ToolBus: The Next Generation

Hayco de Jong[1] and Paul Klint[1,2]

[1] Centrum voor Wiskunde en Informatica,
{Hayco.de.Jong,Paul.Klint}@cwi.nl
[2] Informatics Institute, University of Amsterdam, The Netherlands

Abstract The TOOLBUS is a software coordination architecture for the integration of components written in different languages running on different computers. It has been used since 1994 in a variety of projects, most notably in the complete renovation of the ASF+SDF Meta-Environment. In this paper we summarize the experience that has been gained in these projects and sketch preliminary ideas how the TOOLBUS can be further improved. Topics to be discussed include the structuring of message exchanges, crash recovery in distributed applications, and call-by-value versus call-by-reference.

1 Generic Language Technology

Our primary interest is *generic language technology* that aims at the rapid construction of tools for a wide variety of programming and application languages. Its central notion is a *language definition* of some programming or application language.

The common methodology is that a language is identified in a given domain, that relevant aspects of that language are formally defined and that desired tools are generated on the basis of this language definition. This generative approach is illustrated in Figure 1. Using a definition for some language *L* as starting point, a generator can produce a range of tools for editing, manipulating, checking or executing *L* programs.

Language aspects have to be defined, analyzed, and used to generate appropriate tooling such as compilers, interpreters, type checkers, syntax-directed editors, debuggers, partial evaluators, test case generators, documentation generators, and more.

Language definitions are used, on a daily basis, in application areas as disparate as Cobol renovation, Java refactoring, smart card verification and in application generation for domains including finance, industrial automation and software engineering. In the case of Cobol renovation, the language in question is Cobol and those aspects that are relevant for renovation have to be formalized. In the case of application generation, the language in question is probably new and has to be designed from scratch.

1.1 One realization: the ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [25,10] is an incarnation of the approach just described and covers both the interactive development of language definitions and the generation of tools based on these language definitions.

In this paper we are primarily interested in the *software engineering aspects* of building such a system. Starting point is the ASF+SDF Meta-Environment as we had

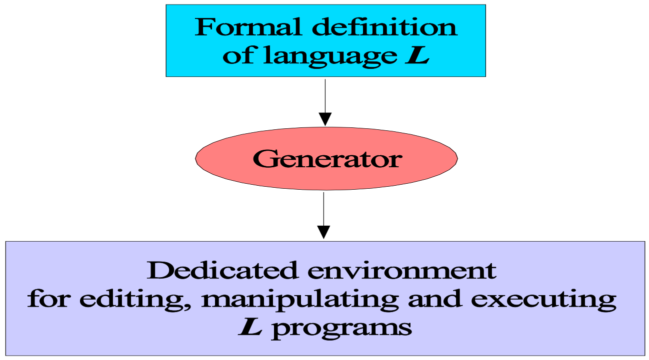

Figure 1. From language definition to generated programming environment

completed it in the beginning of the 1990's. This was a monolithic 200 KLOC Lisp program that was hard to maintain. It had all the traits of a legacy system and was the primary motivation to enter the area of system and software renovation.

1.2 Towards a component-based architecture

We give a brief time line of the efforts to transform the old, monolithic, implementation of the Meta-Environment into a well-structured, component-based, implementation.

In 1992, first, unsuccessful, experiments were carried out to decompose the system into separate parts [1]. The idea was to separate the user-interface and the text editor from the rest of the system. The user-interface was completely re-implemented as a separate component and as text editor we re-used Emacs. In hindsight, we were unaware of the fact that we made the transition from a completely sequential system to a system with several concurrent components. Unavoidably, we encountered hard to explain deadlocks and race conditions.

In 1993, a next step was to write a formal specification of the desired system behavior [34] using PSF, a specification language based on process algebra and algebraic specifications [27]. Simulation of this specification unveiled other, not yet observed, deadlocks. Although this was clearly an improvement over the existing situation, this specification approach also had its limitations and drawbacks:

– The specification lacked generality. It would, for instance, have been a major change to add the description of a new component.
– The effort to write the PSF specification was significant and there was no way to derive an actual implementation from it.

In 1994, the first version of the TOOLBUS was completed [4,6]. The key idea was to organize a system along the lines of a software bus and to make this bus programmable by way of a scripting language (TSCRIPT) that was based on ACP (Algebra of Communicating Processes, [8]). Another idea was to use a uniform data format (called TOOLBUS terms) to exchange data between TOOLBUS and tools. At the implementation level, TSCRIPTs were executed by an interpreter and communication between tools and TOOLBUS took place using TCP/IP sockets. In this way, multi-language, distributed, applications could be built with significantly less effort than using plain C and sockets.

Based on various experiments [30,18,26,20], in 1995 a new version of the TOOLBUS was designed and implemented: the Discrete Time ToolBus [5,7,2]. Its main innovations were primitives for expressing timing considerations (delay, timeout) and for operating on a limited set of built-in data-types (booleans, integers, reals, lists). The Discrete Time TOOLBUS has been used for the restructuring of the ASF+SDF Meta-Environment [11]. A first version was released in 2001 [10].

In the meantime, the exchange format has also evolved from the TOOLBUS terms mentioned above to ATerms [12]: a term format that supports maximal subterm sharing and a very concise, sharing preserving, binary exchange format. ATerms decrease memory usage thanks to sharing and they permit a very fast equality test since structural equality can be replaced by pointer equality thanks to the maximal subterm sharing.

Another line of development is the TOOLBUS Integrated Debugging Environment (TIDE) described in [31].

Today, beginning 2003, it turns out that the original software engineering goals that triggered the development of the TOOLBUS have been achieved and that the Meta-Environment can now be even further stretched than anticipated [13]. Therefore, it is time for some reflection. What have we learned from this major renovation project and what are the implications for the TOOLBUS design and implementation?

1.3 Plan of this Paper

In §2 we discuss component coordination, representation and computation and introduce the TOOLBUS: our component coordination architecture. Following, in §3, we demonstrate some of the TOOLBUS-features by means of an example. In §4 we show how we used the TOOLBUS in the ASF+SDF Meta Environment to migrate from a monolithic to a distributed architecture. Then, in §5 we elaborate on the various issues that we would like to tackle in a next generation of the TOOLBUS. We conclude the paper with an overview of the current status of our current implementation of this next generation TOOLBUS (§6) and some concluding remarks (§7).

2 The TOOLBUS Architecture

In [22] is was advocated that the overall architecture of a software system can be improved by separating *coordination* from *computation*. In addition to this, we also distinguish *representation* and use the following definitions:

- Coordination: the way in which program and system parts interact (using procedure calls, remote method invocation, middleware, and others).
- Representation: language and machine neutral data exchanged between components.
- Computation: program code that carries out a specialized task.

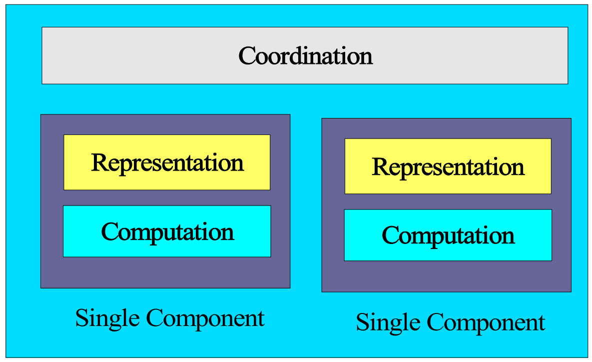

Figure 2. Separating coordination from computation

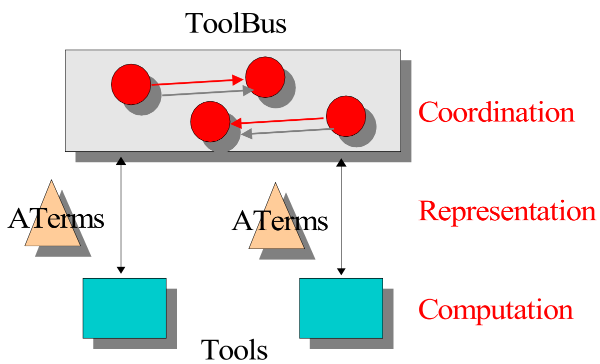

Figure 3. The TOOLBUS architecture

The assumption is now that *a rigorous separation of coordination, representation and computation leads to flexible and reusable systems.* This subdivision is sketched in Figure 2. Our TOOLBUS approach follows this paradigm and is illustrated in Figure 3

The goal of the TOOLBUS is to integrate tools written in different languages running on different machines. This is achieved by means of a programmable software bus. The TOOLBUS coordinates the cooperation of a number of tools. This cooperation is described by a TSCRIPT that runs inside the TOOLBUS. The result is a set of concurrent processes inside the TOOLBUS that can communicate with each other and with the tools. Tools can be written in any language and can run on different machines. They exchange data by way of ATerms.

A typical cooperation scenario is illustrated in Figure 4. A user-interface (UI) and a database (DB) are combined in an application. Pushing a button in the user-interface leads to a database action and the result is displayed in the user-interface. In a traditional approach, the database action is directly connected to the user-interface button by means of a call-back function. This implies that the user-interface needs some knowledge about the database tool and *vice versa*. In the TOOLBUS approach the two components are

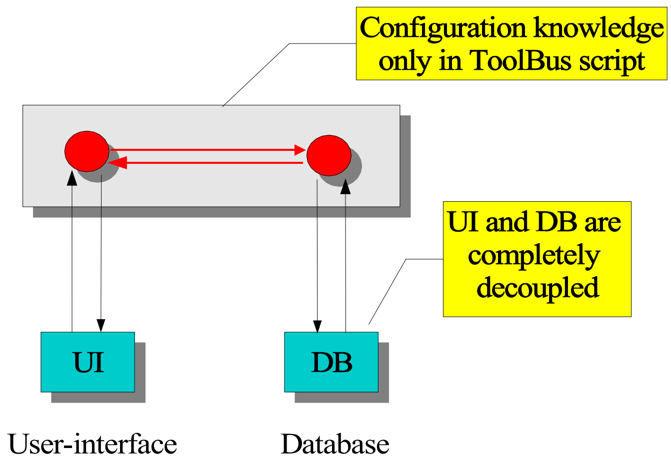

Figure 4. A typical cooperation scenario

completely decoupled: pushing the button only leads to an event that is handled by some process in the TOOLBUS. This process routes the event to the database tool (likely via some intermediary process) and gets the answer back via the inverse route. This implies that the configuration knowledge is now completely localized in the TSCRIPT and that UI and DB do not even know about each others existence.

The primitives that can be used in TSCRIPTs are listed in Table 1.

3 An example: the Address Book Service

To make the scenario from Figure 4 more concrete, we describe the construction of an address book holding (name, address) pairs. Typical uses include creating a new address, finding an address based on the name, etc. First we consider some aspects of the User Interface. An instance of the UI connects to the TOOLBUS and during the subsequent session, the user can:

create a new entry in the address book database;
delete an existing entry from the database;
search for an entry in the database;
update an existing entry in the database.

Each of these use cases can be described as a TOOLBUS process which, together with a process that explains how these use cases interact, form the TOOLBUS script describing our Address Book Service.

3.1 TOOLBUS processes for the Address Book Service

The ADDRESSBOOK process tells the TOOLBUS that an instance of our `address-book` tool is to be executed, followed by a loop which invokes *one of* the processes

Primitive	Description
`delta`	inaction ("deadlock")
`+`	choice between two alternatives (P_1 or P_2)
`.`	sequential composition (P_1 followed by P_2)
`*`	iteration (zero or more times P_1 followed by P_2)
`create`	process creation
`snd-msg`	send a message (binary, synchronous)
`rec-msg`	receive a message (binary, synchronous)
`snd-note`	send a note (broadcast, asynchronous)
`rec-note`	receive a note (asynchronous)
`no-note`	no notes available for process
`subscribe`	subscribe to notes
`unsubscribe`	unsubscribe from notes
`snd-eval`	send evaluation request to tool
`rec-value`	receive a value from a tool
`snd-do`	send request to tool (no return value)
`rec-event`	receive event from tool
`snd-ack-event`	acknowledge a previous event from a tool
`if ... then ... fi`	guarded command
`if ... then ... else ... fi`	conditional
	expressions
`||`	communication-free merge (parallel composition)
`let ... in ... endlet`	local variables
`:=`	assignment
`delay`	relative time delay
`abs-delay`	absolute time delay
`timeout`	relative timeout
`abs-timeout`	absolute timeout
`rec-connect`	receive a connection request from a tool
`rec-disconnect`	receive a disconnection request from a tool
`execute`	execute a tool
`snd-terminate`	terminate the execution of a tool
`shutdown`	terminate TOOLBUS
`attach-monitor`	attach a monitoring tool to a process
`detach-monitor`	detach a monitoring tool from a process

Table1. Overview of TOOLBUS primitives

`CREATE`, `UPDATE` or `DELETE` in each iteration. This construction, using the + operator ensures that at this level, the sub-processes can be regarded atomically: this means that for example no `DELETE` will happen during an `UPDATE`.

```
process ADDRESSBOOK is
let AB : address-book
in
  execute(address-book, AB?) .
  (
    CREATE(AB) + DELETE(AB) + SEARCH(AB) + UPDATE(AB)
  ) * delta
endlet
```

The operating system level details of starting the tool are defined in a separate section (one for each tool if multiple tools are involved):

```
  tool address-book is {
    command = "java-adapter -class AddressBookService"
  }
```

In this case, the TOOLBUS is told that our tool is written in Java, and that the main class to be started is called `AddressBookService`.

The `CREATE` process can be described as a TOOLBUS process as follows:

```
process CREATE(AB : address-book) is
let AID : int
in
  rec-msg(create-address) .
  snd-eval(AB, create-entry) .
  rec-value(AB, new-entry(AID?)) .
  snd-msg(address-created(AID))
endlet
```

The request to create a new address book entry is received and delegated to the tool, so it can update its state. In this case, our tool yields a unique id for reference to the new entry, which is returned as the result of the creation message. Note that communication between processes involves the matching of the arguments of `snd-msg` and `rec-msg`. The same holds for the communication between a process and a tool using `snd-eval` and `rec-value`. In all these cases, a *result variable* of the form `V?` gets a value assigned as the result of a successful match.

The `DELETE` process differs only from the CREATE process in that it does not need a return value:

```
...
  rec-msg(delete-address(AID?) .
  snd-do(AB, delete-entry(AID)) .
  snd-msg(address-deleted(AID))
...
```

The `SEARCH` process in our example implements but a single query: finding an address book entry by name. It shows how different results from a tool-evaluation request can be processed in much the same way that different messages are handled. Upon receiving a `find-by-name` message from another process, this request is delegated to the tool. Depending on whether or not the entry exists in the database, the tool replies with a `found` or a `not-found` message, respectively. This result is then propagated to the process that sent the initial `find-by-name` message.

```
process SEARCH(AB : address-book) is
let
  Aid : int,
  Name : str
in
  rec-msg(find-by-name(Name?)) .
  snd-eval(AB, find-by-name(Name)) .
  (
    rec-value(AB, found(Aid?)) .
    snd-msg(found(Aid))
  +
    rec-value(AB, not-found) .
    snd-msg(not-found)
  )
endlet
```

The `UPDATE` process is more interesting. It shows that each update of an address entry is *guarded*. A process wanting to update an entry first has to announce this fact by sending an `update-address` message, before it can do one or more updates to the entry. It then finishes the update by sending an `update-address-done` message. This message pair thus acts as a very primitive locking scheme. More elaborate schemes are very well possible, but are not discussed in this paper. Summarizing, the `UPDATE` process shows that *outside* the implementation of the address book service, we can enforce the order in which certain parts of the service are invoked, as well as mutual exclusion of some of its sections.

```
process UPDATE(AB : address-book) is
let
  AID : int,
  Name : str,
  Address : str
in
  rec-msg(update-entry(AID?)) .
  ( rec-msg(set-name(Name?)) .
    snd-do(AB, set-name(AID, Name))
  + rec-msg(set-address(Address?)) .
    snd-do(AB, set-address(AID, Address))
  ) *
  rec-msg(update-entry-done(AID))
endlet
```

3.2 TOOLBUS process for the User Interface

Because users can connect at any time to the TOOLBUS to start a session with the Address Book Service, the TOOLBUS itself does not execute instances of the UI (as it did with the address book tool). Instead `UITool` instances can connect, make zero or more requests to the service, and disconnect at their convenience. The following definition of the `UI` process shows how UI requests for the creation of a new entry and a name-change can be realized:

```
process UI is
let
  UITool : ui,
  AID : int,
  Name : str
in
  rec-connect(UITool?) .
  (
    rec-event(UITool, create-address) .
    snd-msg(create-address) .
    rec-msg(address-created) .
    snd-ack-event(UITool, create-address)
  +
    rec-event(UITool, update-name(AID?, Name?)) .
    snd-msg(update-entry(AID)) .
    snd-msg(set-name(Name)) .
    snd-msg(update-entry-done(AID)) .
    snd-ack-event(UITool, update-name(AID, Name))
  +
    ... /* more UI requests */
  )
  * rec-disconnect(UITool)
endlet
```

4 Application to the ASF+SDF Meta-Environment

As already sketched in §1.2, the TOOLBUS has been used to restructure the ASF+SDF Meta-Environment. It consists of a cooperation of 27 tools ranging from a user-interface, graph browser, various editors, compiler and interpreter, to a parser generator and a repository for parse trees. A simplified view is shown in Figure 5.

Our insight can be further increased by considering some statistics. Table 2 shows the relative sizes of the various implementation languages used in the Meta-Environment. In the column *language* the various languages are listed. In column *KLOC* the size (in Kilo Lines Of Code) is given for each language. The result is 107 KLOC for the whole system of which 4.6% are TSCRIPTS. If we consider the fact that ASF+SDF specifications are compiled to C code, another view is possible as well: 12 KLOC of ASF+SDF generates 170 KLOC of C code. Taking this generated code into account, the total size of the whole system amounts to 277 KLOC of which 1.8% are TSCRIPTS. This is compatible with the expectation that TSCRIPTS are relatively small and form high-level "glue" to connect much larger components.

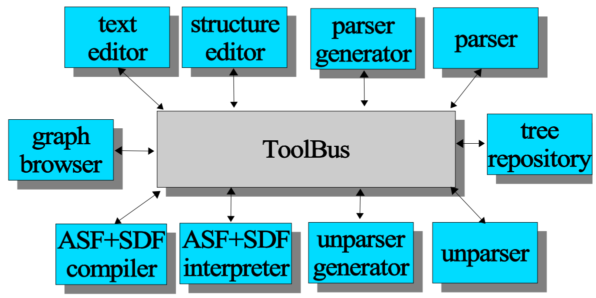

Figure 5. Architecture of the ASF+SDF Meta-Environment.

Language	KLOC†	Generated KLOC	Total KLOC
ASF+SDF	12	170 (C)	
C	80††		
Java, Tcl/Tk	5		
Makefiles, etc	5		
TSCRIPT	5		
Total LOC:	107	170	277
TSCRIPT	4.6%		1.8%

† Kilo Lines of Code excluding third party code such as emacs, dot, and the like.
†† This includes 10 KLOC (C code) for the TOOLBUS implementation itself.

Table2. Facts concerning implementation languages

Part of the generated C code is currently done by ApiGen[23]: an API generator which takes an SDF grammar as input and generates a C library which gives type-safe access to the underlying ATerm representation of the parse trees over this grammar.

Another conclusion from these facts is that low-level information for building the software (makefiles and configuration scripts) are of the same size as the high level TSCRIPTs. This points into the direction that the level of these build scripts should be raised. This conclusion will, however, not be further explored in this paper.

Another view is given in Table 3 where the frequency of occurrence of TSCRIPT primitives is shown. Clearly, sequential composition (`.`) is the dominant operator and sending/receiving (`snd-msg`, `rec-msg`) messages is the dominant communication mechanism, followed by communication with tools (`snd-do`, `snd-eval`). It may be surprising that parallel composition (`||`) is used so infrequently. However, one should be aware that at the top level all TOOLBUS processes run concurrently and that `||` is only used for explicit concurrency inside a process. The level of concurrence is therefore approximately 100 (104 process definitions and 3 explicit `||` operators).

Empirical evidence shows that:

– The TOOLBUS-based version of the ASF+SDF Meta-Environment is more flexible as illustrated by the fact that clones of the Meta-Environment start to ap-

Primitive	Number of occurrences
process definitions	104
tool definitions	27
`.` (sequential composition)	4343
`+` (choice)	341
`*` (iteration)	243
`||` (parallel composition)	3
`snd-msg`	555
`rec-msg`	541
`snd-note`	100
`rec-note`	24
`snd-do/snd-eval`	220
`rec-event`	56
`create`	58

Table3. Facts concerning TSCRIPT primitives

pear for other languages than ASF+SDF. Examples are Action Semantics [28] and Elan [14].
– Various components of the ASF+SDF Meta-Environment are being reused in other projects [18,9].

5 Issues in a Next-Generation TOOLBUS

The TOOLBUS has been used in various applications of which the Meta-Environment is by far the largest. Some of the questions posed by our users and ourselves are:

– I find it difficult to see which messages are requests and which are replies; can you provide support for this? See §5.1.
– If a tool crashes, what is the best way to describe the recovery in the TSCRIPT? See §5.2.
– I have huge data values that are exchanged between tools and the TOOLBUS becomes a data bottleneck; can you improve this? See §5.3.
– The TOOLBUS and tools are running as separate tasks of the operating systems. Would it not be more efficient to run TOOLBUS and tools in single task? See §6.

5.1 Undisciplined Message Patterns

The classical pattern of a remote procedure call is shown in Figure 6: a caller performs a call to a callee. During the call the caller suspends execution and the callee executes until it has computed a reply. At that point in time, the caller continues its execution.

Compare this simple situation with general message communication as shown in Figure 7: the caller continues execution after sending a message *msg1* to *Callee1* and may even send a message *msg2* to *Callee2*. At a certain point in time *Callee2* may send message *msg3* back to *Caller*. In this case, the three parties involved continue their

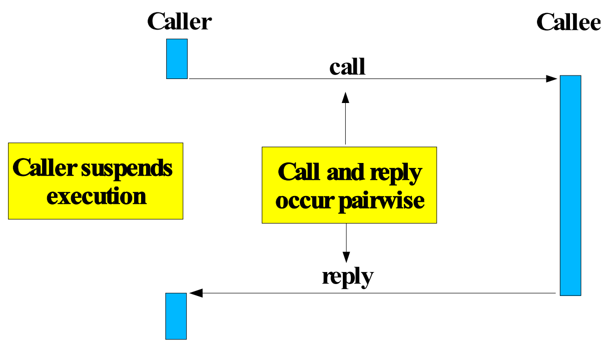

Figure 6. Communication pattern for remote procedure call

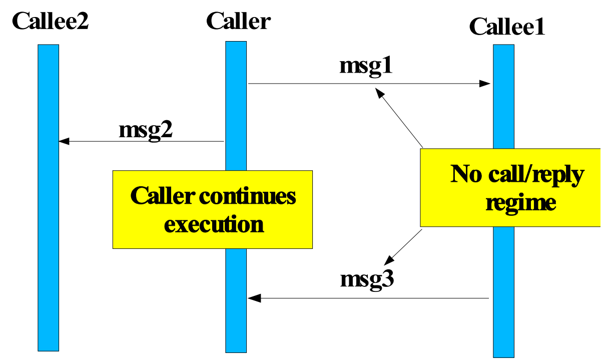

Figure 7. Communication pattern for general messages

execution while messages are being exchanged and there is no obvious pairing of calls and replies.

In the TOOLBUS case, a `snd-msg` and a `rec-msg` can interact with each other if their arguments match. A typical sequence is:

Process A:	**Process B:**
`snd-msg(calculate(E)) .`	`rec-msg(calculate(E?)) .`
... *other actions* ...	... *actions that compute value V* ...
`rec-msg(value(E, V?))`	`snd-msg(value(E, V))`

What we see here is that a form of call/reply regime is encoded in the messages: process B returns the value `V` that it has computed as `snd-msg(value(E, V))`. The `E` is completely redundant but serves as identification for process A to which message this is an answer.

The call/reply regime is thus implicitly encoded in messages. This makes error handling harder (which reply did not come?) and makes the TSCRIPTs harder to understand. This is particularly so, since unstructured combinations of `snd-msg`/`rec-msg` and sequential composition, choice, iteration and parallel composition are allowed.

The only solution for the above problems is to limit the occurrences of `snd-msg` or `rec-msg` in such a way that a form of very general call/reply regime is enforced. Our approach is to syntactically enforce that `snd-msg`/`rec-msg` or `rec-msg`/`snd-msg` may only occur in (possibly nested) pairs and that in between arbitrary operations are allowed. In fact, the matching `snd-msg` or `rec-msg` may be an arbitrary expression provided that all its alternatives begin with a matching `snd-msg` or `rec-msg`.

We replace thus

`snd-msg(req(E))` . *arbitrary process expression* . `rec-msg(ans(A?))`

by

`snd-msg(req(E))` { *arbitrary process expression* } `rec-msg(ans(A?))`

and also allow more general cases like:

`snd-msg (req(E))` { *arbitrary process expression* }
`( rec-msg(ans(A?)) + rec-msg(error(M?) )`

It is an interesting property of Process Algebra that every process expression can be normalized to a so-called *action prefix form*: a list of choices where each choice starts with an atomic action. An action prefix form has the following structure: $a_1.P_1 + a_2.P_2 + \ldots + a_n.P_n$. Using this property we can formulate the most general constraint that we impose on occurrences of `snd-msg` and `rec-msg`. Consider P_1 { Q } P_2 and let P_1' and P_2' be the action prefix forms of, respectively, P_1 and P_2. Our requirement is now that each choice in P_1' starts we a `snd-msg` and each choice in P_2' with a `rec-msg`, or *vice versa*. Note that this constraint can be checked statically.

5.2 Exception Handling

Exception handling is notorious for its complexity and impact on the structure of program code. The mainstream exception handling approach as used in, for instance, Java associates one or more exception handlers with a specific method call. If the call completes successfully, the handlers are ignored. If the call raises an exception, it is checked whether this exception can be handled locally by one of the given handlers. If not, the exception is propagated to the caller of the current code. This model does, however, not work well in a setting where multiple processes are active and the occurrence of an exception may require recovery in several processes.

Local Exception Handling We start with the simpler case of local error handling and introduce the *disrupt operator* (>>) proposed in LOTOS [15]. A process algebra variant of this operator is described in [19]. It has the form `P >> E`, where `P` describes the normal processing and `E` the exceptional processing. It adds the exception `E` as alternative to each atomic action in `P`. If the action prefix form of `P` is $a_1.P_1 + a_2.P_2 + \ldots + a_n.P_n$, then

$$P >> E \equiv (a_1 + E).(P_1 >> E) + .. + (a_n + E).(P_n >> E).$$

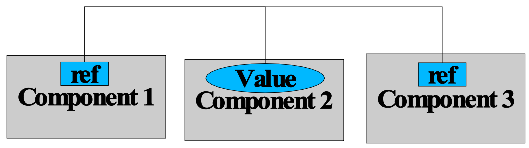

Figure 8. Call-by-reference in a distributed application

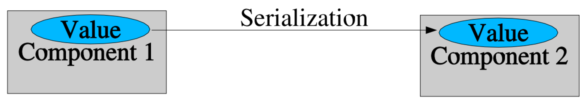

Figure 9. Call-by-value in a (Java-based) distributed application

Global Exception Handling Global exception handling in distributed systems is a very well-studied subject from the perspective of crash recovery and transaction management in distributed databases. An overview of rollback-recovery protocols in message-passing systems is, for instance, given in [21].

In the context of system reliability, the notion of a *recovery block* has been introduced by Randell [32]. Its purpose was to provide several alternative algorithms for doing the same computation. Upon completion of one algorithm, an acceptance test is made. If the test succeeds, the program proceeds normally, but if it fails a rollback is made to the system state before the algorithm was started and one of the alternative algorithms is tried. In [24] this idea is applied to backtracking in string processing languages. It turns out that the preservation of the system state can be done efficiently by only saving updates to the state after the last recovery point.

Recovery blocks also form the basis for Coordinated Atomic Actions described in [36]. Recovery blocks are intended for the error recovery in a single process. They can be generalized to *conversations* between more than one process: several processes can enter a conversation at different times but they can only leave it simultaneously, when all participating processes satisfy their acceptance test. In case one participant fails to pass its test, each participant is rolled back to the state when it entered the conversation.

We are currently studying this model since it can be fit easily in the TOOLBUS framework and seems to solve our problem of global exception handling. It is helpful that a backtrack operator similar to the one described in [24] has also been described for Process Algebra [3]. What remains to be studied is how the recovery of *tools* has to be organized. Most likely, we will add a limited undo request to the tool interface to recover from the last few operations carried out by a tool.

5.3 Call-by-value versus Call-by-reference

Background The concepts of call-by-reference and call-by-value are well-known in programming languages. They describe how an actual parameter value is transmitted from a procedure call to the body of the called procedure. In the case of call-by-reference, a *pointer* to the parameter is transmitted to the body. Call-by-reference is

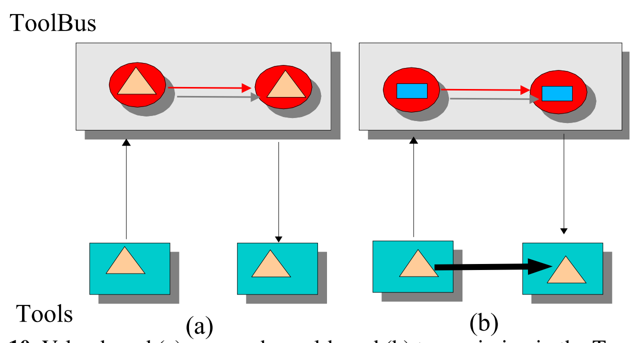

Figure 10. Value-based (a) versus channel-based (b) transmission in the TOOLBUS

efficient (only a pointer has to be transmitted) and the parameter value can be changed during execution of the procedure body (via the pointer). In the case of call-by-value, a *physical copy* of the parameter is transmitted to the procedure body. Call-by value is less efficient for large values and does not allow the called procedure to make changes to the parameter value in the calling procedure.

These considerations also apply to value transmissions in a distributed setting, with the added complication that values can be accessed or modified by more than one party. Call by reference (Figure 8) is efficient for infrequent access or update. It is the prevalent mechanism in, for instance, CORBA [17]. However, uncontrolled modifications by different parties can lead to disaster.

Call-by-value (Figure 9) is inefficient for large values and any sharing between calls is lost. To us, this is of particular interest, because we need to preserve sharing in huge parse trees. In the case of Java RMI [33], value transmission is achieved via serialization and works only for communication with other Java components. Using IIOP [29] communication with non-Java components is possible.

Current TOOLBUS *approach* Currently, the TOOLBUS provides a transport mechanism based on call-by-value as shown in Figure 10(a). It is transparent since the transmitted values are ATerms (see §1.2) that can be exchanged with components written in any language. Since pure values are exchanged, there is no need for distributed garbage collection.

Note that the call-by-reference model can easily be mimicked in the TOOLBUS. For instance, one tool can maintain a shared database and can communicate with other tools using record keys and field names so that only the values of record fields have to be exchanged (as opposed to complete records or even the complete database). In this way the access control to the shared database can be spelled out in detail and concurrency conflicts can be avoided. This solves one of the major disadvantages of the pure call-by-reference model in a distributed environment.

The downside is, however, that the TOOLBUS becomes a data bottleneck when huge values really have to be transmitted between tools. Currently, two workarounds are used. A first workaround is to get temporary relief by sending compressed values rather than the values themselves. A second workaround is to store the large value in the filesystem and to send a file name rather than the file itself. It does scale, but it also creates an additional inter-tool dependency and assumes that both tools have access to the same shared file system.

We will now first discuss how related frameworks handle call-by-reference and then we come back to implications for the TOOLBUS design. In particular, we will discuss channel-based transmission as already shown in Figure 10(b).

5.4 Related frameworks: Java RMI, RMI-IIOP and Java IDL

Given our needs and desires for a next generation TOOLBUS it is interesting to see what other solutions are applied in similar projects. In this section, we briefly look at three related mechanisms:

- Java Remote Method Invocation (RMI) which connects distributed objects written in Java;
- Java RMI over Internet Inter-ORB Protocol (IIOP) which is like RMI, but uses IIOP as the underlying protocol;
- Java IDL which connects Java implementations of CORBA interfaces.

Java RMI Java Remote Method Invocation is similar to the TOOLBUS architecture in the sense that it connects different tools, possibly running on different machines. It differs from the TOOLBUS setting because it is strictly Java based: only components written in Java can communicate via RMI.

For components to work together in RMI, first a *remote interface* is established. This is a Java interface that has a "real" implementation in the tool (or *server*) and a "stub" implementation on the client sides (Figure 11). The interface is written by the programmer as opposed to the generated interfaces in a TOOLBUS setting where they are derived from the communication patterns found in the TOOLBUS script. The stubs in the RMI setting are then generated from this Java interface using `rmic`: the RMI compiler. Stubs act as a client-side proxy, delegating the method call via the RMI system to the server object. In RMI, any object that implements a remote interface is called a *remote object*.

In RMI, arguments to or return values from remote methods can be primitive data (e.g. `int`), remote objects, or *serializable* objects. In Java, an object is said to be serializable if it implements the `java.util.Serializable` interface. Both primitive data and serializable objects are passed by value using Java's object serialization. Remote objects are essentially passed by reference. This means that changes to them are actually performed on the server, and updates become available to all clients. Only the behavior that was defined in the remote interface is available to the clients.

RMI programmers should be aware of the fact that any parameters, return values and exceptions that are not remote objects are passed by value. This makes it hard to understand when looking at a system of RMI objects exactly which method calls will result in a *local* (i.e. client side) state change, and which will have *global* (server side) effect.

Consider, again, our address book example. If the AddressBookService is implemented as a remote object in RMI, then client-side invocations of the `setAddress`

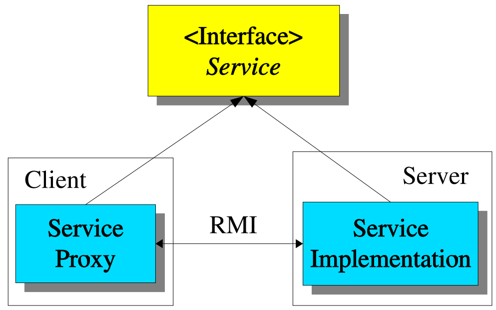

Figure 11. Client-server model in RMI framework.

method will cause a *global* update. If, on the other hand, the AddressBookEntries are made *serializable* and instances of this class are returned as the result of a query to the AddressBookService, then updates on these instances will have a *local* state change only.

Finally, before two RMI components can connect, the server side needs to register itself with an `rmiregistry`, after which the client needs to explicitly obtain a reference to the (remote) server object.

Java RMI over IIOP By making RMI programs conform to some restrictions, they can be made available over the Internet Inter-ORB Protocol (IIOP). This means that functionality offered by the RMI program can be made available to CORBA clients written in any (CORBA supported) language. The restrictions are mostly namespace oriented: programmers need to take special care not to use certain names that might collide with CORBA generated names, but some reservations should also be made regarding sharing preservation of object references. References to objects that are equal according to the == operator in one component, need not necessarily be equal in a remote component. Instead the `equals` method should be used to discern equality.

RMI over IIOP is best used when combining several Java tools for which the programmer would like to use RMI, and some tools written in another CORBA-supported language need to use (some of) the services provided by the Java tools. The component's interface is established by writing a Java interface, just as in plain RMI.

Java IDL Apart from Java RMI, which is optimized for connecting components that are all written in Java, there is also a connection from Java to CORBA using the Java Interface Definition Language (IDL). This alternative to Java RMI is for Java programmers who want to program in the Java programming language, based on interfaces defined in the CORBA Interface Definition Language.

Using this bridge, it becomes possible to let Java components communicate with CORBA objects written in any language that has Interface Definition Language (IDL) mappings.

Instead of writing a Java interface as is done in RMI, in Java IDL the definition is written in IDL: a special purpose interface language used as the base for CORBA

	TOOLBUS	RMI	RMI-IIOP	Java IDL
Architecture	Component coordination	Client Server	Client Client	Client Server
Interface	TSCRIPT	Java Interface	Java Interface	IDL
GC	yes	yes	no	no
parameters / return values	by-value	local: by-value remote: by-ref	local: by-value remote: by-ref	depends on signature
language	any with TB adapter	only Java	CORBA objects if interface in Java	any with IDL binding
component coordination	yes	no	no	no

Figure 12. Related architectures: a feature overview.

implementations. This IDL definition is then used to generate the necessary *stubs* (client side proxies to delegate method invocations to the server) and *skeletons*, *holder* and *helper* classes (server side classes that hide low-level CORBA details).

Feature summary Figure 12 shows some of the similarities and differences in TOOLBUS, RMI, RMI-IIOP and Java IDL.

- RMI, RMI-IIOP and Java IDL make an explicit distinction between *client* and *server* sides of a set of cooperating components. In the TOOLBUS setting all components are considered equal (and none are more equal than others).
- In RMI and RMI-IIOP, the programmer writes a Java interface which describes the component's incoming and outgoing method signature, from which stubs and skeletons are generated. In Java IDL a CORBA interface is written. In the TOOLBUS setting, these signatures are generated from the TOOLBUS script which describes much more of the component's behavior in terms of method call interaction, rather than just method signatures.
- The TOOLBUS takes care of garbage collection of the ATerms that are used to represent data as it is sent from one component to another. RMI allows programmers access to Java's Distributed Garbage Collection API. In RMI-IIOP and Java IDL however, this is not possible, because the underlying CORBA architecture is used, which does not support (distributed) GC, but places this burden entirely on the developer.
- In the TOOLBUS all data is sent by-value. RMI and RMI-IIOP use both pass-by-value and pass-by-reference, depending on whether the relevant data is serializable (it is a primitive type, or it implements `Serializable`) or is a remote object. In Java IDL the components abide by IDL prescribed interfaces. Determination of whether a parameter is to be passed by-value or by-reference is made by examination of the parameter's formal type (i.e. in the IDL signature of the method it is being passed to). If it is a CORBA *value type*, it is passed by-value. If it is an ordinary CORBA *interface type* (the "normal" case for all CORBA objects), it is passed by-reference.

– The TOOLBUS allows components in any language for which a TOOLBUS adapter exists. Programming languages such as C and Java are supported, but adapters also exist for a wide range of languages and applications, including e.g., Perl, Prolog, MySQL, Tcl and ASF+SDF. In RMI, only Java components can be connected; in RMI-IIOP the service is implemented in Java, its functionality (client-side) is available to CORBA clients. The Java IDL framework is fully CORBA compliant.
– Only the TOOLBUS has coordination support for component interaction. In the three other cases any undesired sequence of incoming and outgoing method calls will have to be prohibited by adding code to the component's internals. Whereas RMI, RMI-IIOP and Java IDL just perform the *wiring* that connects the components, the TOOLBUS also provides *workflow* support. In relation to this workflow support, it would be interesting to compare the TOOLBUS to related workflow description languages such as the Business Process Modeling language [16] and the Web Services Description Language [35].

Implications for the TOOLBUS *Approach* To overcome the problems of value-based transmission, we envisage the introduction of channels as sketched in Figure 10(b). This model is inspired by the second workaround mentioned at the end of §5.3 and is completely transparent for the user.

The idea is to stick to the strict call-by-value transmission model, but to implement the actual value transmission by data communication between sending tool and receiving tool thus offloading the TOOLBUS itself. Via the TOOLBUS, only an identification of the data value is transmitted between sender and receiver. The downside of this model is that it introduces the need for distributed garbage collection, since a value may be distributed to more than one receiving tool and the sender does not known when all receivers have retrieved their copy. Adding expiration times to values or reference counting at the TOOLBUS level may solve this problem.

6 Current Status

The current TOOLBUS was first specified in ASF+SDF and has then been implemented manually in C. Its primary target was the renovation of the ASF+SDF Meta-Environment.

The next generation TOOLBUS is being implemented in Java and aims at supporting larger applications such as, for instance, a multi-user game site like `www.gamesquare.nl` with thousands of users. High performance and recovery of crashed game clients are then of paramount importance. The Java implementation is organized in such a way that the actual implementation of tools is as much hidden as possible. This is achieved by introducing the interface `ToolInterface` that describes the required TOOLBUS/tool interaction. This interface can be implemented by a variety of classes:

`ClassicToolBusTool:` this implements the TOOLBUS/tool communication as used in current applications. The tool is executed as a separate, operating system level, process and the TOOLBUS/tool communication is achieved using sockets.

`JavaTool:` this implements a new model that addresses one of the issues mentioned in §5: when TOOLBUS and tool run on the same computer *and* the tool is written in Java, then the tool can be loaded dynamically in the executing TOOLBUS, e.g. using Java Threads. In this way, the overhead of interprocess communication can be eliminated.

`JavaRMITool:` this is a special case where a Java tool runs on another computer.

`SOAPTool:` this implements communication with a tool that has a SOAP interface.

A prototype implementation is under development that allows experimentation with the features mentioned in this paper.

7 Concluding Remarks

In this paper we have reflected on our experiences over the past years with the use of the TOOLBUS as a means to refactor a previously monolithic system: the ASF+SDF Meta Environment. This real test case of the TOOLBUS has taught us some of its shortcomings: its data bottleneck in case very large data items are sent using pass-by-value, maintenance issues related to undisciplined message passing and questions such as how to deal with exceptions caused by e.g. crashing tools.

Some of the ideas we showed in this paper could be implemented by changing or extending the TSCRIPT (e.g. to implement a call-reply regime as discussed in §5.1), others will also require extending the TOOLBUS and the tool-adapters (e.g. to detect crashed tools in combination with exception handling as discussed in §5.2).

We have also studied some related ideas and frameworks and we are now in a position where we have a new prototype of the TOOLBUS in Java, with a very open structure which allows for all sorts of experiments and case studies based on the experience we have with the existing TOOLBUS and the ideas presented in this paper.

Acknowledgments

We thank Pieter Olivier for his contribution and input to the many interesting and fruitful discussions we have had about TOOLBUS related issues, and his efforts to get `www.gamesquare.nl` TOOLBUS enabled.

References

1. H.C.N. Bakker and J.W.C. Koorn. Building an editor from existing components: an exercise in software re-use. Technical Report P9312, Programming Research Group, University of Amsterdam, 1993.
2. J. A. Bergstra and P. Klint. The discrete time TOOLBUS—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.
3. J. A. Bergstra, A. Ponse, and J. van Wamel. Process algebra with backtracking. In *REX School/Symposium*, pages 46–91, 1993.
4. J.A. Bergstra and P. Klint. The TOOLBUS: a component interconnection architecture. Technical Report P9408, University of Amsterdam, Programming Research Group, 1994.
5. J.A. Bergstra and P. Klint. The discrete time TOOLBUS. Technical Report P9502, University of Amsterdam, Programming Research Group, 1995.

6. J.A. Bergstra and P. Klint. The TOOLBUS coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88, 1996.
7. J.A. Bergstra and P. Klint. The discrete time TOOLBUS. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, 1996.
8. J.A. Bergstra and J.W. Klop. Process algebra: specification and verification in bisimulation semantics. In M. Hazewinkel and J.K. Lenstra andL.G.L.T. Meertens, editors, *Mathematics & Computer Science II*, volume 4 of *CWI Monograph*. North-Holland, 1986.
9. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the CAV 2001*, volume 2102 of *LNCS*, pages 250–254. Springer, July 2001.
10. M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
11. M. G. J. van den Brand, J. Heering, and P. Klint. Renovation of the ASF+SDF meta-environment - current state of affairs. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer-Verlag, 1997.
12. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.
13. M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. Environments for Term Rewriting Engines for Free. In *Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer Verlag, 2003. *To appear*.
14. M.G.J. van den Brand and C. Ringeissen. ASF+SDF parsing tools applied to ELAN. In Kokichi Futatsugi, editor, *Third International Workshop on Rewriting Logic and its Applications (WRLA'2000)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.
15. E. Brinksma. *On the Design of Extended LOTOS–A Specification Language for Open Distributed Systems*. PhD thesis, University Twente, 1988.
16. Business Process Management Initiative. *Business Process Modeling Language*, November 2002. `http://www.bpmi.org/bpmi-downloads/BPML1.0.zip`.
17. *The Common Object Request Broker: Architecture and Specification*, 1999. `http://www.omg.org/technology/documents/formal/corba_2.htm`.
18. D. Dams and J. F. Groote. Specification and Implementation of Components of a muCRL toolbox. Logic Group Preprint Series 152, Utrecht University, Dept. of Philosoph, 1995.
19. B. Diertens. New features in PSF I – Interrupts, Disrupts, and Priorities. Technical Report P9417, Programming Research Group, University of Amsterdam, 1994.
20. B. Diertens. Simulation and animation of process algebra specifications. Technical Report P9713, Programming Research Group, University of Amsterdam, 1997.
21. E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
22. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.
23. H.A. de Jong and P.A Olivier. Generation of abstract programming interfaces from syntax definitions. Technical Report SEN-R0212, St. Centrum voor Wiskunde en Informatica (CWI), August 2002. Submitted to Journal of Logic and Algebraic Programming.

24. P. Klint. *A Study in String Processing Languages*, volume 205 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
25. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.
26. B. Lisser and J. J. van Wamel. Specification of components in a proposition solver. Technical Report SEN-R9720, Centrum voor Wiskunde en Informatica (CWI), 1997.
27. S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.
28. P.D. Mosses. System demonstration: Action semantics tools. In M.G.J. van den Brand and R. Lämmel, editors, *Proceedings of the Second Workshop on Language Descriptions, Tools and Applications (LDTA 2002)*, volume 65.3 of *Electronic Notes in Theoretical Computer Science*, 2002.
29. Object Management Group. *CORBA IIOP Specification*, 2003. `www.omg.org/technology/documents/formal/corba_iiop.htm`.
30. P.A. Olivier. Embedded system simulation: testdriving the ToolBus. Technical Report P9601, University of Amsterdam, Programming Research Group, 1996.
31. P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, University of Amsterdam, 2000.
32. B. Randell. System structures for software fault tolerance. *IEEE Transactions on Software Engineering*, SE–1(3):21–232, June 1975.
33. Sun MicroSystems Inc. *Java Remote Method Specification*, 2003. `http://java.sun.com/j2se/1.4/docs/guide/rmi`.
34. S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. Control and data transfer in the distributed editor of the ASF+SDF meta-environment. Technical Report P9415, University of Amsterdam, Programming Research Group, 1994.
35. W3C: World Wide Web Consortium. *Web Services Description Language*, March 2001. `http://www.w3.org/TR/wsdl`.
36. A. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud, and I. Welch. Using coordinated atomic actions to design dependable distributed object systems. In *OOPSLA'97 Workshop on Dependable Distributed Object Systems*, 1997.