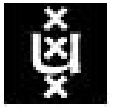
# Introduction to the ToolBus Coordination Architecture

Paul Klint

UNIVERSITEIT VAN AMSTERDAM

# Road map

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

# Road map

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

# 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 heterogenous 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, ...
- 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

- There is a general trend towards multi-threaded user-interfaces (concurrency)
- There is a general trend to use formal techniques to specify user-interfaces
- 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

# Road map

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

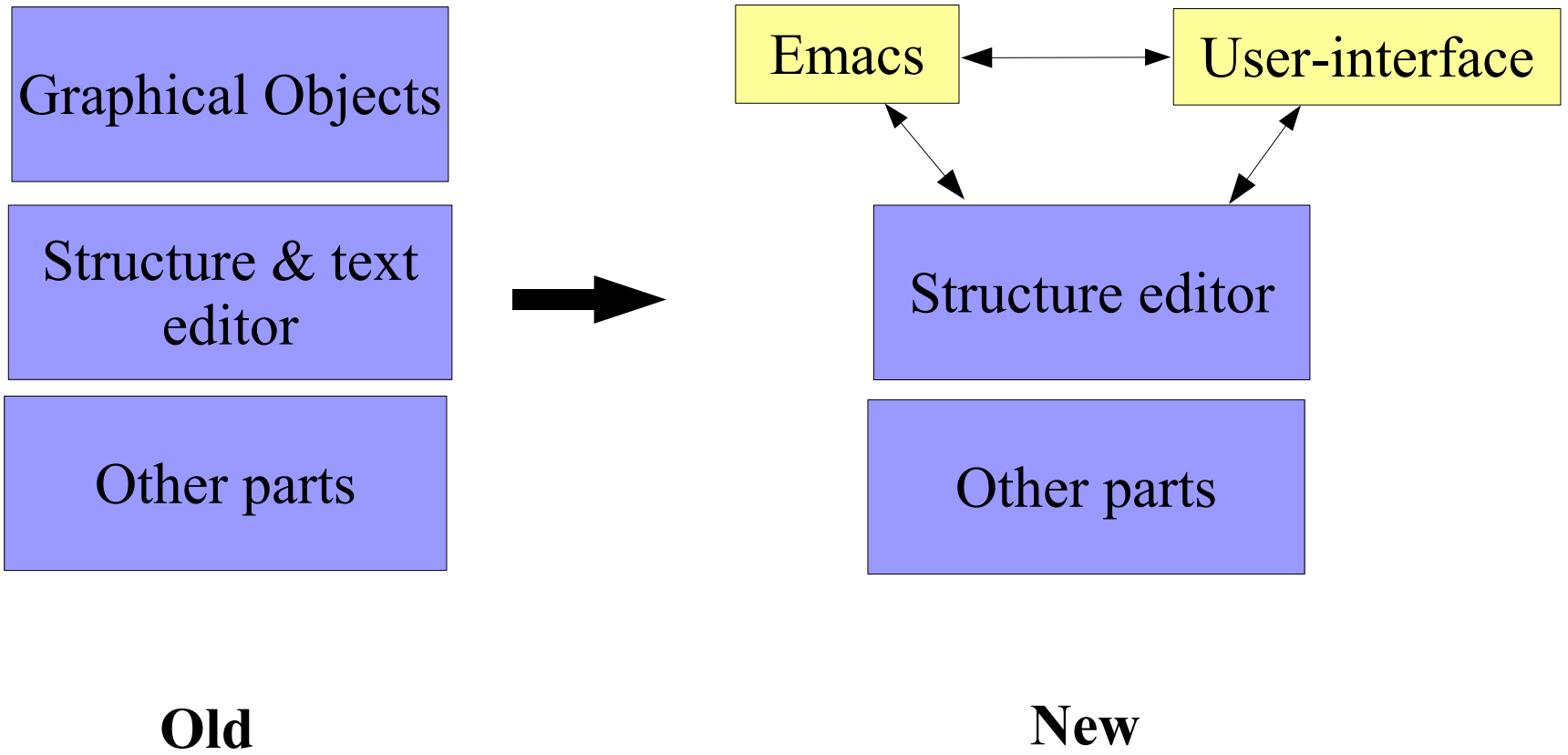
# Brief history of the ToolBus

- Around 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 ToolBus
- 1995: Discrete time ToolBus
- 2001: Meta-Environment restructured
- 2002/3: preparing for the next generation ...

# 1992


Graphical Objects
Structure & text editor
Other parts
Emacs
User-interface
Structure editor
Other parts
Old
New

# 1993

- Difficult synchronization and communication problems 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
- End 2002 we have started on a new generation ToolBus based on these experiences ...

# ToolBus requirements

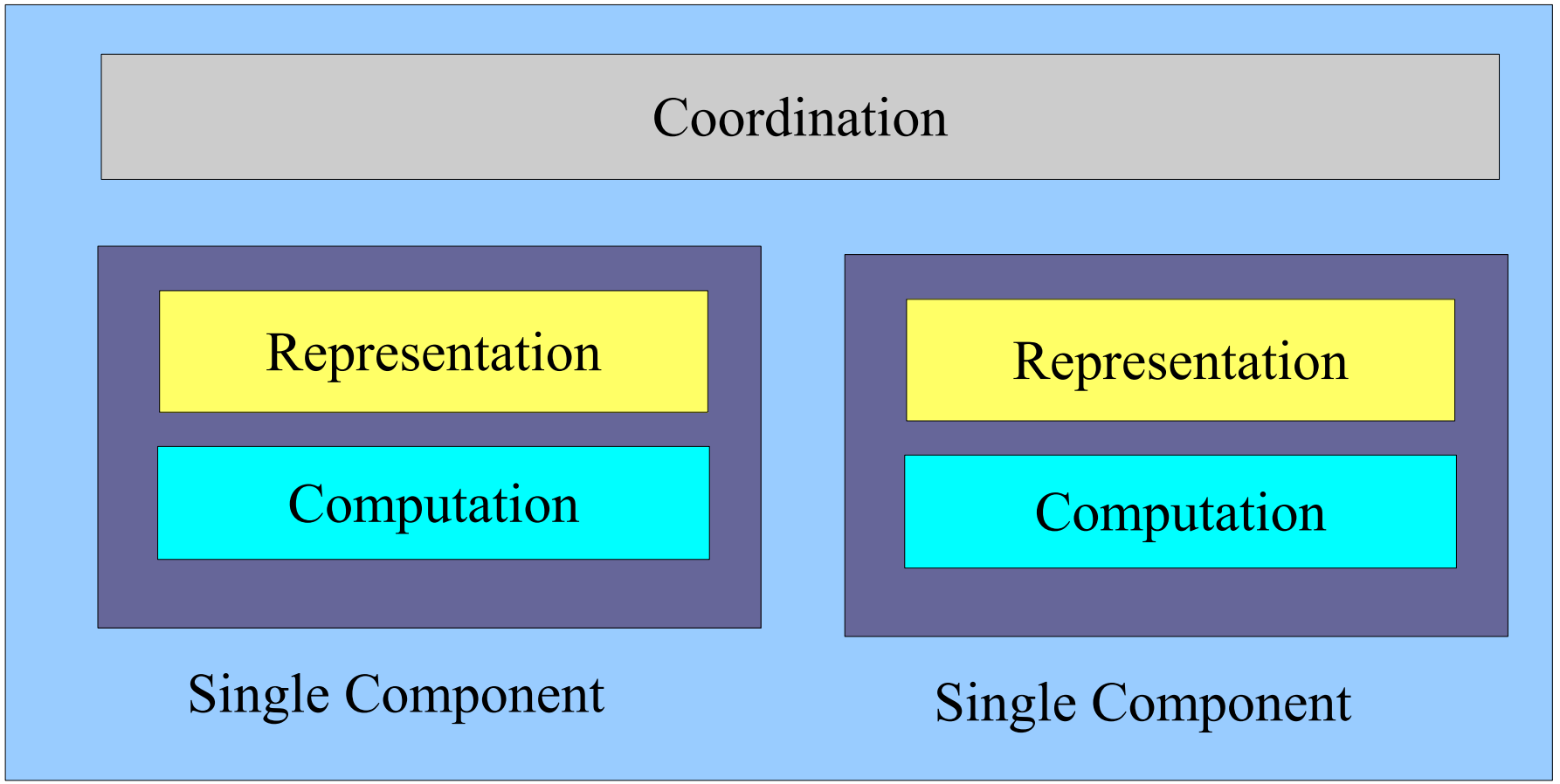
- Flexible interconnection architecture for software components
- Good control over communication
- Relatively simple descriptions
- Uniform data exchange format
- Multi-lingual: C, Java, Perl, ASF+SDF, ...
- Potential for verification
- Use existing concurrency theory

# 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




**Cooperating Components**

# Road map

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

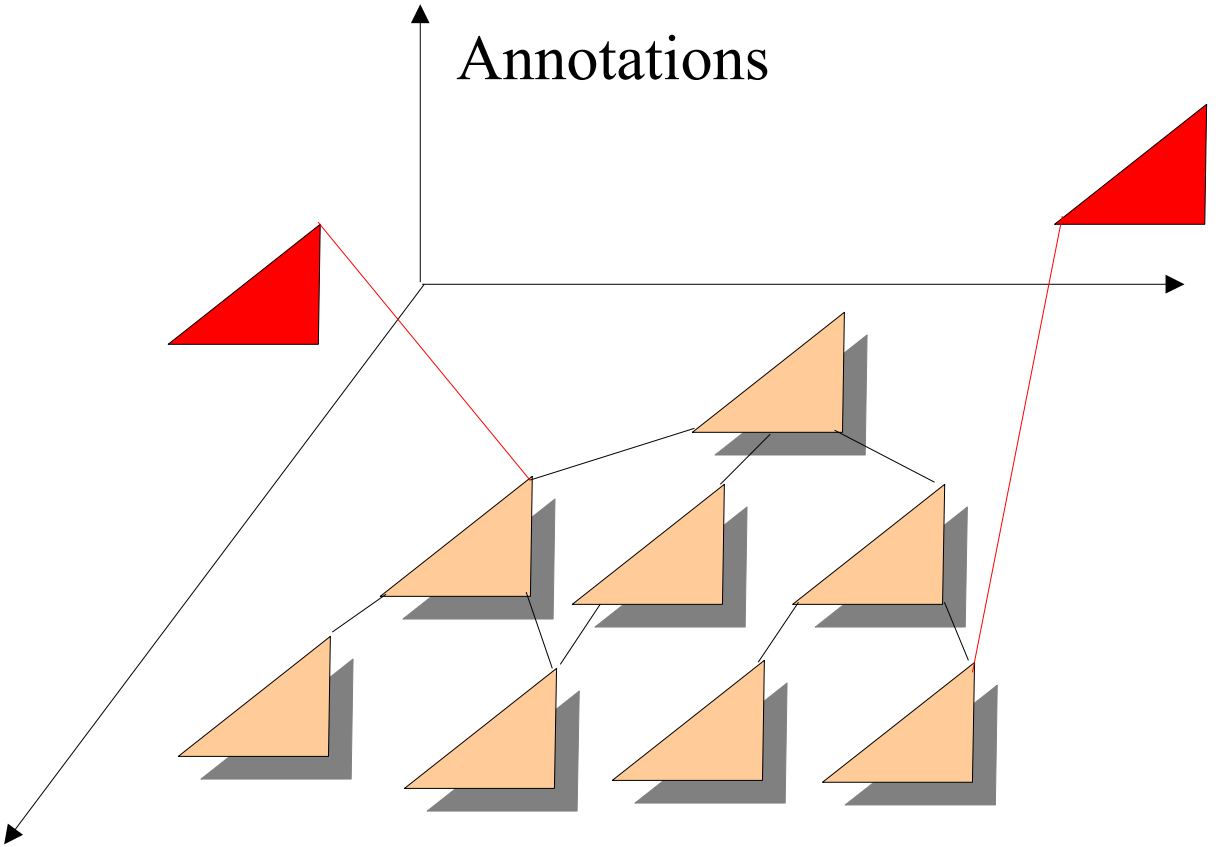
# Road map

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

# 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



# 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?

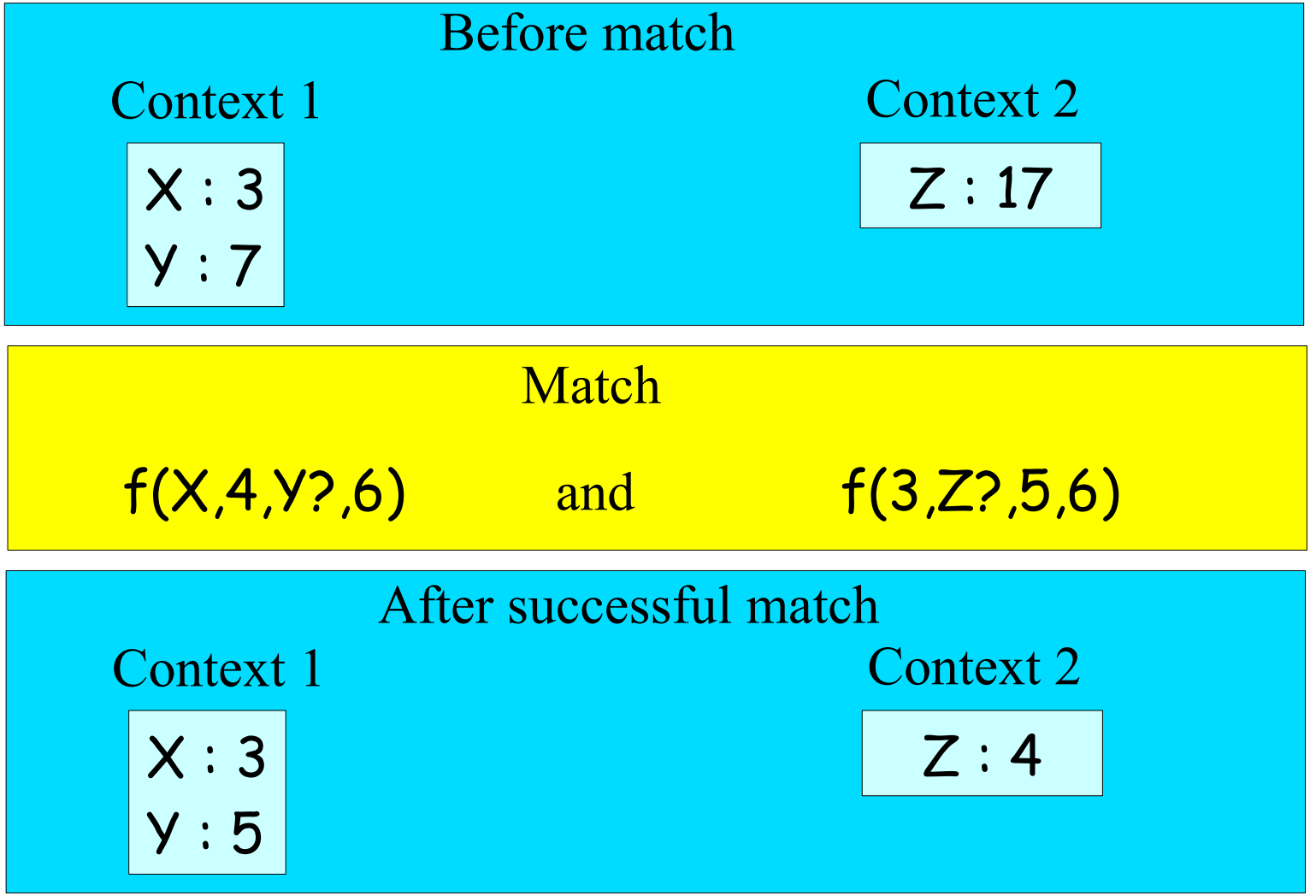
# A term is ...

- 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 the 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



# 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)`, ...
- $Id(T_1, ..., T_n)$
  - `f(int, str)` accepts `f(3,"abc")` but not `f(3)`
- $[T_1, ..., T_n]$ : 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(I?)` succeeds
  - match `f(13)` and `f(S?)` fails
  - match `f(13)` and `f(T?)` succeeds

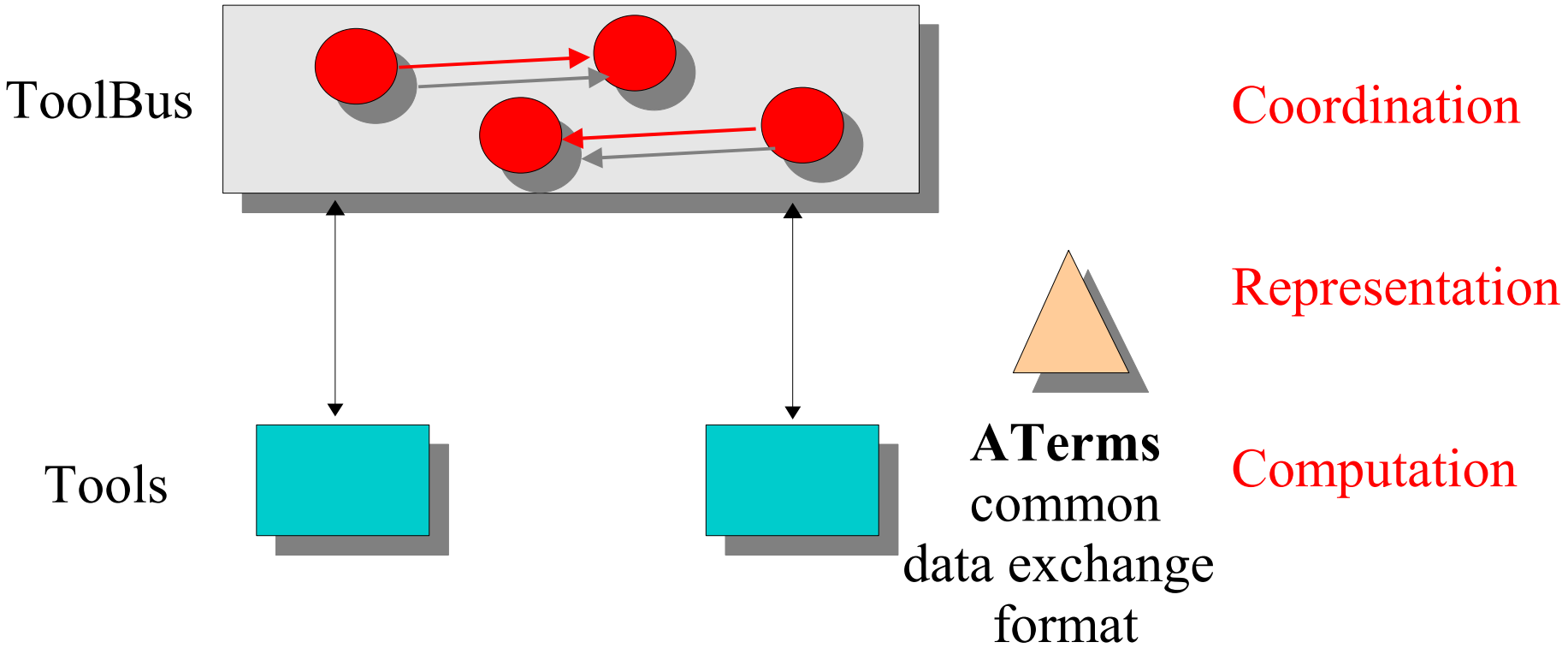
# Road map

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

# Road map

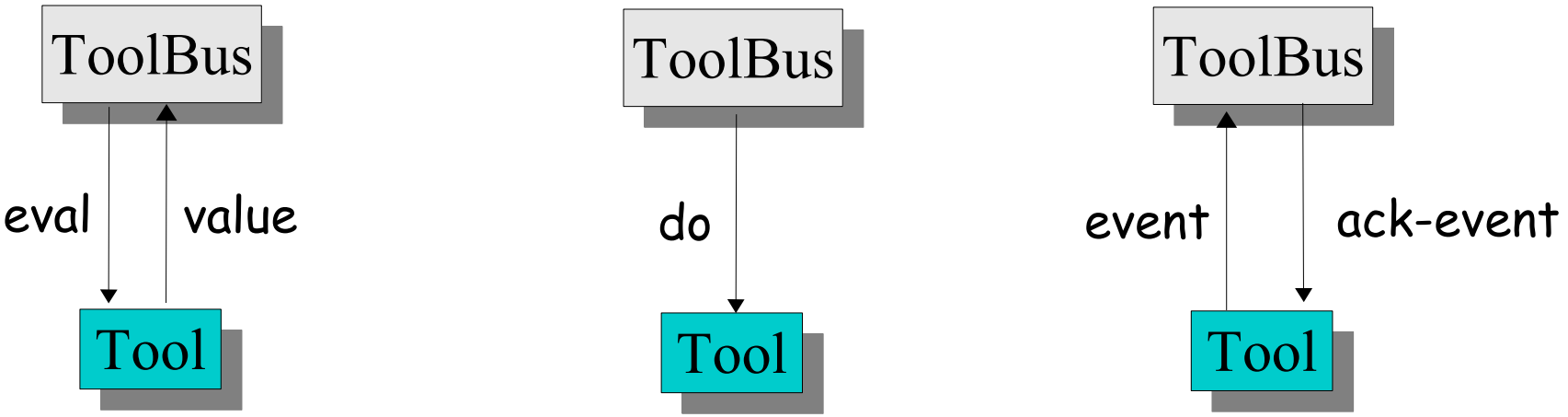
- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples

# The ToolBus architecture

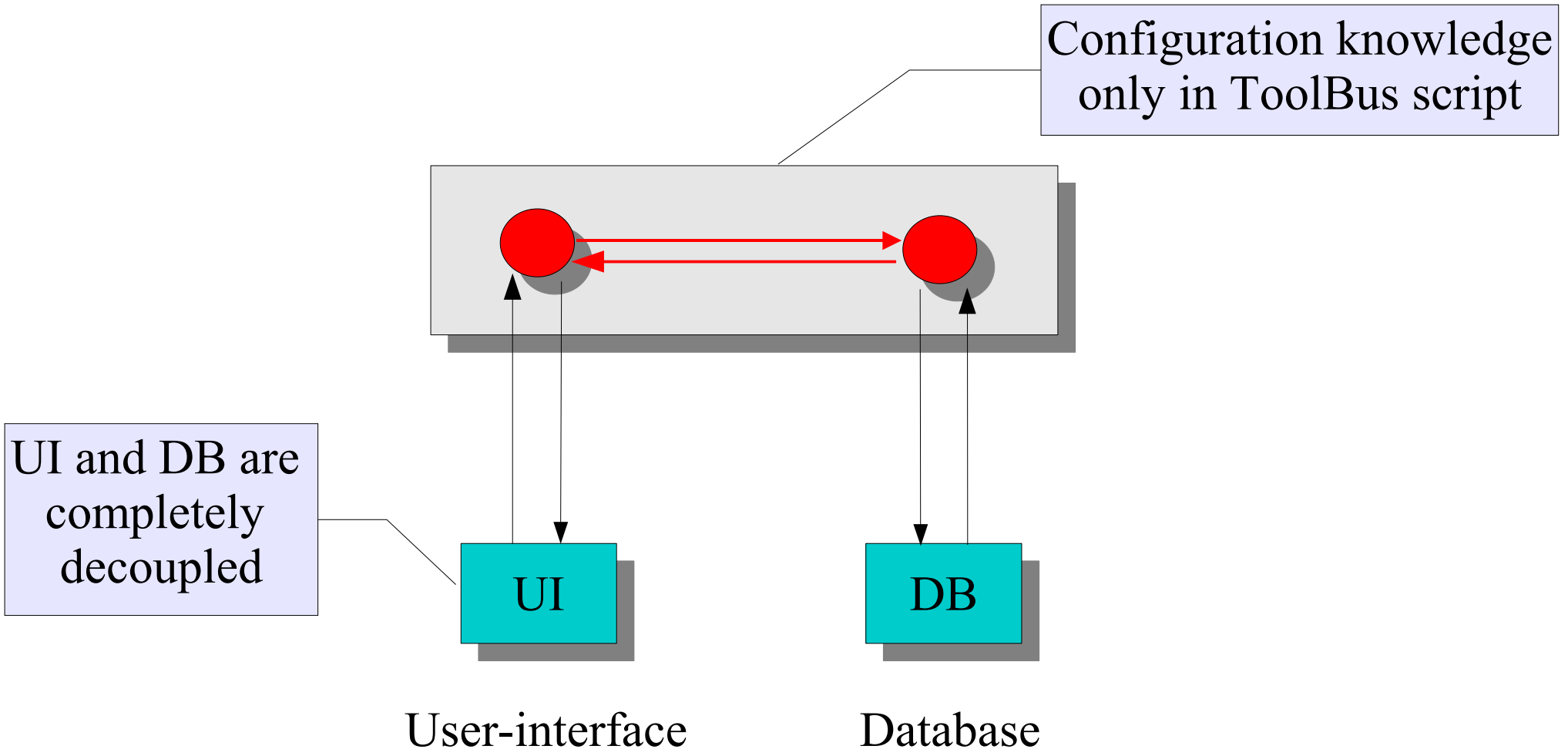


# The ToolBus architecture

- 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



# Road map

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

# Road map

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

# ToolBus scripts: processes

- The ToolBus: a parallel composition of processes
- Private variables per process
- $P_1 + P_2$ $P_1 . P_2$ $P_1 || P_2$ $P_1 * P_2$
- :=, 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

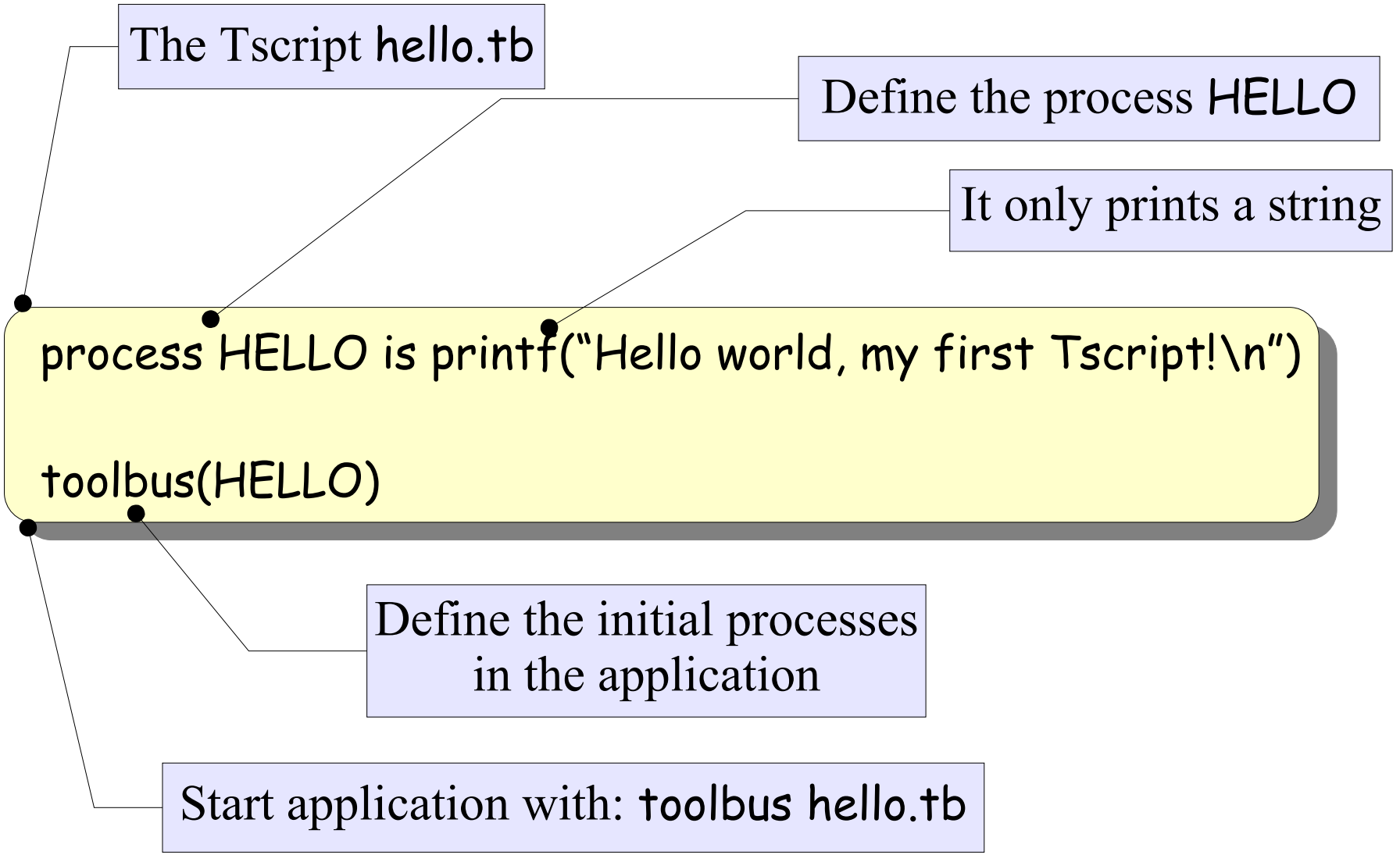
# ToolBus scripts: processes

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

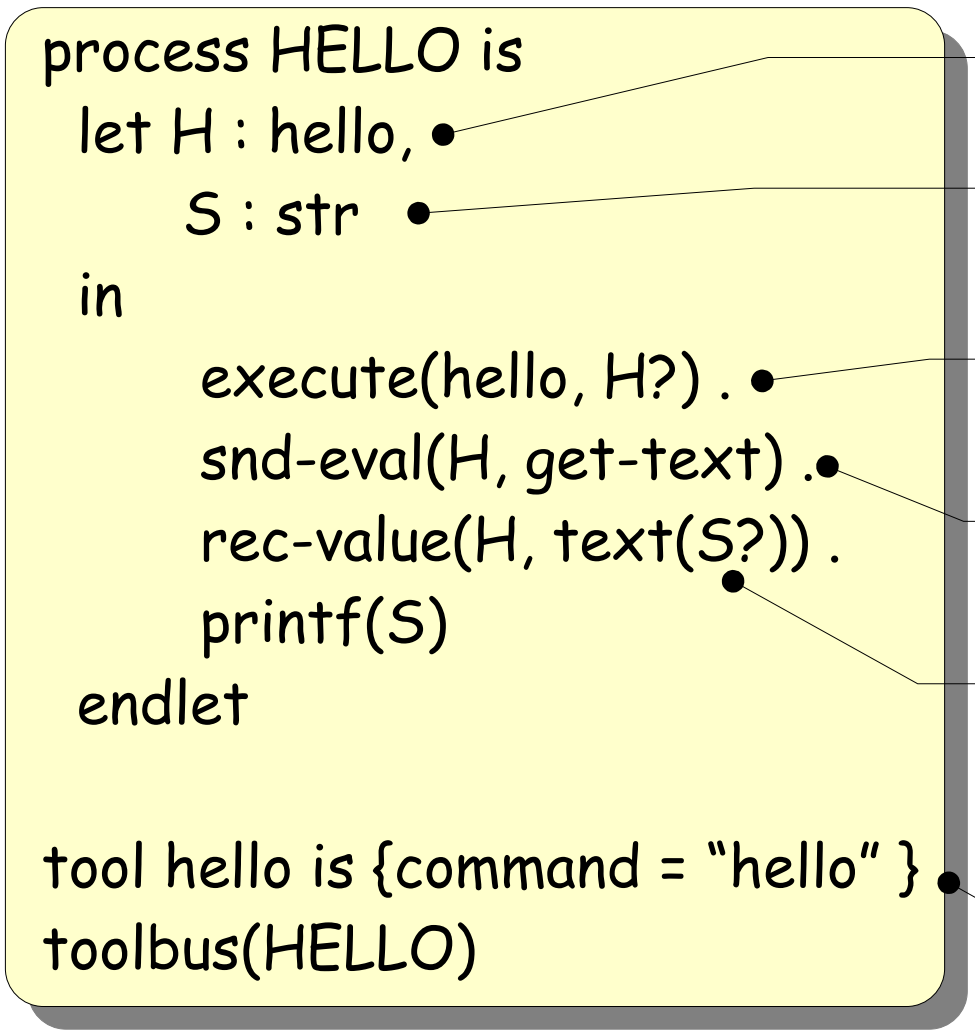
# ToolBus scripts: tools

- 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

# Hello World


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)
Define the initial processes in the application
Start application with: toolbus hello.tb

# Hello World: string generated by tool




H will represent the tool

S is a string variable

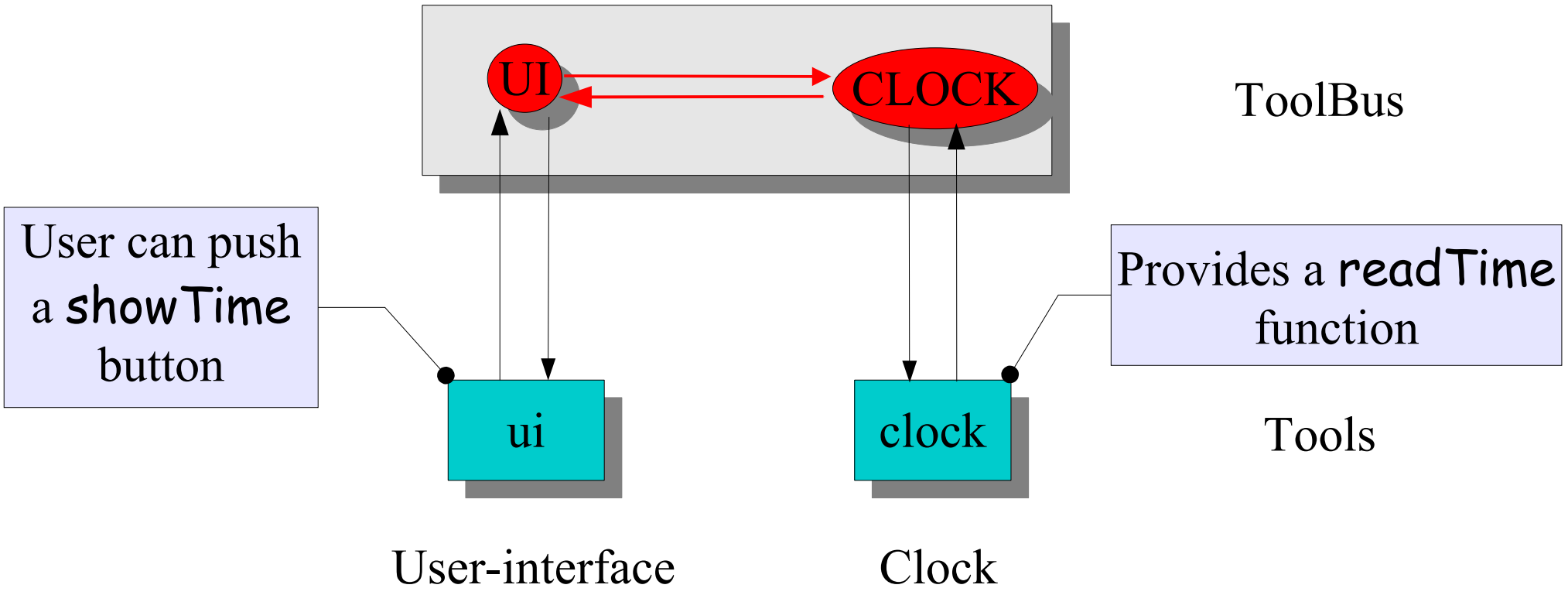
Execute hello,
H gets a tool id as value

Request a text from hello tool

Receive it,
S gets the value assigned

Definition of hello tool:
may be written in any language

# Simple clock with user-interface



# 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)
```

# Clock

```
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
```

Receive a message from another process

Get time from clock tool

Reply to the message

( ... ) * delta is an endless loop

```
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 acknowledgement

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

# Tscripts: revisited

- 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)
- $P_1$. $P_2$: 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
- create(P($T_1$, $T_2$, ...), Pid?): dynamic process creation
- V := Expr: evaluate Expr and assign result to V
- if Expr then $P_1$ else $P_2$ fi
- if Expr then $P_1$ fi = if Expr then $P_1$ else delta 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: delay(Expr) or timeout(Expr)
- Absolute time: abs-delay(y, mon, d, h, min, s) or abs-timeout(y, mon, d, h, min, s)
- Example:
  - printf("expired") delay(10)
  - printf("Renew account") abs-timeout(2004,4,1,12,0,0)

# 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(`$Pname_1$, ..., $Pname_n$`)`
  - Eache *Pname* is optionally followed by parameters

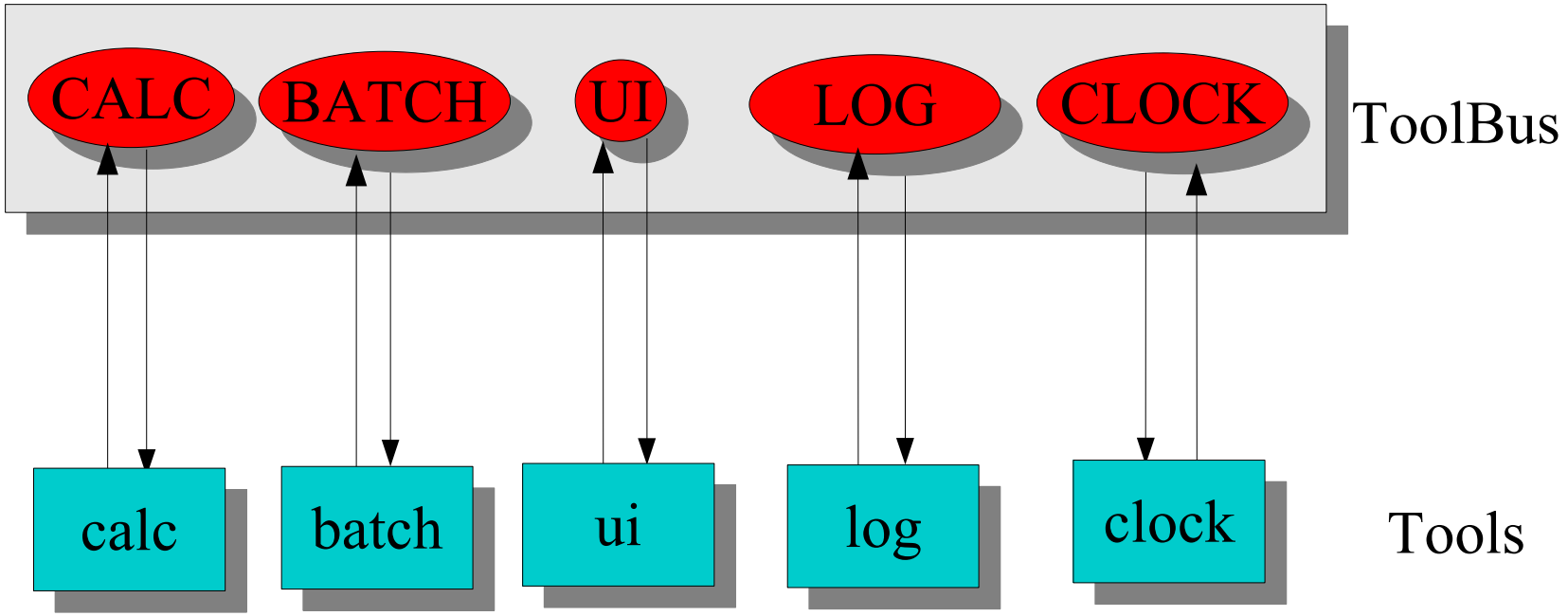
# Road map

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

# Road map

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

# 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"}
```

Receive compute message

Let calc do the computation

Note for the logger

Reply to compute message

# Process BATCH

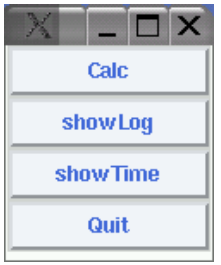
```
process BATCH is
    let Tid : batch, E : str, V : int
    in
        execute(batch, Tid?).
        (
            snd-eval(Tid, fromFile) . rec-value(Tid, expr(E?)) .
            snd-msg(compute, E) . rec-msg(compute, E, V?) .
            snd-do(Tid, toFile(E, V))
         ) * delta
    endlet

tool batch is {command = "batch"}
```

Get an expression from batch tool

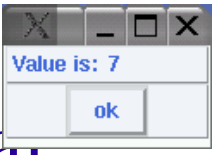
Evaluate expression

Value back to batch tool

# User-interface




- When the user presses Calc, a dialog window appears to enter an expression
- The result is diplayed in a separate window
- Pressing showLog display all calculations so far
- Pressing showTime displays the current time
- Pressing Quit ends the application

# User-interface: process UI

```
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

# User-interface: CALC-BUTTON

```
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)
    endlet
```

Calc button is pressed

Ask for an expression

Get cancel or an expression

Compute expression and display its value

Acknowledge the button event

# User-interface: LOG-BUTTON

```
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

```
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
```

```
process QUIT-BUTTON(Tid : ui) is
        rec-event(Tid, button(quit)) .
        shutdown("End of calc demo")
```

# Process LOG

```
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
```

Log all calculations

Show the log of calculations

# Process LOG1

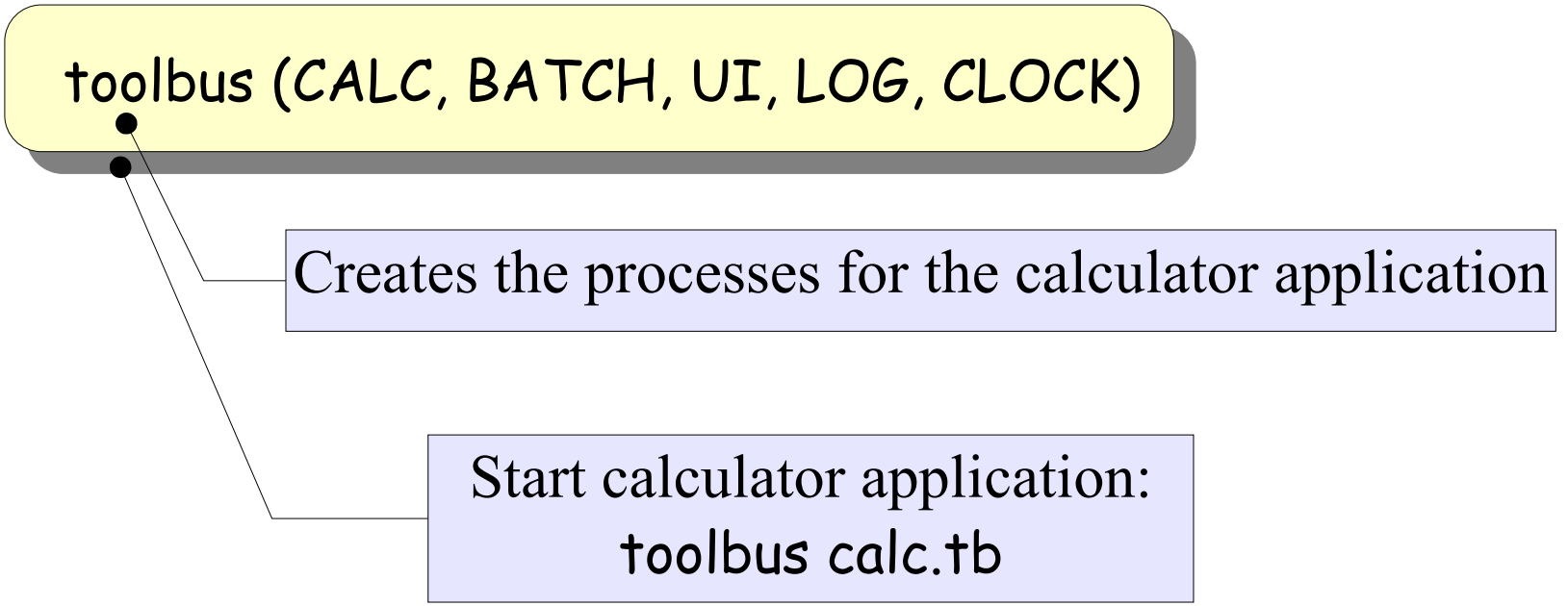
```
process LOG1 is
   let TheLog : list, E : str, V : term
   in     subscribe(compute(<str>, <term>)) .
          TheLog := [] .
             (      rec-note(compute(E?, V?)) .
                    TheLog := join(TheLog, [[E, V]])
              +
                    rec-msg(showLog) .
             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

Alternative definition of logger: maintain the log in a list

# Process CLOCK

```
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



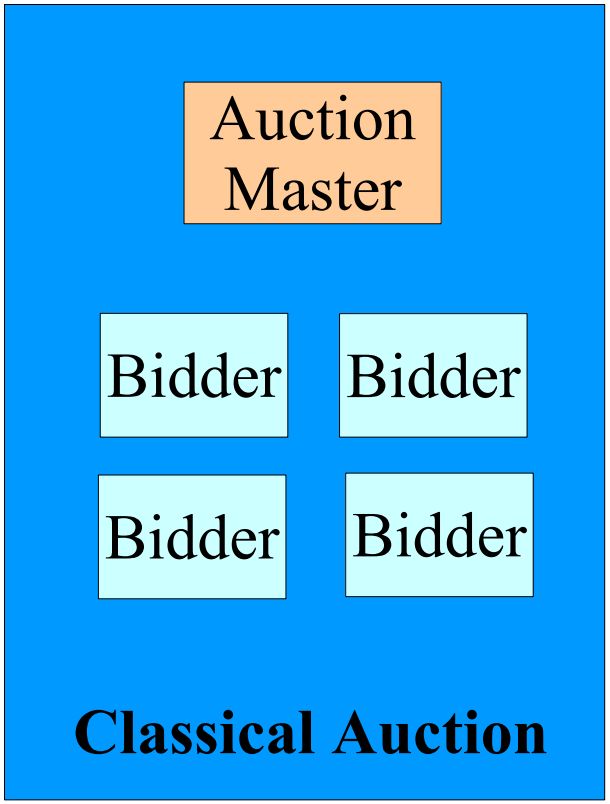
# Road map

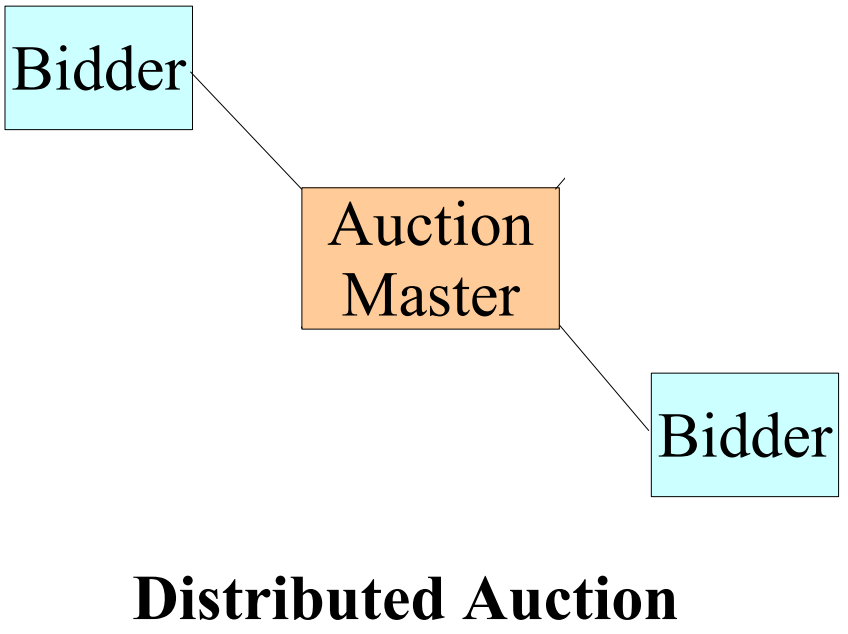
- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
  - calculator; auction; waves

# Road map

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

# Example: distributed auction

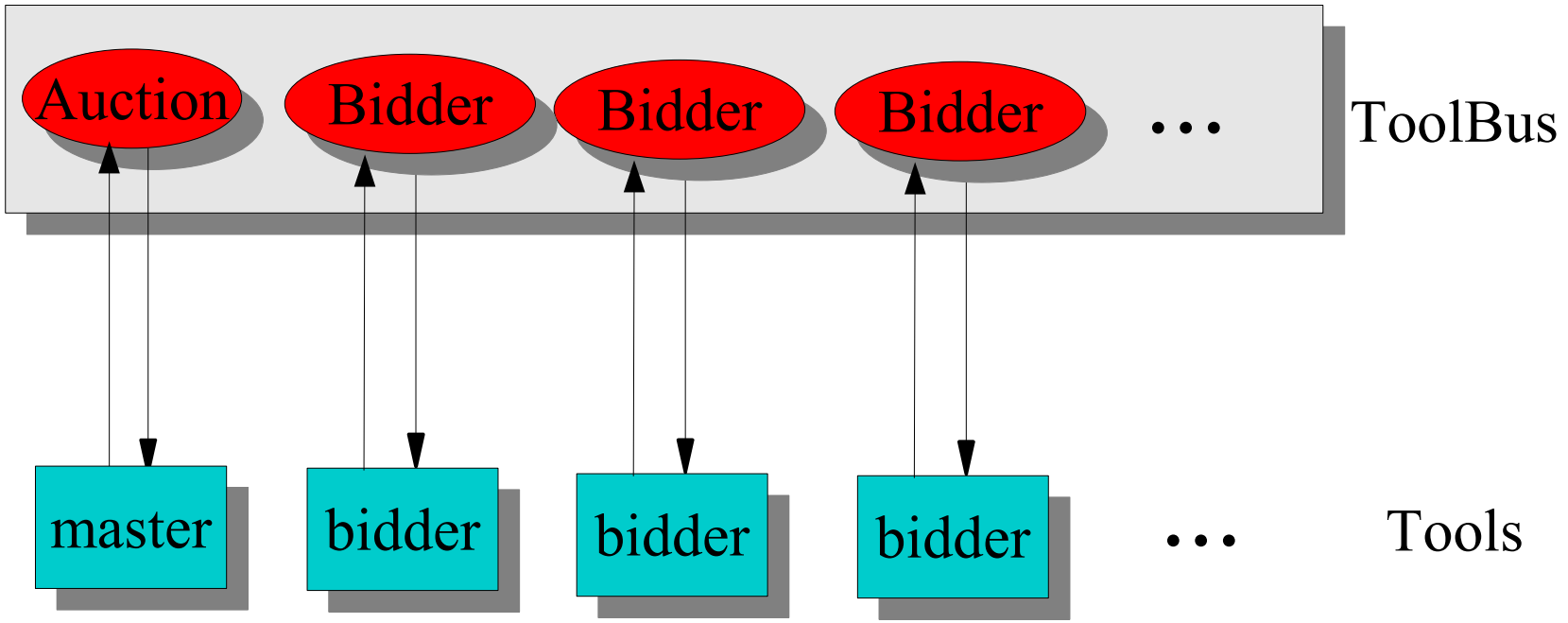


# Example: distributed auction

- 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 provess 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)) .
      Final := false . Sold := false .
      ( ...
      ) * if Sold then snd-ack-event(Mid, new-item(Descr, InAmount)) fi
  endlet
```

Where the action is ...

# Process OneSale

```
( if not(Sold) then ... fi
+ if not(or(Final, Sold)) then    ... fi
+ if and(Final, not(Sold)) then ... fi
+ ConnectBidder(Mid, Bid?) ...
) * if Sold then ... fi
```

# 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
```

Receive a bid from a bidder

Inform auction master about it

Reject bid if it is too low

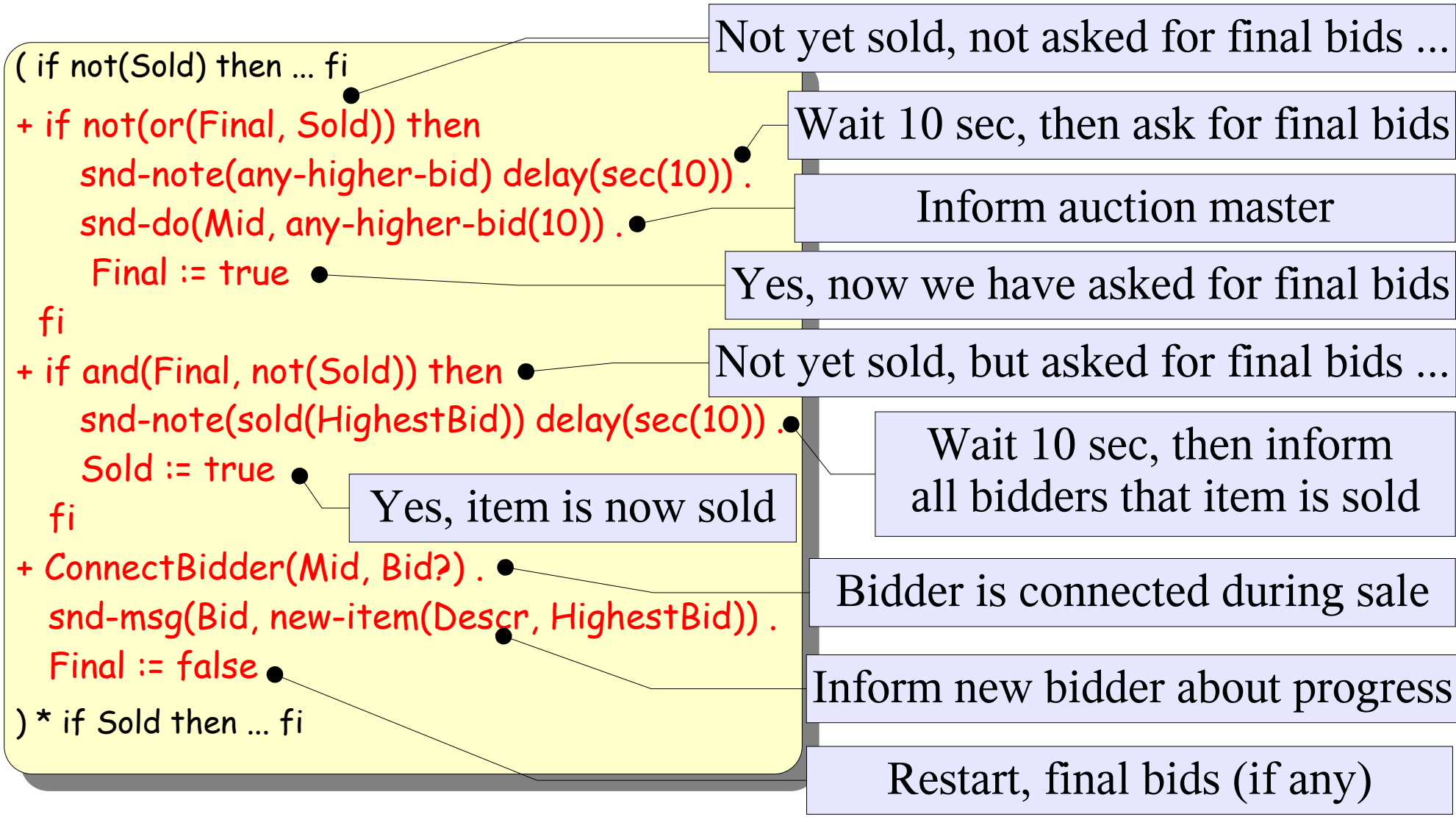
Remember as highest bid

Inform bidder: bid is accepted

Inform *all* bidders

Update auction master

# Process OneSale



# Process Bidder

```
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
```

# Process Bidder

Get info about item for sale after connection

```
( ( rec-msg(Bid, new-item(Descr?, Amount?))
  + rec-note(new-item(Descr?, Amount?))
  + rec-disconnect(Bid) . delta
  ) .
  snd-do(Bid, new-item(Descr, Amount)) .
  ( rec-event(Bid, bid(Amount?)) .
    snd-msg(bid(Bid, Amount)) . rec-msg(Bid, Acceptance?) .
    snd-do(Bid, accept(Acceptance)) . snd-ack-event(Bid, bid(Amount))
  + rec-note(update-bid(Amount?)) . snd-do(Bid, update-bid(Amount))
  + rec-note(any-higher-bid) . snd-do(Bid, any-higher-bid)
  + rec-disconnect(Bid) . delta
  ) *
  rec-note(sold(Amount?)) . snd-do(Bid, sold(Amount))
) *
delta
```

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

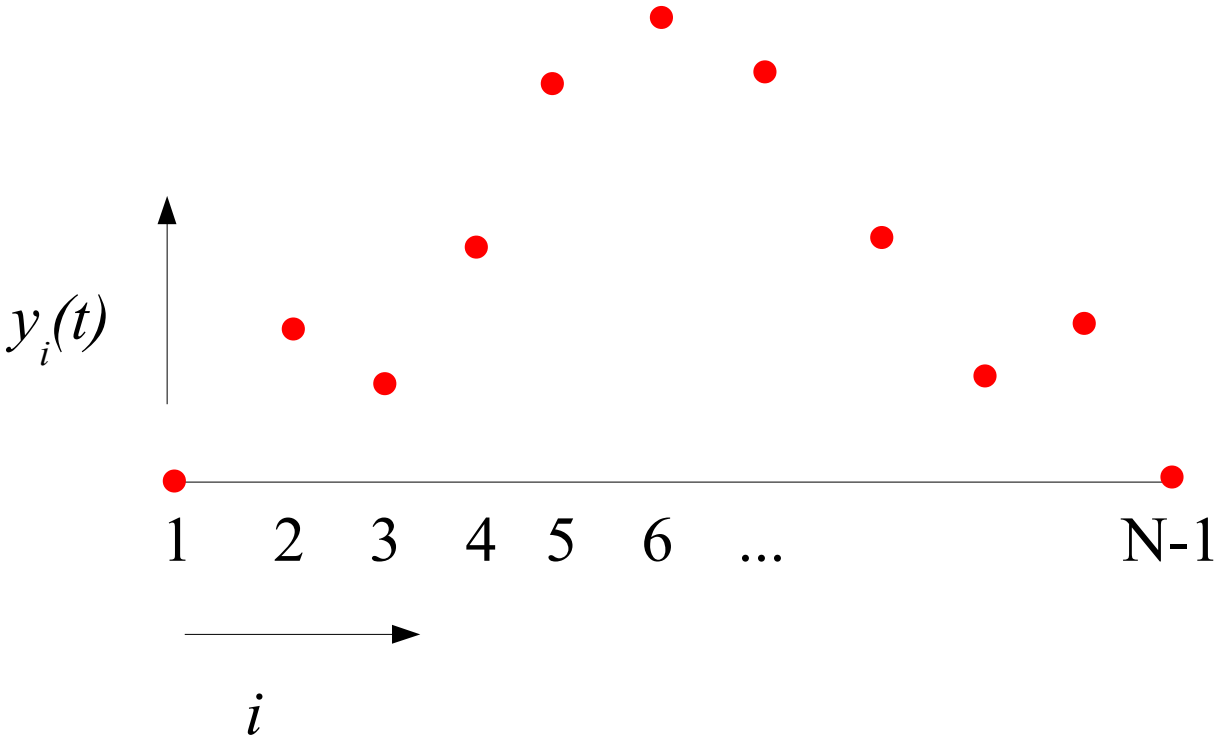
- The problem: component interconnection
- History & requirements
- Terms, types & matching
- The ToolBus architecture
- ToolBus scripts (Tscripts)
- Larger examples
  - calculator; auction; waves

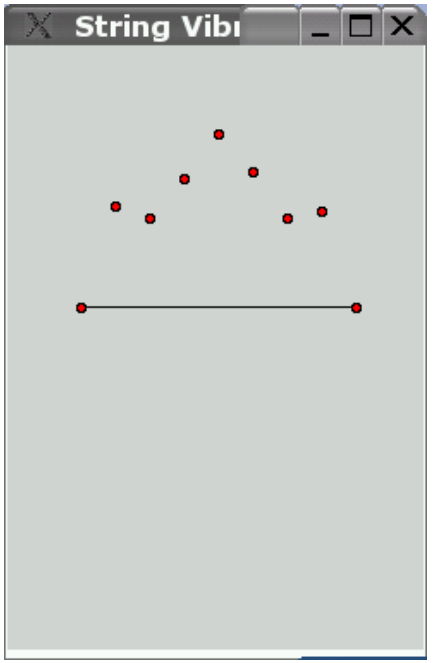
# Road map

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

# 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:

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

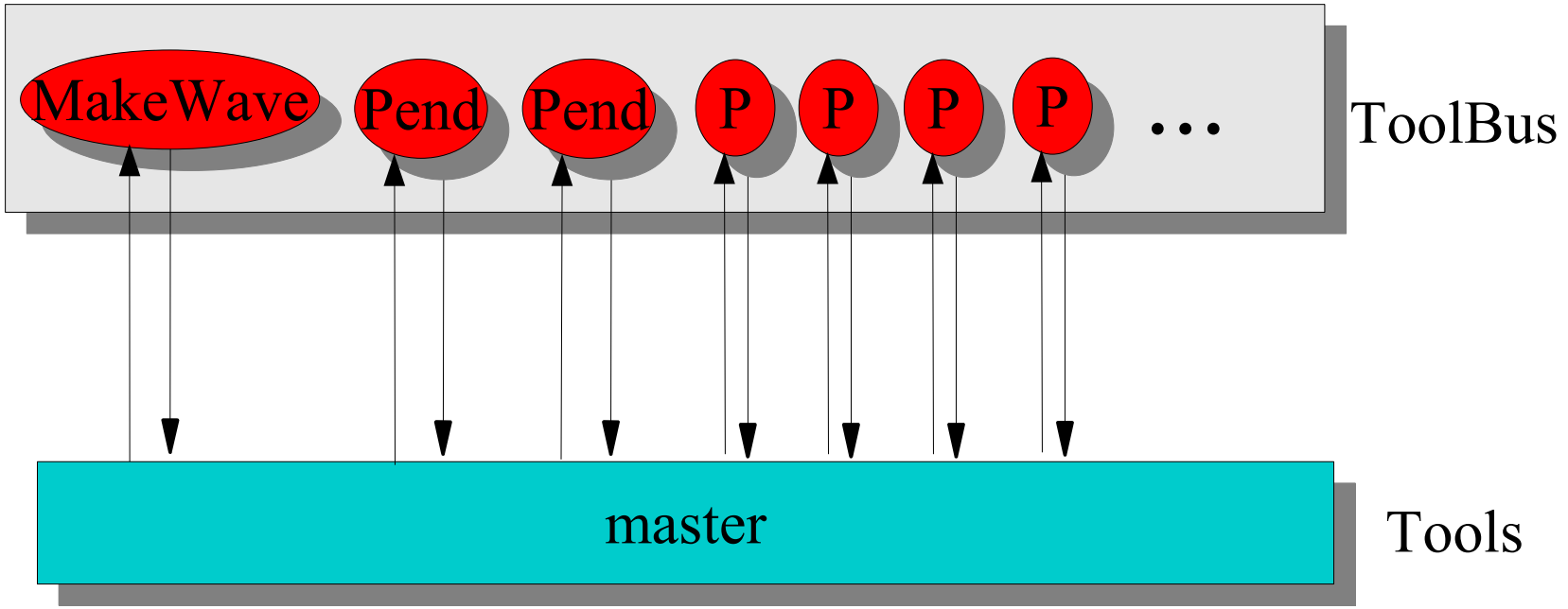
and

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

$\Delta x$: 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 `Pend` models the end points
- Process `MakeWave` constructs *N* connected instances of `P` and two end points
- Tool `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: amplitutes of this point

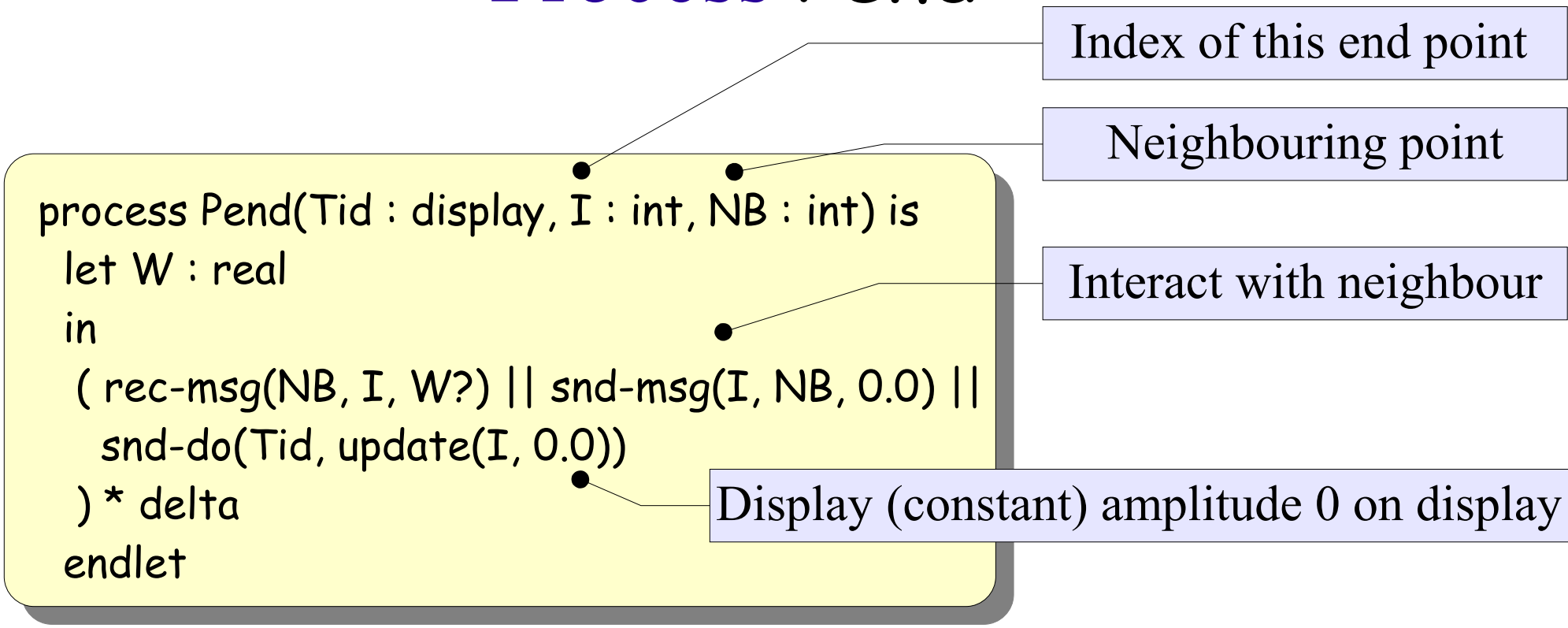
Receive amplitudes of neighbours

Send our amplitude to neighbours

Update our amplitude on display

Compute new versions of D and E

# Process Pend



# Process 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
```

Execute display tool and initialize it

Create the two end points

Create the other points

Shutdown after one minute

# 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

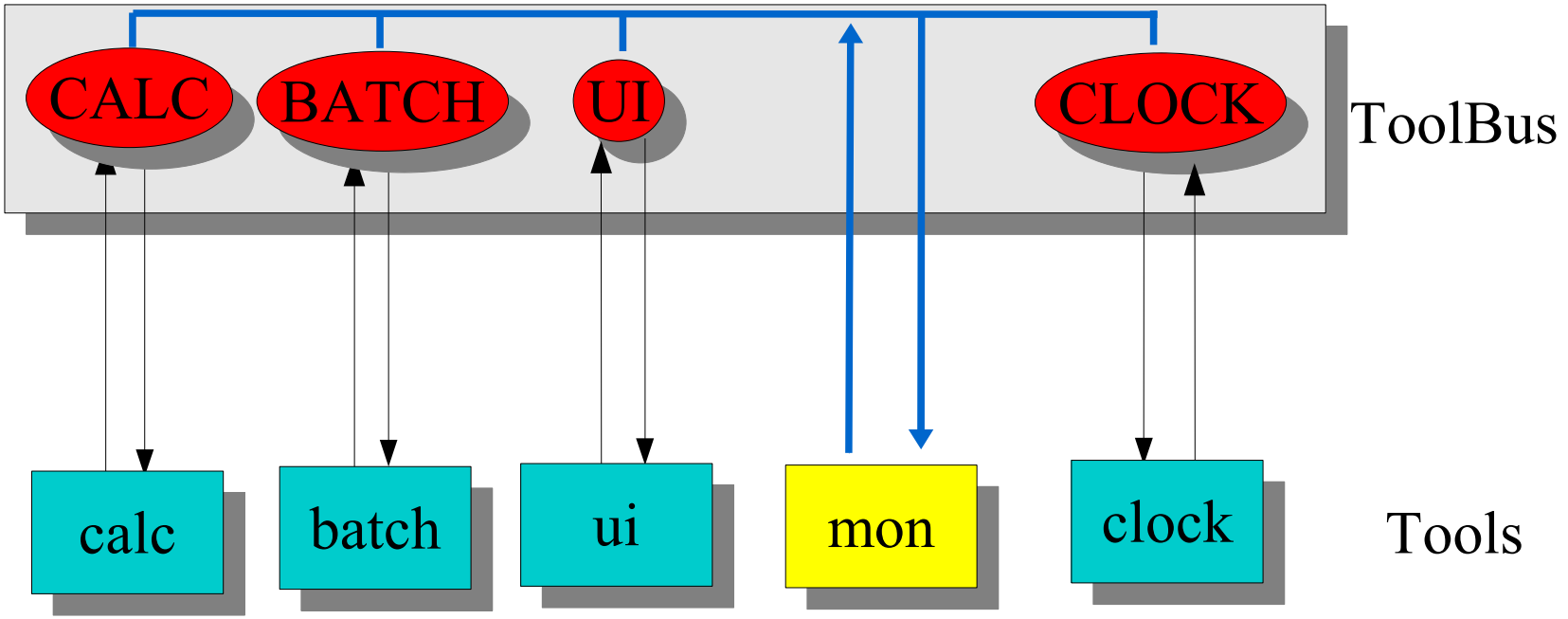
# Road map

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

# Not yet discussed: monitors

- A tool can act as a monitor for one or more ToolBus processes
- Monitors exist in three categories:
  - Loggers
  - Viewers
  - Controllers
- Depending on the category, monitors get all info about steps of the monitored processes they need

# A Monitor



# Loggers

- Non-interactive = processes don't wait for logger
- Non-intrusive = logger can't change process state
- Recording of bahaviour of processes
- Examples:
  - System logging
  - Generation of play back scripts for UI testing
  - Performance measurement
  - `toolbus -logger calc.tb`

# Viewers

- Interactive = processes wait for viewer
- Non-intrusive = viewer can't change state
- Viewing of processes
- Monitored processes wait for a continue message before proceeding
- Example:
  - Non-intrusive debugger
  - `toolbus -viewer calc.tb`

# Controllers

- Interactive & Intrusive control over processes
- Controlled processes wait for a continue message (containing new state/process expression)
- Examples:
  - Intrusive debugger
  - Advanced control applications that perform arbitrary computations on process expression in tool, but want to reuse other ToolBus tools
- Not implemented in current version!