsoft's Common Object Model (COM), which has evolved from the OLE (Object Linking and Embedding) framework on Microsoft Windows platforms.

COM is first and foremost an efficient, low-level binary standard. A component in COM consists of a number of interfaces, each containing a table with function pointers (a vtable). Data is transferred between components on different platforms using a common representation (Network Data Representation, NDR). COM offers a security mechanism, and a complicated persistence mechanism. Meta-information is specified using the COM Interface Description
Language (IDL), and then compiled into type libraries.

If we try to look at DCOM as a component architecture instead of a wiring standard, we find that much of its features are needlessly complex, partly because of the emphasis on efficiency, and partly because the DCOM standard has evolved from a complex platform specific framework. Typical examples of this complexity are the error prone way in which distributed memory management is organized using reference counting, and the use of ‘dispatch’ and ‘dual’ interfaces to make it possible to call component methods in a generalized matter.

2.8.3 Java based component architectures

The Java success story is also extending towards component software. Any component architecture that is based on ‘pure’ Java can automatically use standard Java features like introspection to retrieve meta information and serialization to store persistent objects.

JavaBeans  The JavaBeans standard introduces a very lightweight approach to component software. The standard is aimed at creating small to medium-sized controls and is therefore not very suitable as a ‘general purpose’ component architecture. However, the JavaBeans architecture is general enough to be used as a base layer for the InfoBus component architecture discussed below.

RMI  The Java Remote Method Invocation standard (RMI) can be used to call methods across Java virtual machines and across networks. It can be seen as a component architecture, as it offers features like object copying across network connections, a naming service to find remote objects, and most important, fully distributed garbage collection.

InfoBus  The InfoBus architecture is based on the concept of the information bus metaphor, not unlike the TOOLBUS. Objects can plug into this information bus, and listen to events they are interested in. The InfoBus is built on top of the JavaBeans standard, and adds standardization of data transfer between JavaBeans.

JavaSpaces  The JavaSpaces architecture offers an interesting approach to component software. Instead of being based on a message passing or remote procedure call paradigm, JavaSpaces is based on a shared memory paradigm. It is also interesting because it is an example of academic cross-fertilization. Many of the ideas behind JavaSpaces are directly derived from the Linda coordination architecture [Gel85].

In the JavaSpaces architecture, components communicate using a number of object spaces. These object spaces represent an abstract form of shared memory. Each component can put objects into such an object space, inspect them, and remove objects from an object space.

The JavaSpaces implementation relies heavily on RMI for communication between components and object spaces.

1COM’s IDL is completely unrelated to CORBA’s IDL.
2.8.4 Linda

Most coordination and component architectures are based on message passing. In contrast, Linda [Gel85] is a coordination architecture that is based on the shared memory paradigm. All components communicate using a central ‘tuple space’. Linda offers a small number of primitives to place tuples in this tuple space, inspect tuples, and to take tuples out of the tuple space again. Reading tuples is done using matching.

2.8.5 Manifold

Most coordination languages aim at making a clear separation between communication and computation. Manifold [Arb96] does this by separating components (processes) into two categories: worker processes and manager processes. Worker processes perform computations on data transferred through channels connected to the input and output ports of the processes. Manager processes control the creation and destruction of processes and channels.

2.8.6 TOOLBUS strengths

One common characteristic among the previously discussed component architectures is that the medium used by components to communicate is fixed. CORBA, COM, and Java’s RMI all use remote method invocation. The only thing configurable in such architectures is which components are actually connected. The InfoBus offers a couple of different data exchange methods like a subscription and a broadcast mechanism. JavaSpaces offer the object spaces to communicate.

The TOOLBUS on the other hand can be programmed using process descriptions in the form of T-scripts. The possibility to express the coordination logic of a distributed application in a special purpose coordination layer offers a number of advantages.

Because the component interaction is described at a high level of abstraction, language constructs can be used that are well suited for this task. A typical example of this kind of construct is the non-deterministic choice operator (+). Not many languages have a use for this kind of operator, but it is very useful when describing the behaviour of (communicating) processes.

The TOOLBUS enforces that the coordination logic is “pulled out” of the components, having the effect of making them more general. This in turn promotes software reuse at the component level. As one author puts this “The TOOLBUS promotes the use of generic solutions to specific problems” [dJ99]. It is not uncommon that a set of TOOLBUS tools is so generic that a number of different applications can be constructed by only changing the TOOLBUS script that connects them.

The use of a specialized language to describe the possible interactions between tools has another major benefit. Much of the “meta-information” that needs to be specified manually with the other architectures, for instance using some IDL, can be derived automatically in the TOOLBUS case. Automatic derivation of component interface descriptions is a major
The TOOLBUS as a component/coordination architecture / 2.8

strength of the TOOLBUS, and actually changed the way we design and implement software. One of the most important deliverables of the design process is now a fully functional T-script that acts as an "executable design specification". Based on this script, a set of tool interfaces is generated that provide an excellent starting point for the implementation of the individual tools. When the initial design changes, the T-script needs to be changed as well. In this case the tool interfaces are regenerated, and the tools can be adapted to fit their changed interfaces.

When compared to the TOOLBUS the other component architectures operate at a lower level of abstraction. Except for the JavaSpaces architecture, the other approaches can be characterized as "wiring standards" [Szy97]. If a developer wants to isolate the communication behavior of an application on a higher level of abstraction he is forced to introduce one or more controlling components that regulate this communication. But because there is no standard approach for this, each developer has to reinvent the wheel in this area and different approaches are incompatible. The TOOLBUS architecture offers exactly this: a central component that controls the communication between other components in a system. This in combination with a specialized language to describe this communication offers a powerful approach to tackle the intricate problem of developing heterogeneous distributed applications.

2.8.7 TOOLBUS weaknesses

When we compare the TOOLBUS to the previously discussed component architectures, we find weaknesses in two categories: performance and functionality.

Performance problems The TOOLBUS performance is lacking because of two major design decisions. The first decision is the choice for a centralized approach. Conceptually, all communication between components and therefore all data transfers go through the TOOLBUS. This means that in many distributed applications the TOOLBUS will be the bottleneck. In order to lift this bottleneck while keeping the conceptually simple view of a centralized mediating bus intact, it will be necessary to transparently route the actual data communication directly between tools. Future research is needed to find out if it is possible to use static or dynamic T-script analysis for this purpose.

The second design decision that influences the performance is the use of a common data exchange format in the form of ATerms. Most other component architectures use a similar approach to exchange data between components located on different hosts or in different processes on the same host. But in the TOOLBUS case, ATerms are used to exchanged data even in the case where components are located in the same operating system level process.

Lacking functionality If we make a pass over the features offered by the other component architectures, we can quickly identify some important areas in which the TOOLBUS is lacking support:

- Distributed garbage collection
2 / TOOLBUS

- Transactions
- Security
- Persistency

Most of these shortcomings can be overcome by programming the support using the T-script, in combination with some specialized tools. This has two drawbacks however. First, a T-script should be a relatively small and clear description of the communication allowed between components. Adding support for some of the missing TOOLBUS features might reduce the amount of clutter in T-scripts significantly.

Secondly, because this kind of support is not standardized, each TOOLBUS programmer has to "reinvent the wheel". In cases where support from the components is required, different solutions might not be compatible with each other. This in turn would make component reuse more difficult. This suggests that it would be beneficial to develop standard solutions for these problems and distribute these standard solutions with the TOOLBUS.

2.8.8 Enlarging the TOOLBUS application domain

To widen the application domain of the TOOLBUS architecture, the efficiency has to be improved drastically. We believe that static and dynamic analysis of T-scripts can be used to transparently break the centralized nature of the TOOLBUS. This could lead to an architecture whose efficiency is comparable to that of CORBA. The DCOM approach of using direct method calls when working with in-process components will be hard to match, but comparable performance can be reached in the case where both approaches need to use interprocess communication.

Standardization of features like distributed garbage collection, transactions, a security mechanism, and persistence is important to increase the reusability of TOOLBUS tools. Another important area in which this kind of standardization can have a major impact is the area of compound documents. This includes support for things like drag and drop, some clipboard mechanism, and a standardized way to mix document content from different components.