Introduction to the ToolBus Coordination Architecture

Paul Klint
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
The problem: component interconnection

• Systems become **heterogeneous** because we want to couple existing and new software components
  – different implementation languages
  – different implementation platforms
  – different user-interfaces

• Systems become **distributed** in local area networks

• Needed: interoperability of heterogeneous systems
Component interconnection: reasons

- **Reusing** existing components decreases construction costs of new systems
- **Decomposing** large, monolithic systems into smaller, cooperating components increases
  - modularity
  - flexibility
Component interconnection: issues

- **Data integration**: exchange of data between components
- **Control integration**: flow of control between components
- **User-interface integration**: how do the user-interfaces of components cooperate?
Data integration

• Data representations differ per
  – **machine**: word size, byte order, floating point representation, ...
  – **language implementation**: size of integers, emulation of IEEE floating point standard, ...

• How can we exchange data between components:
  – integers, reals, record => linear encoding
  – pointers => impossible in general
Data integration

- Assume a common representation $R$
- For each component $C_i$ (with data domain $D_i$) there exist conversion functions
  
  - $f_i : D_i \rightarrow R$ and $f_i^{-1} : R \rightarrow D_i$
  
  - Convert a value $d_i$ from $C_i$ to $C_j$ by $f_j^{-1}(f_i(d_i))$

- Examples: IDDL, ASN-1, XML, ...

- ToolBus uses ATerms as common representation
Control integration

- **Broadcasting**: each component can notify other components of state changes
- **Remote procedure calls**: components can call each other as procedures
- **General message passing**: the most general approach
- **In the ToolBus Tscripts** are used to model the interactions between components
User-interface integration

- The ToolBus does not address user-interface integration as separate issue but can be used to achieve it
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Brief history of the ToolBus

- In 1992 the first implementation of the ASF+SDF Meta-Environment was completed:
  - 200 KLOC Lisp code
  - Monolithic
  - Hard to maintain

- ... all traits of a legacy system
Time line

- 1992: Unsuccessful decomposition experiments
- 1994: First generation: ToolBus
- 1995: Second generation: Discrete time ToolBus
- 2001: Meta-Environment based on ToolBus
- 2002/7: Extensions, new functions and structure
- 2007: Third generation: Java-based ToolBus
Introduction to the ToolBus Coordination Architecture

1992

Graphical Objects

Structure & text editor

Other parts

Old

Emacs

User-interface

Structure editor

Other parts

New
1993

- Difficult synchronization and communication problems start to appear
- PSF specification of communication; simulation reveals several deadlocks
- Problems with this specification:
  - complex (> 20 pages) and ad hoc
  - difficult to extend
  - cannot be used to directly coordinate the components
1993/1994

- Idea of a “ToolBus” as general communication structure appeared
- First design and implementation
- Several experiments
  - Feature interaction in telephone switches (RUU/PTT)
  - Traffic control (Nederland Haarlem/UvA/CWI/RUU)
  - Management of complex bus stations (idem)
  - Definition of user interfaces (UvA)
1994/1995

- Fall 1994: redesign based on this experience
- Spring 1995: design and implementation of Discrete Time ToolBus completed
- First experiments to prototype parts of the Meta-Environment started
More recently ...

- In 2001 a new implementation of the Meta-Environment based on the ToolBus was completed.
- In 2007 we have completed a new generation ToolBus (Java-based) that is used by the Meta-Environment.
- The ToolBus can be seen as a Service-oriented Architecture (SOA) avant la lettre ...
## Structuring and Composition of Software

<table>
<thead>
<tr>
<th>Statements</th>
<th>Procedures</th>
<th>Modules</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured programming</td>
<td>Functions, procedures &amp; libraries</td>
<td>Object-oriented programming &amp; Modules</td>
<td>Coordination languages &amp; SOA</td>
</tr>
<tr>
<td>Unix pipes</td>
<td>DCOM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Meta-Environment
Service-oriented Architecture (SOA)

- Loose coupling
- Service contract
- Autonomy
- Abstraction
- Reusability
- Composability
- Statelessness
- Discoverability

- Message exchange patterns
- Coordination
- Atomic transactions
ToolBus requirements

- Flexible interconnection architecture for software components
- Good control over communication
- Relatively simple descriptions
- Uniform data exchange format
- Multi-lingual: C, Java, Perl, ASF+SDF, ... 
- Potential for verification
- Use existing concurrency theory: Process Algebra
Process Algebra

- A theoretical framework to describe process behaviour
- Consists of
  - Constants: deadlock ($\delta$), silent step ($\tau$)
  - Atomic actions: a, b, c, ...
  - Processes x, y, z, ... built with the operators:
    - sequential compositions: .
    - non-deterministic choice: +
    - parallel composition: ||
Basic Process Algebra (BPA)
The basic axioms for choice (+) and sequential composition (·):

A1. \( x + y = y + x \)
A2. \( (x + y) + z = x + (y + z) \)
A3. \( x + x = x \)
A4. \( (x + y) \cdot z = x \cdot z + y \cdot z \)
A5. \( (x \cdot y) \cdot z = x \cdot y \cdot z \)

Axioms for deadlock:
A6. \( x + \delta = x \)
A7. \( \delta \cdot x = \delta \)
Merge ($\parallel$)

Use the auxiliary operator left merge ($\parallel_-$):

M1. $x \parallel y = x \parallel_- y + y \parallel_- x$
M2. $a \parallel_- x = a \cdot x$
M3. $a \cdot x \parallel_- y = a \cdot (x \parallel y)$
M4. $(x + y) \parallel_- z = x \parallel_- z + y \parallel_- z$

Examples:

$a \parallel b = a \parallel_- b + b \parallel_- a = a \cdot b + b \cdot a$

$a \cdot b \parallel c = a \cdot b \parallel_- c + c \parallel_- a \cdot b = a \cdot (b \parallel c) + c \cdot a \cdot b = a \cdot (b \cdot c + c \cdot b) + c \cdot a \cdot b$
Process Algebra versus ToolBus

- Process Algebra can be used to describe all (possibly infinite) behaviours of a collection of parallel processes.
- This behaviour has the form of a process tree still containing all possible choices.
- Properties of the parallel processes can be verified by verifying this behaviour description, e.g.
  - absence of deadlock.
Process Algebra versus ToolBus

- Atomic actions may be enabled/disabled as a result of conditions or time constraints.
- The ToolBus executes a process expression but randomly selects one of the enabled arguments of a choice operator.
- The steps taken by the ToolBus are thus just one possible series of steps that is contained in the complete behaviour of the process expression:
  - $a \parallel b$ executes as $a \cdot b$ or as $b \cdot a$ (and not both!)
Coordination, Representation & Computation

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

A rigorous separation of coordination from computation is the key to flexible and reusable systems
Architectural Layers

Coordination

Representation
Computations

Single Component

Cooperating Components
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Why not using XML as terms?

- Has been tried in various language processing projects
- XML is too verbose to represent parse trees of large (> 100 KLOC) programs
- XML does not provided sharing
- For discussion see: M.G.J. van Brand and P. Klint, ATerms for manipulation and exchange of structured data: It's all about sharing, *Information and Software Technology*, 49(1), 2007, 55-64.
Generic Representation
Annotated Terms (ATerms)

- Applicative, prefix terms
- Maximal subterm sharing ($\Rightarrow$ DAG)
  - cheap equality test, efficient rewriting
  - automatic generational garbage collection
- Annotations (text coordinates, dataflow info, ...)
- Very concise, binary, sharing preserving encoding
- Language & machine independent exchange format
ATerms
Term and Annotations

Annotations
A term is ...

- a Boolean, integer, real or string
  - true, 37, 3.14e-12, “rose”

- a value occurrence of a variable
  - X, InitialAmount, Highest-bid

- a result occurrence of a variable
  - X?, InitialAmount?
A term is ...

- a single identifier
  - f, pair, zero
- a function application
  - pair("rose", address("Street", 12345))
- a list
  - [a, b, c], [a, 1.25, "last"], [[a, 1], [b, 2]]
- a placeholder
  - <int>, add(<int>,<int>)
Matching of terms

• Term matching is used to
  – determine which actions can communicate
  – to transfer data between sender and receiver

• Intuition:
  – terms match if they are structurally identical
  – value occurrence: use variable's value
  – result occurrence: assign matched subterm to variable
    (only if overall match succeeds!)
Example of term matching

Before match

Context 1
X : 3
Y : 7

Context 2
Z : 17

Match

f(X,4,Y?,6) and f(3,Z?,5,6)

After successful match

Context 1
X : 3
Y : 5

Context 2
Z : 4
Types

- The ToolBus uses its own type system
  - static checks & automatic generation of interface code
- bool, int, real, str
- list: list with arbitrary elements
- list(Type): list with Type elements
  - list(int)
- term: arbitrary term
Types

- **Id**: all terms with function symbol *Id* (allows partial type declarations)
  - *f* accepts *f*, *f*(1), *f*("abc",3), ...

- **Id(T₁, ..., Tₙ)**
  - *f*(int, str) accepts *f*(3,"abc") but not *f*(3)

- **[T₁, ..., Tₙ]**: list of elements with given types
  - [int, str] accepts [1,"abc"] but not [1,2,3]
Types

• All variables have types
• Types are checked statically when possible
• Types play a role during matching:
  − I is int variable, S is str variable, T is term variable
  − match $f(13)$ and $f(\text{I?})$ succeeds
  − match $f(13)$ and $f(\text{S?})$ fails
  − match $f(13)$ and $f(\text{T?})$ succeeds
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

• The problem: component interconnection
• History & requirements
• Terms, types & matching
• The ToolBus architecture
• ToolBus scripts (Tscripts)
• Larger examples
• Implementation issues
• Conclusions
The ToolBus architecture

ToolBus

Tools

ATerms
common data exchange format

Coordination

Representation

Computation

Introduction to the ToolBus Coordination Architecture
The ToolBus architecture

- Processes inside the ToolBus can communicate with each other
- Tools can not communicate with each other
- Tools can communicate using a fixed protocol:
A typical scenario

UI and DB are completely decoupled

Configuration knowledge only in ToolBus script

UI and DB are completely decoupled

UI and DB are completely decoupled

User-interface

Database
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
ToolBus scripts: processes

- The ToolBus: a parallel composition of processes
- Private variables per process
- \[ P_1 + P_2 \quad P_1 \cdot P_2 \quad P_1 || P_2 \quad P_1 * P_2 \]
- :=, if then else
- All data are terms that can be matched
- A limited set of built-in operations on terms
- No other support for datatypes
ToolBus scripts: processes

- Send, receive **message** (handshaking)
- Send/receive **notes** (broadcasting)
- Subscription to notes
- Dynamic process creation
- Absolute/relative delay, timeout
ToolBus scripts: tools

- Execute/terminate tools
- Connect/disconnect tools
- Communication between process and tool is synchronous
- Process can send evaluation request to tool (which returns a value later on)
- Tool can generate events to be handled by the ToolBus
Define the initial processes in the application

The Tscript `hello.tb`

Define the process `HELLO`

It only prints a string

`process HELLO is printf("Hello world, my first Tscript!\n")`

`toolbus(HELLO)`

Start application with: `toolbus hello.tb`
Hello World: string generated by tool

process HELLO is
  let H : hello, •
  S : str •
  in
    execute(hello, H?). •
    snd-eval(H, get-text). •
    rec-value(H, text(S?)). •
    printf(S)
  endlet

H will represent the tool
S is a string variable

Execute hello,
H gets a tool id as value
Request a text from hello tool
Receive it,
S gets the value assigned

Definition of hello tool:
may be written in any language

tool hello is {command = "hello" } •
toolbus(HELLO)
Simple clock with user-interface

User can push a `showTime` button

Provides a `readTime` function

User can push a `showTime` button

User-interface

Clock

ToolBus

Tools

Introduction to the ToolBus Coordination Architecture
Simple clock with user-interface

process CLOCK is process-expression-1
tool clock is tool-definition-1

process UI is process-expression-2
tool ui is tool-definition-2

toolbus(CLOCK, UI)
process CLOCK is
  let Tid : clock, T : str
  in
    execute(clock, Tid?).
    ( rec-msg(showTime) .
      snd-eval(Tid, readTime) .
      rec-value(Tid, time(T?)) .
      snd-msg(showTime, T)
    ) * delta
   endlet

tool clock is { command = "clock" }
User-interface

process UI is
let Tid : ui, T : str
in
execute(ui, Tid?) .
  ( rec-event(Tid, button(showTime)) .
    snd-msg(showTime) .
    rec-msg(showTime, T?) .
    snd-do(Tid, displayTime(T)) .
    snd-ack-event(Tid, button(showTime))
  ) * delta
endlet

Receive event from ui tool
Get the time
Display it in ui tool
Processing of the event complete: send acknowledgment

tool ui is { command = "wish-adapter -script ui.tcl" }
Tscripts: in more detail

- Process communication: messages & notes
- Composite processes
- Expressions & built-in functions
- Time primitives
- Tools
Process communication: messages

- Messages used for synchronous, two-party communication between processes
- `snd-msg` and `rec-msg` synchronize sender/receiver
- Communication is possible if the arguments match
- There is two-way data transfer between sender and receiver (using result variables)
Process communication: notes

- Notes used for asynchronous, broadcasting communication between processes
- Each process must subscribe to the notes it wants to receive
- Each process has a private note queue on which snd-note, rec-note and no-note operate
Process communication: notes

- **subscribe** to notes of a given form
  - subscribe(compute(<str>,<int>))
- **unsubscribe** from certain notes
- **snd-note** to all subscribers
  - snd-note(compute(E,V))
- **rec-note**: receive a note of a given form
- **no-note** received of given form
Composite process expressions

- One of the atomic processes mentioned above
- $\delta$ (deadlock), $\tau$ (silent step)
- $P_1 + P_2$ : choice (non-deterministic)
- $P_1 \cdot P_2$ : sequential composition
- $P_1 || P_2$ : parallel composition
- $P_1^* P_2$ : repetition
Composite process expressions

- \( P(T_1, T_2, ...) \): a named process (with optional parameters) will be replaced by its definition
- \( \text{create}(P(T_1, T_2, ...), \text{Pid?}) \): dynamic process creation
- \( V := \text{Expr} \): evaluate \( \text{Expr} \) and assign result to \( V \)
- \( \text{if} \ \text{Expr} \ \text{then} \ P_1 \ \text{else} \ P_2 \ \text{fi} \)
- \( \text{if} \ \text{Expr} \ \text{then} \ P_1 \ \text{fi} = \text{if} \ \text{Expr} \ \text{then} \ P_1 \ \text{else} \ \text{delta} \ \text{fi} \)
Expressions

• An expression is evaluated in the current environment of the process in which it occurs
• Constants evaluate to themselves: \( a \)
• Variables evaluate to their current values
• Lists evaluate to a list of their evaluated elements
• Some function symbols have a built-in meaning
Built-in functions

- **Booleans**: not, and, or
- **Integers**: add, sub, mul, mod, less, less-equal, greater, greater-equal
- **Lists**: first, next, get, put, join, member, subset, diff, inter, size
- **Miscellaneous**: equal, not-equal, process-id, process-name, current-time, quote
Time primitives

- A (relative or absolute) delay or time out may be associated with each atomic process
- Relative time: \texttt{delay(Expr) or timeout(Expr)}
- Absolute time: \texttt{abs-delay(y, mon, d, h, min, s) or abs-timeout(y, mon, d, h, min, s)}
- Example:
  - \texttt{printf("expired") delay(10)}
  - \texttt{printf("Renew account") abs-timeout(2008,4,1,12,0,0)}
Process definitions

- **Process definition**: `process Pname Formals is P`
- **Formals** are optional and contain a list of formal parameter names
  - `process MakeWave(N : int) is ...`
- All variables (including formals) must be declared and have a type
- `let VarDecls in P endlet` introduces variables:
  - `let E : str, V : int in ... endlet`
Tools

- Tools have to be executed or connected before they can be used
- Requires a tool definition: `tool ui is { ... }`
- Introduces a new type, e.g. `ui`
- Execute a tool: `execute(ui, Uid?)`
- Receive connection request: `rec-connect(ui, Uid?)`
- Tool identification is assigned to `Uid` (of type `uid`)
Tools

- **snd-terminate**: terminate an executing tool
  - `snd-terminate(Tid)`
- **rec-disconnect**: receive disconnection request from tool
  - `rec-disconnect(Uid)`
- **shutdown**: terminate the whole ToolBus
  - `shutdown("Auction ends")`
Tools

- **snd-eval, rec-value**: request tool to evaluate a term, and receive the resulting value from tool
  - initiative: ToolBus

- **snd-do**: request tool to perform some action, there is no reply
  - initiative: ToolBus

- **rec-event, snd-ack-event**: receive event from tool, acknowledge it after appropriate processing
  - initiative: tool
Tscripts

- A Tscripts consists of
  - a list of process and tool definitions
  - a single ToolBus configuration
- A ToolBus configuration describes the initial set of active processes in the ToolBus:
  - toolbus\((Pname_1, \ldots, Pname_n)\)
  - Each \(Pname\) is optionally followed by parameters
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples: calculator; auction; waves
- Implementation issues
- Conclusions
Example: calculator
Example: calculator

- **CALC**: the calculation process
- **BATCH**: reads expressions from file, calculates their value, writes result back to file
- **UI**: the user-interface
- **LOG** maintains a log of all calculations
- **CLOCK** provides current time
**Process CALC**

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

tool calc is { command = "calc"}
```
Process \textbf{BATCH}

\begin{verbatim}
process BATCH is
  let Tid : batch, E : str, V : int
  in
    execute(batch, Tid?).
    (
      snd-eval(Tid, fromFile) . rec-value(Tid, expr(E?)).
      snd-msg(compute, E) . rec-msg(compute, E, V?).
      snd-do(Tid, toFile(E, V))
    ) * delta
  endlet

tool batch is \{command = "batch"\}
\end{verbatim}
User-interface

- When the user presses Calc, a dialog window appears to enter an expression
- The result is displayed in a separate window
- Pressing showLog displays all calculations so far
- Pressing showTime displays the current time
- Pressing Quit ends the application
User-interface: process $\text{UI}$

```plaintext
process UI is
  let Tid : ui
  in
    execute(ui, Tid?) .
    ( CALC-BUTTON(Tid) + LOG-BUTTON(Tid))\* delta
    ||
    TIME-BUTTON(Tid) * delta
    ||
    QUIT-BUTTON(Tid)
  endlet

tool ui is { command = wish-adapter -script calc.tcl" }
```

Calc and Log button are exclusive

Time and Quit button are independent
process CALC-BUTTON(Tid : ui) is
  let N : int, E : str, V : term
  in
    rec-event(Tid, N?, button(calc)) .
    snd-eval(Tid, get-expr-dialog) .
    ( rec-value(Tid, cancel) .
      + rec-value(Tid, expr(E?)) .
        snd-msg(compute, E) .
        rec-msg(compute, E, V?) .
          snd-do(Tid, display-value(V))
    ) . snd-ack-event(Tid, N)
User-interface: **LOG-BUTTON**

```plaintext
process LOG-BUTTON(Tid : ui) is
  let N : int, L : term
  in
    rec-event(Tid, N?, button(showLog)) .
    snd-msg(showLog) .
    rec-msg(showLog, L?) .
    snd-do(Tid, display-log(L)) .
    snd-ack-event(Tid, N)
  endlet
```
User-interface: **TIME-BUTTON**

```plaintext
process TIME-BUTTON(Tid : ui) is
    let N : int, T : str
    in
    rec-event(Tid, N?, button(showTime)) .
    snd-msg(showTime) .
    rec-msg(showTime, T?) .
    snd-do(Tid, display-time(T)) .
    snd-ack-event(Tid, N)
endlet
```

```plaintext
process QUIT-BUTTON(Tid : ui) is
    rec-event(Tid, button(quit)) .
    shutdown("End of calc demo")
```
process LOG is
  let Tid: log, E: str, V: term, L: term
  in subscribe(compute(<str>, <term>)) .
    execute(log, Tid?).
    ( rec-note(compute(E?, V?)) .
      snd-do(Tid, writeLog(E, V))
    + rec-msg(showLog) .
      snd-eval(Tid, readLog) .
      rec-value(Tid, history(L?)) .
      snd-msg(showLog, history(L))
    ) * delta
endlet
Process **LOG1**

```
process LOG1 is
    let TheLog : list, E : str, V : term
    in
        subscribe(compute(<str>, <term>)) .
        TheLog := [] .
        ( rec-note(compute(E?, V?)) .
            TheLog := join(TheLog, [[E, V]])
            +
            rec-msg(showLog) .
            snd-msg(showLog, TheLog)
        ) * delta
    endlet
```

Alternative definition of logger: maintain the log in a list
Process **CLOCK**

```plaintext
process CLOCK is
  let Tid : clock, T : str
  in
    execute(clock, Tid?).
    ( rec-msg(showTime).
      snd-eval(Tid, readTime).
      rec-value(Tid, time(T?)).
      snd-msg(showTime, T)
    ) * delta
endlet
```
ToolBus Configuration

```
ToolBus Configuration

toolbus (CALC, BATCH, UI, LOG, CLOCK)

Creates the processes for the calculator application

Start calculator application:
  toolbus calc.tb
```
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples: calculator; auction; waves
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples: calculator; auction; waves
- Implementation issues
- Conclusions
Example: distributed auction
Example: distributed auction

- How are bids synchronized?
- How to inform bidders about higher bids?
- How to decide when the bidding is over and the item is sold?
- Bidders may come and go during the auction
Example: distributed auction
Example: distributed auction

• The *Auction* process
  – executes *master* tool: user-interface of auction master
  – connection/disconnection of new bidders
  – introduces new items for sale (at the initiative of the auction master)
  – controls the bidding process via *OneSale*

• A *Bidder* process is created for each new bidder
Process Auction

process Auction is
let Mid : master, Bid : bidder in
execute(master, Mid?).
(ConnectBidder(Mid, Bid?) +
OneSale(Mid)) *
rec-event(Mid, quit).
shutdown("Auction is closed").
endlet

Execute the master tool

Repeat:
● add new bidder between sales, or
● perform one sale

Until:
● auction master quits

Close the auction application

tool master is { command = "wish-adapter -script master.tcl" }
Process `ConnectBidder`

```
process ConnectBidder(Mid : master, Bid : bidder?)
is
  let Pid : int, Name : str
  in
  rec-connect(Bid?) .
  create(Bidder(Bid), Pid?) .
  snd-eval(Bid, get-name) .
  rec-value(Bid, name(Name?)) .
  snd-do(Mid, new-bidder(Bid, Name))
endlet
```

- Receive a connection request from a new bidder tool
- Create a new Bidder process
- Ask bidder for its name
- Send name to master tool
Process **OneSale**

process OneSale(Mid : master) is


let Descr : str,                %% Description of current item for sale
    InAmount : int,            %% Initial amount for item
    Amount : int,                %% Current amount
    HighestBid : int,           %% Highest bid so far
    Final : bool,                 %% Did we already issue a final call for bids?
    Sold : bool,                   %% Is the item sold?
    Bid : bidder                  %% New bidder tool connected during sale

in rec-event(Mid, new-item(Descr?, InAmount?)) .

    HighestBid := InAmount .
    snd-note(new-item(Descr, InAmount)) .

    ( ... ) * if Sold then snd-ack-event(Mid, new-item(Descr, InAmount)) fi
endlet

Where the action is ...
Process \textbf{OneSale}

\begin{align*}
&( \text{if not}(\text{Sold}) \text{ then } \ldots \text{ fi} \\
+ &\text{if not}(\text{or}(\text{Final}, \text{ Sold})) \text{ then } \ldots \text{ fi} \\
+ &\text{if and}(\text{Final}, \text{ not}(\text{Sold})) \text{ then } \ldots \text{ fi} \\
+ &\text{ConnectBidder}(\text{Mid}, \text{ Bid?}) \ldots \\
)* &\text{if Sold then } \ldots \text{ fi}
\end{align*}
Process OneSale

if not(Sold) then
  rec-msg(bid(Bid?, Amount?)) .
  snd-do(Mid, new-bid(Bid, Amount)) .
  if less-equal(Amount, HighestBid) then
    snd-msg(Bid, rejected) .
  else
    HighestBid := Amount .
    snd-msg(Bid, accepted) .
    snd-note(update-bid(Amount)) .
    snd-do(Mid, update-highest-bid(Bid, Amount)) .
    Final := false
  fi
fi

+ if not(or(Final, Sold)) then ... fi
+ if and(Final, not(Sold)) then ... fi
+ ConnectBidder(Mid, Bid?) ...
)
* if Sold then ... fi
Introduction to the ToolBus Coordination Architecture

**Process OneSale**

- **Not yet sold, not asked for final bids ...**
  - Wait 10 sec, then ask for final bids
  - Inform auction master
  - Yes, now we have asked for final bids

- **Not yet sold, but asked for final bids ...**
  - Wait 10 sec, then inform all bidders that item is sold

- **Yes, item is now sold**

- **Bidder is connected during sale**
  - Inform new bidder about progress

- **Yes, item is now sold**

- **Restart, final bids (if any)**
Introduction to the ToolBus Coordination Architecture

**Process Bidder**

```plaintext
process Bidder(Bid : bidder) is
  let Descr : str, Amount : int, Acceptance : term
  in
    subscribe(new-item(<str>, <int>)) . subscribe(update-bid(<int>)) .
    subscribe(sold(<int>)) . subscribe(any-higher-bid).

  ( ... )
  * delta
endlet
```
Get info about item for sale after connection

**Process Bidder**

\[
\begin{align*}
& ( \text{rec-msg(Bid, new-item(Descr?, Amount?))} \\
& \quad + \text{rec-note(new-item(Descr?, Amount?))} \\
& \quad + \text{rec-disconnect(Bid) . delta} ) . \\
& \text{snd-do(Bid, new-item(Descr, Amount))} . \\
& ( \text{rec-event(Bid, bid(Amount?))} . \\
& \quad \text{snd-msg(bid(Bid, Amount))} . \text{rec-msg(Bid, Acceptance?) .} \\
& \quad \text{snd-do(Bid, accept(Acceptance))} . \text{snd-ack-event(Bid, bid(Amount))} \\
& \quad + \text{rec-note(update-bid(Amount?))} . \text{snd-do(Bid, update-bid(Amount))} \\
& \quad + \text{rec-note(any-higher-bid)} . \text{snd-do(Bid, any-higher-bid)} \\
& \quad + \text{rec-disconnect(Bid) . delta} ) * \\
& \text{rec-note(sold(Amount?))} . \text{snd-do(Bid, sold(Amount))} ) * \\
& \text{delta}
\end{align*}
\]

- **Same, but normal case**
- **Disconnect between sales**
- **Inform bidder tool**
- bidder comes with new bid
- **Inform bidder**
- **Disconnect during sale**
- **End of this sale sale**
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples: calculator; auction; waves
- Implementation issues
- Conclusions
Road map

• The problem: component interconnection
• History & requirements
• Terms, types & matching
• The ToolBus architecture
• ToolBus scripts (Tscripts)
• Larger examples: calculator; auction; waves
• Implementation issues
• Conclusions
One-dimensional wave equation

Simulate a string attached at the two end points:
One-dimensional wave equation

Amplitude at point $i$ at $t+\Delta t$ is given by:

$$y_i(t+\Delta t) = F(y_i(t),y_i(t-\Delta t),y_{i-1}(t),y_{i+1}(t))$$

and

$$F(z_1,z_2,z_3,z_4) = 2z_1 - z_2 + \left(c \frac{\Delta t}{\Delta x}\right)^2 (z_3 - 2z_1 + z_4)$$

$\Delta x$: the (small) interval between sampling points

c: constant representing the propagation velocity of the wave
Example: wave equation
One-dimensional wave equation

- Auxiliary process $F$ computes function $F$
- Process $P$ models a sampling point
- Process $\text{Pend}$ models the end points
- Process $\text{MakeWave}$ constructs $N$ connected instances of $P$ and two end points
- Tool $\text{display}$ visualizes the simulation
Process $F$

Compute $F(z_1, z_2, z_3, z_4) = 2z_1 - z_2 + (c \Delta t/\Delta x)^2 (z_3 - 2z_1 + z_4)$

```
process F(Z1 : real, Z2 : real, Z3 : real, Z4 : real, Res : real?) is
  let CdTdX2 : real
    in
      CdTdX2 := 0.01 .
      Res := radd(rsub(rmul(2.0, Z1), Z2),
                   rmul(CdTdX2,
                        radd(rsub(Z3, rmul(2.0, Z1)), Z4))))
  endlet
```

Arbitrary value for $(c \Delta t/\Delta x)^2$

- $2z_1 - z_2^+$
- $(c \Delta t/\Delta x)^2 (*)$
- $(z_3 - 2z_1 + z_4)$
**Process** $P$

```
process $P$(Tid : display, L : int, I : int, R : int, Dstart : real, Estart : real) is
  let AL : real, AR : real, D : real, D1 : real, E : real
  in
    D := Dstart . E := Estart .
    (( rec-msg(L, I, AL?)
      || rec-msg(R, I, AR?)
      || snd-msg(I, L, E)
      || snd-msg(I, R, E)
      || snd-do(Tid, update(I, E))
    ) .
    D1 := E .
    F(E, D, AL, AR, E?) .
    D := D1
  ) * delta
endlet
```

- $L$: left, $I$: this point, $R$: right
- $D$, $E$: amplitudes of this point
- Receive amplitudes of neighbours
- Send our amplitude to neighbours
- Update our amplitude on display
- Compute new versions of $D$ and $E$
Process Pend

process Pend(Tid : display, I : int, NB : int) is
  let W : real
  in
    ( rec-msg(NB, I, W?) || snd-msg(I, NB, 0.0) ||
      snd-do(Tid, update(I, 0.0))
    ) * delta
endlet
Process **MakeWave**

```
process MakeWave(N : int) is
  let Tid : display, Id : int, I : int, L : int, R : int
  in
    execute(display, Tid?) .
    snd-do(Tid, mk-wave(N)) .
    create(Pend(Tid, 0, 1), Id?) .
    L := sub(N,1) .
    create(Pend(Tid, N, L), Id?) .
    I := 1 .
    if less(I, N) then
      L := sub(I, 1) . R := add(I, 1) .
      create(P(Tid, L, I, R, 1.0, 1.0), Id?) .
      I := add(I, 1)
    fi *
    shutdown("end") delay(sec(60))
endlet
```
Tool definition and ToolBus configuration

tool display is { command = "wish-adapter -script ui-wave.tcl"}

toolbus(MakeWave(8))
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples: calculator; auction; waves
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- ...
- Implementation issues
  - Overview of the ATerm API
  - Brief overview of the ToolBus implementation
  - Writing ToolBus tools
  - Using ToolBus adapters
- Conclusions
Road map

- The problem: component interconnection
- ...
- Implementation issues
  - Overview of the ATerm API
  - Brief overview of the ToolBus implementation
  - Writing ToolBus tools
  - Using ToolBus adapters
- Conclusions
Requirements ATerms

- Open: independent of hw/sw platform
- Simple: a small API
- Efficient: fast reading and writing
- Concise: small memory usage
- Language-independent
- Annotations: applications can transparently store additional information in data structure
ATerm Types

- INT
- REAL
- APPL
- LIST
- PLACEHOLDER
- BLOB (Binary Large OBject)
- ANNOTATION
Examples

- 1 3.14 -0.7E34
- f(a,b) "test!"(1, 2.1, "hello")
- [] [1, 2, "abc"]
- <int> f(<int>, <real>)

- **BLOBs**
  - used to encode images, binary files, ...
  - have no textual representation
The ATerm Implementation

- C and Java API
- Only applicative operations
  - No destructive operations on ATerms
- Maximal subterm sharing
- Automatic garbage collection
- Binary encoding (BAF: Binary ATerm Format)
The ATerm C API

- Level 1: 41 functions
- Level 2: 80 functions (superset of Level 1)
- All function start with `AT`
- Defines types `ATerm` and `ATbool`
- Make and Match
- Read and Write
- Annotate
Intermezzo: Patterns

• A pattern is an ATerm with placeholders:
  \texttt{incr(<int>)}

• A string pattern is a pattern represented as string:
  "incr(<int>)"

• A string pattern resembles the format string in \texttt{printf/scanf} in C

• Placeholders correspond to typed arguments of \texttt{ATmake/ATmatch}
Make and Match

- **ATerm ATmake(String p, ATerm a1, ...)**
  - parse p and fill placeholders with a1, a2, ...

- **ATerm ATmatch(ATerm t, String p, ATerm *a1, ...)**
  - match t against p; assign subterms at placeholders to a1, a2,...

- **ATbool ATisEqual(ATerm t1, ATerm t2)**

- **int ATgetType(ATerm t)**
Read and Write

- ATerm ATreadFromString(String s)
- ATerm ATreadFromTextFile(File f)
- ATerm ATreadFromBinaryFile(File f)
- ATbool ATwriteToTextFile(ATerm t, File f)
- ATbool ATwriteToBinaryFile(ATerm t, File f)
- char *ATwriteToString(ATerm t)
Annotate

- ATerm ATsetAnnotation(ATerm t, ATerm l, ATerm a)
  - add annotation [l, a] to copy of t
- ATerm ATgetAnnotation(ATerm t, ATerm l)
- ATerm ATremoveAnnotation(ATerm t, ATerm l)
Other Functions in the Level 1 API

• Variations on the preceding functions
• ATprintf
• handlers (warnings and errors)
• protect/unprotect
Structure of an ATerm-based Application

#include <stdio.h>
#include <aterm1.h>

int main(int argc, char * argv[])
{
    ATerm bottomOfStack;
    ATinit(argc,argv,&bottomOfStack);
    foo();
    return 0;
}
The Level 2 API

- Detailed operations for efficient ATerm manipulation
- Dictionaries
- Tables
- Indexed sets
The Java API

• Two versions:
  – Native (uses the C version via JNI, not implemented)
  – Pure (a pure Java reimplementation)

• Interface `ATermFactory` encapsulates the whole API

• Separate interfaces for each kind of `ATerm` (`AFun`, `ATermList`, etc.)
Class Structure

- **is-a relation**
- **part-of relation**

![Class Structure Diagram]

- **ATermFactory**
  - **implemented-by**
  - **NativeFactory**

- **ATerm**
  - **type**
  - **next**
  - **first**

- **ATermPlaceholder**
  - **type**

- **ATermInt**

- **ATermReal**

- **ATermAppl**
  - **args**

- **ATermList**
Using ATermFactory

import aterm.*
factory = new PureFactory();
ATerm t1 = factory.makeInt(3)
ATerm t2 = factory.readFile("test.trm");
ATerm t3 = factory.makeAFun("f1", 1, false);
ATerm t4 = factory.make("f(<int>)", 3);
ATerm t5 = factory.parse("f(1, [a, b])");
Road map

• The problem: component interconnection
• ...

• Implementation issues
  – Overview of the ATerm API
  – Brief overview of the ToolBus implementation
  – Writing ToolBus tools
  – Using ToolBus adapters

• Conclusions
Road map

• The problem: component interconnection
• ...

• Implementation issues
  – Overview of the ATerm API
  – Brief overview of the ToolBus implementation
  – Writing ToolBus tools
  – Using ToolBus adapters

• Conclusions
ToolBus design/implementation method

- Specification of ToolBus using ASF+SDF
- Execution of small test cases
  - Tool behaviour is defined very abstractly
- Hand translation of ASF+SDF specification to C
  - Literal translation of Tscripts
  - Implementation of tools is more concrete (see later)
- Very few bugs in ToolBus implementation
  - Some bugs turned out to be bugs in the specification!
The ToolBus implementation

The ToolBus is implemented as a Unix process that interprets Tscripts.

ToolBus/Tool communication is implemented using TCP/IP sockets.

Tools are implemented as separate Unix processes.
The ToolBus Interpreter

- Syntax analysis of Tscript (lex/yacc)
- Typechecking of Tscript
- Create the initial ToolBus configuration
- Start execution
- Delays and timeouts
- Garbage collection of terms
The ToolBus Interpreter

- Execute tools as separate Unix process
- On creation: send expected input signature to tool
  - Permits detection of Tscript/tool mismatches
- During execution of tool: check terms received from tool against their output signature
  - Permits detection of misbehaving tools
- Enforce ToolBus protocol for each tool
Main Interpreter Loop

- Wait for
  - an event coming from one of the tools
  - expiration of a timer
- Compute effect of event/timer on ToolBus state
- Perform any enabled atomic actions
- Repeat as long as possible
- Go back to waiting state
ToolBus Interpreter

- Interpreter maintains a list of processes.
- Each process is compiled into a finite automaton with an action associated with each transition:
  - From the enabled actions, one is selected randomly and executed.
  - The process goes to the corresponding next state.
- A `select` system call waits for I/O on any socket or expiration of timer.
Introduction to the ToolBus Coordination Architecture

ToolBus/tool connection

Tool wants to connect to ToolBus
Well-know socket of ToolBus at fixed address
Tool, connects to well-know socket and sends tool name and host name
Tool receives socket address and tool id

New socket is created
Communication starts
Implementation considerations

- Terms are linearized before sending and parsed when receiving them.
- There is a separate transport layer that provides byte level messages of given length (to avoid system dependent segmentation of the byte stream).
Road map

- The problem: component interconnection
- ...
- Implementation issues
  - Overview of the ATerm API
  - Brief overview of the ToolBus implementation
  - Writing ToolBus tools
  - Using ToolBus adapters
- Conclusions
Road map

- The problem: component interconnection
- ...
- Implementation issues
  - Overview of the ATerm API
  - Brief overview of the ToolBus implementation
  - Writing ToolBus tools
  - Using ToolBus adapters
- Conclusions
Recall the Hello World script

```plaintext
process HELLO is printf("Hello world, my first Tscript!\n")
toolbus(HELLO)
```
Hello World: string generated by tool

process HELLO is
  let H : hello,
    S : str
  in
    execute(hello, H?) .
    snd-eval(H, get-text) .
    rec-value(H, text(S?)) .
    printf(S)
  endlet

tool hello is {command = "hello"}
toolbus(HELLO)

How can we implement this tool?
First version of a hello tool (C)

```c
#include <stdio.h>
#include <aterm1.h>
#include <atb-tool.h>

ATerm hello_handler(int conn, ATerm inp) {...}

int main(int argc, char *argv[]) {
    ATerm bottomOfStack;
    ATBinit(argc, argv, &bottomOfStack);
    if(ATBconnect(NULL, NULL, -1, hello_handler) >= 0){
        ATBeventloop();
    } else {
        fprintf(stderr, "hello: Could not connect to the ToolBus, giving up!\n");
        return -1;
    }
    return 0;
}
```

- **Level 1 interface of ATerms**
- **ToolBus primitives**
- **Interrupt handler**
- **Initialize application**
- **Connect to ToolBus**
- **Start event loop**
- **Give up when connection fails**
hello_handler

ATerm hello_handler(int conn, ATerm inp)
{
    ATerm arg, isig, osig;

    if(ATmatch(inp, "rec-eval(get-text)"))
        return ATmake("snd-value(text("Hello World, my first ToolBus tool in C!\n\n"))");
    if(ATmatch(inp, "rec-terminate(<term>)", &arg))
        exit(0);
    if(ATmatch(inp, "rec-do(signature(<term>,<term>))", &isig, &osig)){
        return NULL;
    }
    Aterror("hello: wrong input %t received\n", inp);
    return NULL;
}
Observations

- Tool consists of main and an event handler
- All processing is routed via the event handler
- Event handler does repetitive (and error prone) decoding of requests using $ATmatch$
- Event handler takes care of standard messages for termination, signature handling, etc.
- Why not automate some of these tasks?
Automatic generation of tool interfaces

- **Script.tb**
  - `toolbus -gentifs Script.tb`
  - Language independent tool interfaces (tifs) for all tools
  - Interface for other languages
  - Tscript for the application

- **Script.tifs**
  - `tifstoc -tool N Script.tifs`
  - Generate C interface for tool N
  - `tifstojava`

The C interface for tool N
Second version of hello tool

#include "hello.tif.c"

ATerm get_text(int conn)
{
    return ATmake("snd-value(text("Hello World, my first ToolBus tool in C!\n"))");
}

void rec_terminate(int conn, ATerm msg)
{
    exit(0);
}

int main(int argc, char *argv[])
{
    ... as before ...
}
Generated file `hello.tif.c`

```c
#include "hello.tif.h"
#define NR_SIG_ENTRIES 2

static char *signature[NR_SIG_ENTRIES] = {
    "rec-eval(<hello>,get-text)",
    "rec-terminate(<hello>,<term>)",
};

ATerm hello_checker(int conn, ATerm siglist)
{
    return ATBcheckSignature(siglist, signature,
                               NR_SIG_ENTRIES);
}
```

Prototypes of generated C functions

The signature of this tool

Checker for input signature
Generated file hello.tif.c

```c
ATerm hello_handler(int conn, ATerm term)
{
    ATerm in, out, t0;

    if(ATmatch(term, "rec-eval(get-text)")) {
        return get_text(conn);
    }
    if(ATmatch(term, "rec-terminate(<term>)", &t0)) {
        rec_terminate(conn, t0);
        return NULL;
    }
    ...
}
```

Call user-defined function `get_text`

Call user-defined function `rec_terminate`
Generated file hello.tif.c

```c
ATerm hello_handler(int conn, ATerm term)
{
    ... 
    if(ATmatch(term, "rec-do(signature(<term>,<term>))", &in, &out)) {
        ATerm result = hello_checker(conn, in);
        if(!ATmatch(result, "[]"))
            ATfprintf(stderr, "warning: not in input signature:\n\t%t\n\tl\n", result);
        return NULL;
    }

    AError("tool hello cannot handle term %t", term);
    return NULL; /* Silence the compiler */
}
```
A larger example: the calc tool

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

tool calc is { command = "calc"}
A larger example: the \texttt{calc} tool

#include <stdlib.h>
#include "calc.tif.c"

ATerm expr(int conn, char *s) { ... }  
void rec_terminate(int conn, ATerm t) { ... }  
int calculate(ATerm t) { ... }  

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;
    ATBinit(argc, argv, &bottomOfStack);
    if(ATBconnect(NULL, NULL, -1, calc_handler) >= 0)
    {  
        ATBeventloop();
    } else
    {
        fprintf(stderr, "calc: Could not connect to the ToolBus, giving up!\n");
        return 0;
    }
}
A larger example: the **calc** tool

```c
ATerm expr(int conn, char *s)
{ ATerm trm = ATmake(s);
   if(!trm)
      return ATmake("snd-value(calc-error(<str>))", s);
   else
      return ATmake("snd-value(val(<int>))", calculate(trm));
}
void rec_terminate(int conn, ATerm t)
{ exit(0);
}
```

- Try to convert argument string to term
- Calculate its value
- Send that value back to the ToolBus
- Handle termination
A larger example: the **calc** tool

```c
int calculate(ATerm t) {
    int n; char *s; ATerm t1, t2;

    if(ATmatch(t, "<int>", &n))
        return n;
    else if(ATmatch(t, "<str>", &s))
        return atoi(s);
    else if(ATmatch(t, "plus(<term>,<term>)", &t1, &t2))
        return calculate(t1) + calculate(t2);
    else if(ATmatch(t, "times(<term>,<term>)", &t1, &t2))
        return calculate(t1) * calculate(t2);
    else {
        Aterror("panic in calculate: %s", t);
        return 0;
    }
}
```

Recursive evaluation of the expression
Road map

• The problem: component interconnection
• ...

• Implementation issues
  – Overview of the ATerm API
  – Brief overview of the ToolBus implementation
  – Writing ToolBus tools
  – Using ToolBus adapters

• Conclusions
Road map

- The problem: component interconnection
- ...
- Implementation issues
  - Overview of the ATerm API
  - Brief overview of the ToolBus implementation
  - Writing ToolBus tools
  - Using ToolBus adapters
- Conclusions
ToolBus Adapters

- Needed to adjust existing programs/libraries to the ToolBus

Separate program:

Library:

Standard input/output are redirected by adapter
A selection of adapters

- **wish-adapter**: execute Tcl/Tk windowing shell
- **tcltk-adapter**: ditto but uses the Tcl/Tk library†
- **java-adapter**: java program as tool
- **perl-adapter**: perl program as tool†
- **python-adapter**: python program as tool†
- **gen-adapter**: arbitrary Unix command as tool†

† = not yet supported in Java-based ToolBus
The wish-adapter

- Execute Tcl/Tk's windowing shell as a tool
- **Ex.** `wish-adapter -script calculator.tcl`
  - `-script`: The Tcl script to be executed
  - `-script-args`: Arguments for the Tcl script
- The command `wish` is executed once and all further requests are directed to this instance of `wish`
The wish-adapter

- **snd-eval**(Tid, Fun(A₁,...,Aₙ)): perform the Tcl function call Fun A₁ ... Aₙ

- **rec-value**(Tid, Res?): return value for previous eval request

- **rec-event**(Tid, A₁,...,Aₙ): event generated by wish

- **snd-ack-event**(Tid, A₁): ack previous event

- **snd-terminate**(Tid, A₁): terminate wish-adapter
The *gen-adapter*

- Execute arbitrary Unix command as tool
- **Example:** `gen-adapter -cmd ls -l`
- `snd-eval(Tid, cmd(Cmd, input(Str))): execute the Unix command *Cmd* with *Str* as standard input`
- `rec-value(Tid, output(Res)): receive the standard output *Res* from a previous command`
- `snd-terminate(Tid, Arg): terminate execution of gen-adapter`
Road map

- The problem: component interconnection
- ...
- Implementation issues
  - Overview of the ATerm API
  - Brief overview of the ToolBus implementation
  - Writing ToolBus tools
  - Using ToolBus adapters
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Road map

- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
- Implementation issues
- Conclusions
Conclusions

- ToolBus is an effective technology for coordination and composition of tools
- ToolBus fits in the popular model of service-oriented architectures
- ToolBus enables incremental software renovation
A Legacy System

A complete blackbox:
• Subsystems unknown
• Subsystem dependencies unknown
Analyze and decompose in major subsystems
Replace dependencies by Tscript

Tscript represents control dependencies between subsystems

ToolBus

Adapter for component

Unmodified legacy code
Separate renovation strategy per subsystem

Replace by standard package

Contains important business logic: implement from scratch

Keep legacy code
Introduction to the ToolBus Coordination Architecture
Further reading

- See at http://www.meta-environment.org (Documentation menu entry):
  - Guide to ToolBus Programming
  - The ATerm Programming Guide
  - Further references can be found in Bibliography