The ToolBus coordination architecture —a demonstration—

P. Klint[1,2] and P. Olivier[2]

[1] Department of Software Technology
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
[2] Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

Abstract. The ToolBus is a generic software architecture for building heterogeneous, distributed, systems. We demonstrate several applications that have been implemented using the ToolBus technology.

1 Introduction

We show a generic software architecture for building heterogeneous, distributed software systems. To get control over the possible interactions between software components ("tools") direct inter-tool communication is forbidden. Instead, all interactions are controlled by a "script" that formalizes all the desired interactions among tools. The result is a component interconnection architecture resembling a hardware communication bus, and therefore we call it the "ToolBus".

We describe the coordination of tools in process-oriented "**T** scripts" featuring, amongst others, (1) sequential composition, choice and iteration of processes; (2) handshaking (synchronous) communication of messages; (3) asynchronous communication of notes to an arbitrary number of processes; (4) subscription to notes; (5) dynamic process creation; (5) absolute and relative timeout and delay; (6) dynamic execution and termination of tools; (7) dynamic connection and disconnection of tools; (8) synchronous communication between ToolBus and tools; and (9) dynamic connection of monitoring tools.

Experience with the ToolBus coordination architecture and further references can be found in [BK96].

2 Motivation and background

The development of the ToolBus has primarily been motivated by the need to re-engineer the existing implementation of the Asf+Sdf Meta-environment [Kli93, vDHK96]: given a formal description of some language (e.g., programming language, query language, mark up language, or other application language) the Meta-Environment generates an interactive programming environment for it. The need to make connections with other, existing, software (e.g., user-interfaces, editors, proof checkers) as well as the flexibility we needed for further development of our system, were the driving forces for designing a generic coordination architecture.

It is our aim to use the ToolBus as the interconnection technology for connecting the various components involved in generated programming environments (such as parser generators, syntax-directed editors, rewrite rule compilers, compiled specifications, and the like). Our main expectations of this approach are therefore:

- By decomposing a single monolithic system into a number of cooperating components, the modularity and flexibility of the systems' implementation can be improved:
  - For each component we can select the optimal re-engineering strategy (e.g., adapt the old component, re-implement it using the best tools available, buy it, etc.)
  - The components can be combined in new ways that were previously inconceivable; rather than using a fixed skeleton for each programming environment generated, it becomes possible to tailor it towards specific needs.
  - The system will be very extensible because of its open nature.
- By connecting existing tools we can reuse their implementation and build new systems with lower costs. Typical examples we have encountered so far are the integration of existing tools for constructing user-interfaces and for constraint solving, as well as implementations of various programming languages.
- The distributed nature of the system immediately opens up possibilities for a parallel implementation.
- Formal methods can be used to create executable prototypes of the components. In a later stage, individual components can be replaced by more efficiently implemented versions. Note that the presence of a "correct" (formally developed) version of each component makes it possible to use some form of automated regression testing when developing a more efficient implementation.

3 Characteristics of the ToolBus

Data integration. How can tools exchange and share data structures representing application specific information? Instead of providing a general mechanism for representing the data in arbitrary applications, we will use a single, fixed, data representation based on term structures. We do not allow the exchange of arbitrary data structures, but insist that all data are represented in the same term format before they can be exchanged between tools. A consequence of this approach is that *existing* tools will have to be encapsulated by a small layer of software that acts as an "adapter" between the tool's internal data formats and conventions and those of the ToolBus.

Control integration. How can tools communicate or cooperate with each other? The control integration between tools is achieved by using process-oriented "**T** scripts" that model the possible interactions between tools. The major difference with other approaches is that we use one, formal, description of *all* tool interactions. Coordination and computation are strictly separated: inside the ToolBus a varying number of parallel processes takes care of the coordination while all actual computation is performed in tools (and not in the ToolBus itself). We uncouple the coordination activities inside the ToolBus by using pattern matching to establish communication between processes rather than using explicitly named communication ports. We support heterogeneity, since tools implemented in different languages running on different machines can be coordinated by way of a single ToolBus.

4 A demonstration

We will show the feasibility of our approach by demonstrating several applications that have been implemented using the ToolBus technology:

- A small (calculator-like) example in order to explain the basic principles and show (a) integration of tools written in different languages (C, Tcl); (b) distributed execution of such tools; (c) interactive monitoring of distributed ToolBus applications.
- A distributed auction featuring (a) a single auction master and a dynamically varying set of bidders all communicating with the auction via different work stations; (b) the use of time and time outs to control the bidding process.
- The use of algebraic specifications (using Asf+Sdf) to specify and prototype tools.
- A C development environment for embedded systems [Oli96].
- Parts of the Asf+Sdf Meta-environment that have been re-engineered by means of the ToolBus.

5 Other applications

The ToolBus has been used for building other applications as well:

- Various simulators (traffic light control, platform allocation).
- Connection of a constraint solver to a graphical editor.
- Various distributed games (including Go Moku and Battleship).
- Connecting different symbolic algebra packackes.
- A toolbox for checking and simulating μCRL specifications [DG95].

6 Current status of the implementation

The ToolBus has been implemented in ANSI C and runs on a variety of platforms (SUN/SOLARIS, SGI/IRIX, IBM PC/LINUX, IBM SP1/AIX).

Tools written in C, C++, Tcl, Python, Perl, Prolog and ASF+SDF can at this moment be connected to the TOOLBUS. In addition, arbitrary executables can be encapsulated in a standard fashion and be used as a tool.

In the current implementation, the **T** script is executed by an interpreter that runs as a single (operating system level) process. Tools are executed as separate processes and they are connected to the interpreter by means of inter-process communication (using TCP/IP). Tools may run on different computers.

7 Discussion

The work presented here stresses the importance of a strict separation between coordination and computation and shows that:

- Coordination can be based on a firm theoretical foundation (Process Algebra) that at the same time yields a feasible implementation strategy.
- Computation should be based on a language paradigm that is most suited for the application at hand. Rather than striving for the integration of many language paradigms into a single linguistic framework, we keep the various paradigms apart and only integrate tools implemented with different paradigms at the coordination level. As a special case, we are investigating the use of algebraic specifications and term rewriting for building tools.
- Formal methods can be used to create an executable prototype of each component. The efficiency of individual components can be increased incrementally by replacing them by faster implementations.

References

[BK96] J.A. Bergstra and P. Klint. The TOOLBUS coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.

[DG95] D. Dams and J.F. Groote. Specification and implementation of components of a μCRL toolbox. Technical Report 152, Department of Philosophy, University of Utrecht, 1995.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Oli96] P. Olivier. A simulator framework for embedded systems. In *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 436–439, 1996.

[vDHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping, An Algebraic Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996. To appear.

This article was processed using the LaTeX macro package with LLNCS style