The Mozart Programming System and Its Use for Agent Applications

Séminaire AXLOG
Ordonnancement et Planification sous
Contraintes pour Agents Autonomes

Peter Van Roy

Department of Computing Science and Engineering
Faculté des Sciences Appliquées
Université catholique de Louvain

Place Sainte-Barbe, 2
B-1348 Louvain-la-Neuve
Belgium

January 28, 2000

Overview

- Language, platform, application areas
  - ‘‘All is run-time’’ environment
  - Ultralightweight concurrency
- Inferencing support
- Distribution support
  - Fault tolerance support
  - Example 1: generic client-server
  - Example 2: fault-tolerant global store
- Agents and software components
  - Comparison with agent platforms
  - Some agent projects and applications
- Conclusions and perspectives

Mozart/Oz at a glance

Language

- Oz: A concurrent, state-aware, compositional, object-oriented language with dataflow synchronization
- First-class software components and resources

Strengths

- Concurrency: ultralightweight threads, dataflow
- Inferencing: constraint and logic programming
- Distribution: network transparent, open, fault tolerant
- Flexibility: dynamic, no limits, first-class compiler

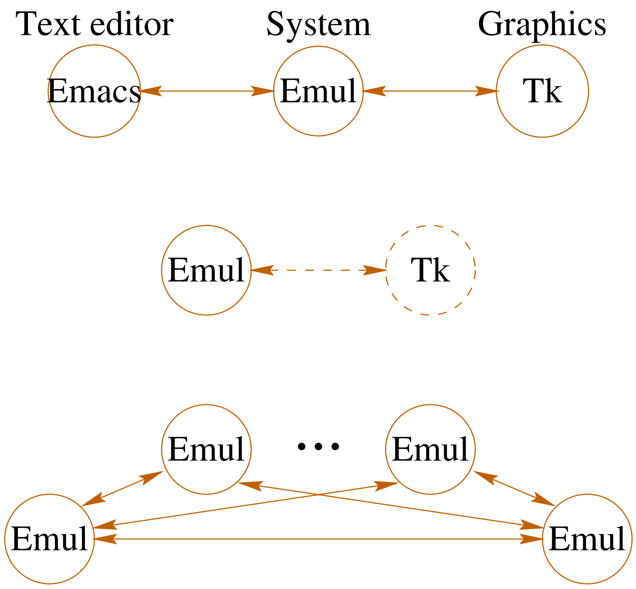
Research & development

- Vehicle: Mozart Programming System (Unix/Windows) by Mozart Consortium (Belgium, Germany, Sweden)
- Research in constraints, distribution, fault tolerance, resource management, security, implementation

Applications

- Major application areas: discrete optimization, ‘‘symbol crunching,’’ collaborative work, multi-agent systems
- For non-Consortium users: liberal license with sources (X11 style), maintenance, user group, technical support

Application development



Development configuration (each circle is a process)

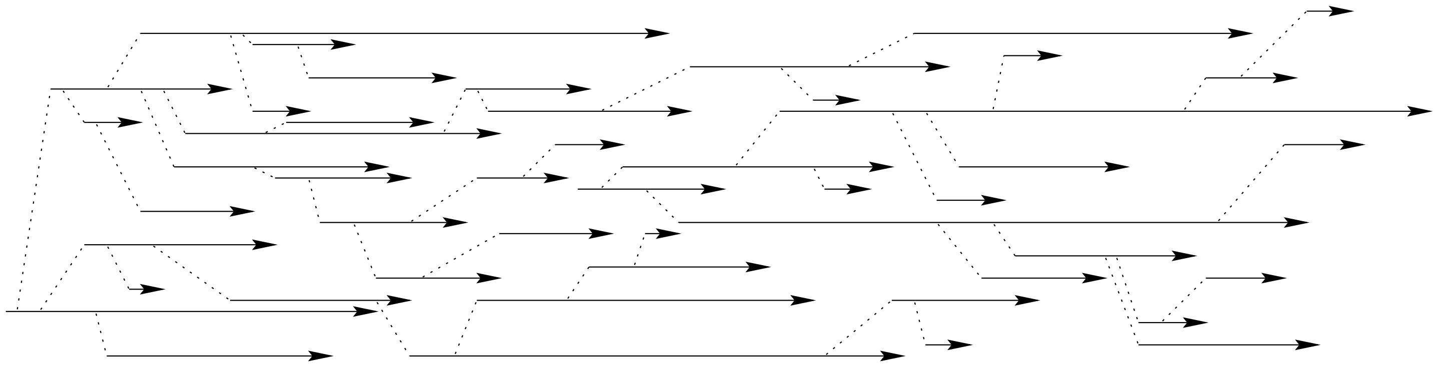
Standalone configuration (in centralized setting)

Distribution (any number of emulators communicate transparently)

- Development environment is part of run-time system:
  - Interactive user interface with concurrent tools (Browser, Explorer, debugger, panel, ...)
  - Incremental compiler is part of run-time system
  - Allows ‘‘gradual development’’: prototype with limited functionality is extended into full-fledged application
  - System is fully extensible at run-time, e.g., adding new types
- Distributed application can be completely developed in a centralized setting

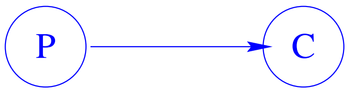
Concurrency is cheap and easy

- Lightweight threads
  - Preemptive scheduling with guaranteed CPU share
  - More than 100000 active threads on standard PC
- Dataflow synchronization makes concurrency easy
  - Synchronizing on data availability is invisible
- Better programming techniques are possible
  - Create thread whenever design requires it
  - Program stays simple and efficient



Performance

- Symbolic computing and inferencing
  - Competitive with best Prolog and constraint systems
  - Manipulating symbolic data 10x faster than Java 2
- Distributed computing



  - Producer/consumer example (stream of 1000000 integers):

System	Centralized	Distributed
Mozart 1.0.1	4 sec	8 sec
Java 2	18 sec	3600 sec

  - Mozart: 32 lines (identical code in both cases)
  - Java 2: 108 lines (centralized), 220 lines (distributed)
- General-purpose computing
  - Comparable with emulated Java 2

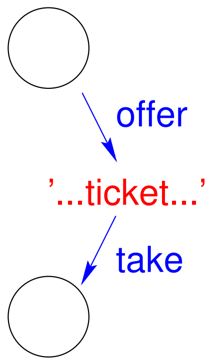
mozart

Inferencing support

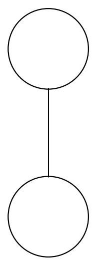
- Support for constraint-based inferencing
  - Basic search and inferencing strategies (module Search)
    - Single, all, best solution search (bab, restart)
    - Lazy or eager (lazy is similar to Prolog top level)
    - Parallel search for speedup
  - Finite domain and finite set support (modules FD and FS)
    - Lots of propagators, reflection, reification, distinct, etc.
    - Specification of search and inference strategies
  - Support for scheduling (module Schedule)
    - Serializers, distributors, cumulative constraints, etc.
- Support for logic programming
  - Rational tree support (like modern Prologs)
  - Don't know (Andorra style) and don't care disjunctions
  - Deep guards (nested spaces)
- Fully user-extensible and customizable (in Oz and C++)
  - Strategies in Oz (computation spaces, disjunctions)
  - Global constraints in C++ (propagators)

Distribution support (1/2)

- Distribution is always network transparent, i.e., same language semantics are obeyed independent of distribution structure.
- ‘‘Open’’ distribution (module Connection)



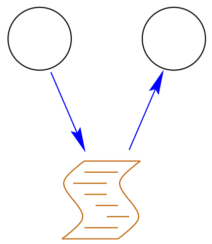
  - Ticket = Ascii representation of Oz store reference
  - Two operations: offer & take a ticket
  - Taking a ticket conceptually merges two stores.
    Any data can then be exchanged transparently.
    This implies distributed lexical scoping.
  - Implemented with distributed algorithm per type
- ‘‘Closed’’ distribution (module Remote)



  - One operation: a process creates another process with a shared reference
  - Useful for resource management and protection, e.g., server executes client command in another process
- Implementation carefully designed to provide a simple model of network communications that is efficient and predictable.

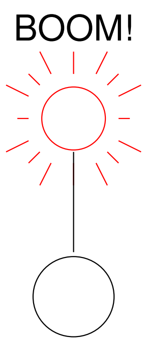
Distribution support (2/2)

- Persistence (module Pickle)



- Two operations: save and load any stateless data
- Stateless data includes procedures, classes, functors
- Can be loaded from URL

- Fault tolerance (module Fault)



- Detects site failures or network inactivity
- Lazy or eager detection per language entity
- Action: exception or replace by user-defined procedure
- High-level abstractions are built on top:
  disconnected operation, generic client-server,
  robust transactional store

- Other operations:

  - Web support (HTML, CGI applets, servlets)

Fault tolerance

- Failure model: (targeted for Internet applications)
  - Detectable permanent site failures (permFail)
  - Network inactivity, i.e., lack of information (tempFail)
- Any language entity can be used to detect these failures
- No timeouts are done by the network layer
  - tempFail is not a time out, but a detection of network inactivity that lets the application react quickly
  - tempFail can come and go repeatedly
  - Messages are never lost due to a tempFail
- This makes life easy:
  - Allows disconnected operation
  - Allows to bypass ‘‘boot time outs’’
  - Allows building powerful abstractions

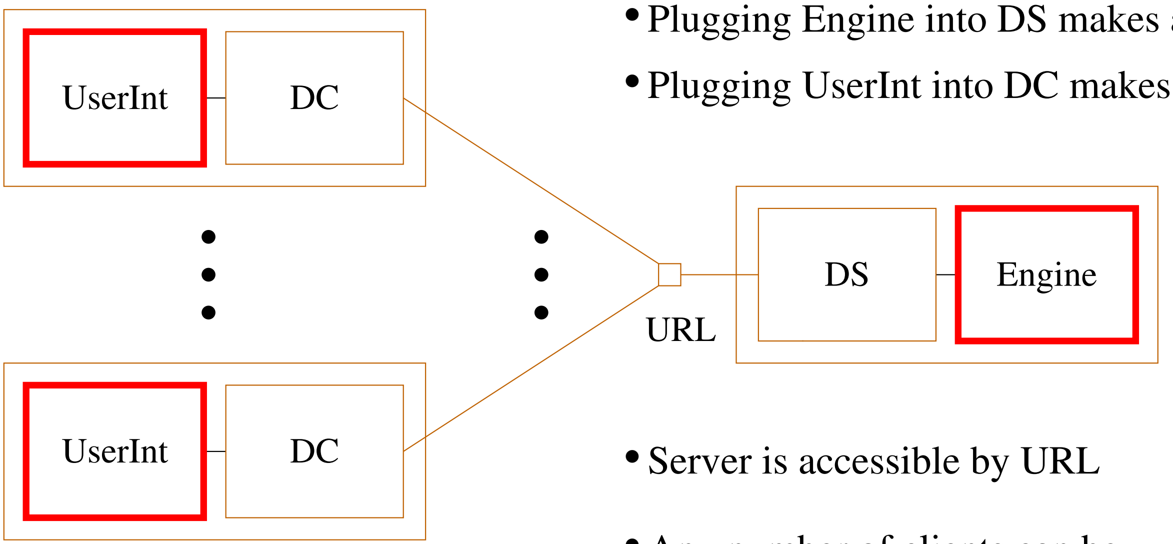
Example abstraction 1: generic client-server

- Take any centralized, sequential client-server application:

- The application must consist of two parts: a front end (UserInt) that queries a back end (Engine)

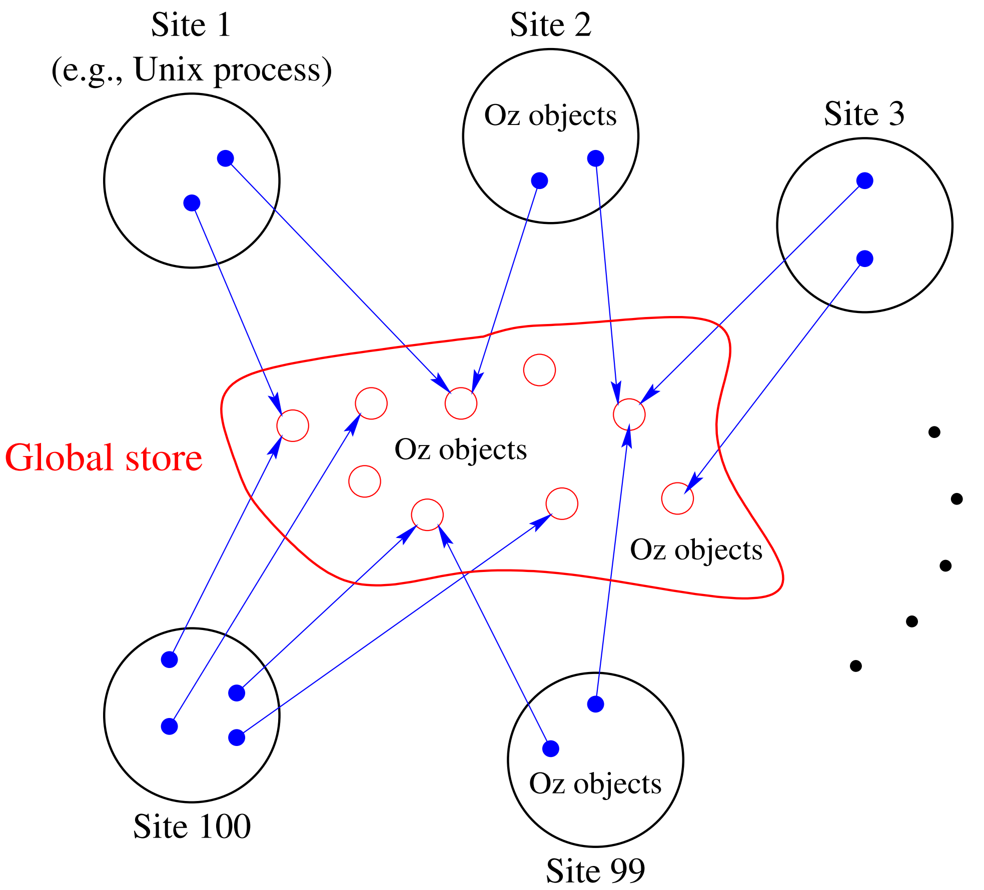
- Convert into an open, robust, distributed, concurrent application:

- No extra code has to be written
- Uses two Oz programs, DC and DS
- Plugging Engine into DS makes a server
- Plugging UserInt into DC makes a client



- Server is accessible by URL
- Any number of clients can be added or removed dynamically

Example abstraction 2: fault tolerant global store



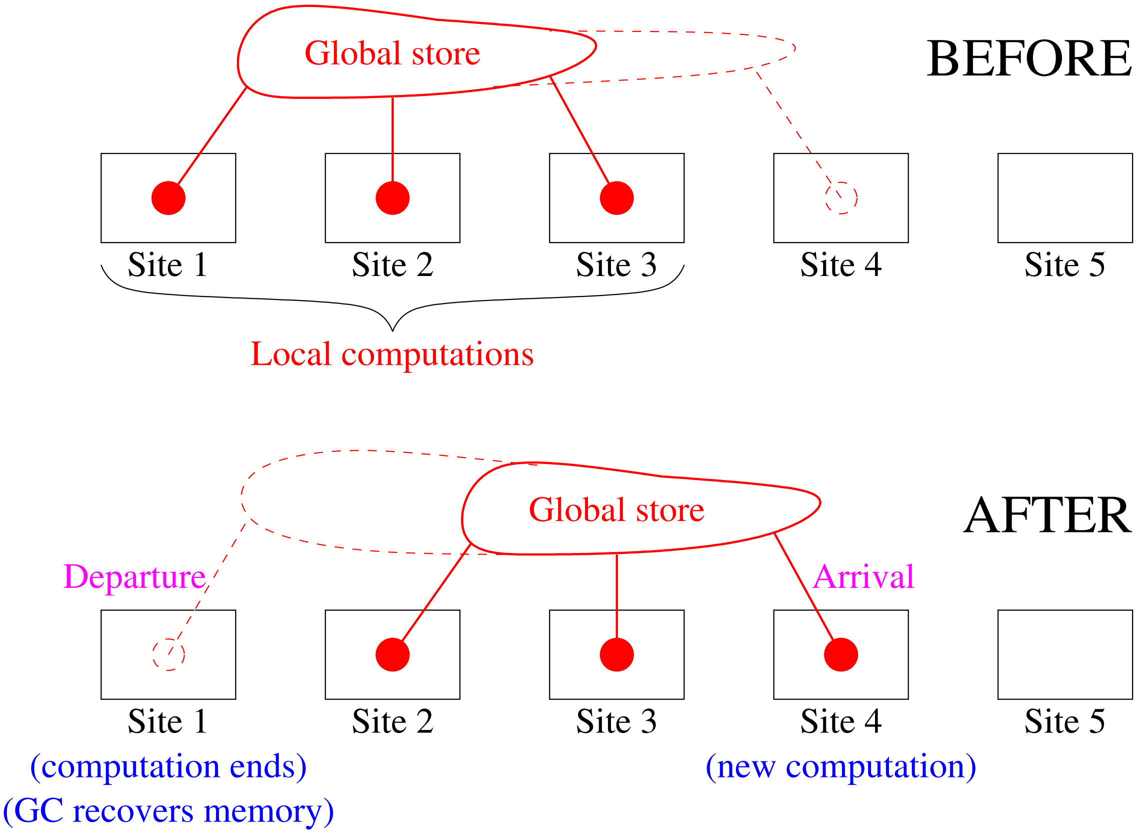
Any centralized application can be made collaborative (i.e., distributed) and robust with almost no effort

- Global store is shared, transparent, and coherent (as if centralized)
- Every site can update the store; store stays coherent (transactional)
  - Without waiting for network (optimistic, speculative)
  - Waiting for network (pessimistic, no wasted work)
- Sites can come and go at any time (open)
- As long as at least one site survives, the whole store survives (fault tolerant)

Agents in Mozart

- The word ‘‘agent’’ covers many concepts
- Let’s assume that a software agent has the following properties:
  - Autonomous
  - Situated in an environment, with which it interacts
  - Can cooperate with other agents
  - Potentially very long-lived
- We can implement this as a resource-aware distributed computation:
  - An agent accesses resources on sites (‘‘environment’’)
  - All agents live in the common Mozart computation space, and can therefore communicate (‘‘cooperate’’)
  - Agents can make parts of themselves persistent (‘‘long-lived’’)
- An agent accesses a new site’s resources by creating a functor and installing it on the site (functor = module w/ resource spec)
- An agent moves by creating new functors and installing them
  - Mobility is a consequence of network transparency; somewhere between ‘‘weak’’ and ‘‘strong’’ migration
  - Agent migration is a special case of the global store: with no extra effort, it is fault tolerant and permits home communication

Mobile agents with the global store



- Mobile agents are a simple special case of the global store
- Properties:
  - Migration without any dependencies
  - Communication with home site, independent of migration
  - Full fault tolerance: local computations can assume they're reliable

Software components: modules, resources, migration

- Module: a record grouping related operations together
- Resource: a module that is tied to a site, i.e., it has site-dependent state that is outside of the Oz store (e.g., in emulator or OS)
  - Typical resources: Connection, Fault, Tk, OS, Browse, ...
- Functor: a module specification
  - Defines module operations, initialization, and the resources that the module needs
  - Unlike a module, a functor is stateless (storable in file)
  - Functors are first-class: they can be created at run-time and they can have external references (‘‘computed functors’’)
- Functors are the migration units
  - Just install a functor on a new site
  - This automatically plugs in the resources the functor needs

Resource example

- The Tk module is a resource since it refers to the tk process
- The following code defines a module QTk that needs Tk
  (QTk defines a high-level abstraction for building user interfaces)

```
declare
functor QTk
   import Tk Open Pickle ...
   export DialogBuilder DialogBox ...
define
   ...here comes the body code...
end
```

- QTk can be compiled and linked either inside or outside
  of the Mozart environment

Comparison with agent platforms

- Examples: ObjectSpace Voyager, IBM Aglets, Mitsubishi Concordia

  All are ‘‘100% pure Java’’

- Above platforms summary:

  + Agent-specific abstractions: mobility, communication, host access

  + Security support

  + Easy interoperability with ‘‘mainstream’’ Java

  - Built on top of Java: inherit Java problems (non-transparent distribution, class-name conflicts, expensive & nonpreemptive threads, site-based security)

- Mozart summary:

  - Less agent-specific support

  + More powerful concurrency, distribution, fault tolerance

  ++ Much more powerful inferencing and symbolic computation support

  + More powerful abstraction ability (fewer ‘‘basic concepts’’)

  + Much simpler formal semantics (many fewer ‘‘quirks’’)

  + Interactive incremental development (all is ‘‘run-time’’)

  + Open source license

  - Non-mainstream language and system

Some agent projects

- DMS project (SICS + Ericsson Hewlett-Packard)
  - FIPA-style platform on top of Mozart
  - Study interaction protocols and build a library for real-world agent applications
  - Take advantage of distribution, symbolic manipulation
- InfoCities project (European Union Fifth Framework)
  - Study evolution of Internet ‘‘information cities’’ and the laws governing their evolution
  - Large-scale simulations (millions of agents)
- COORD project (SICS)
  - Develop market-based interaction models for agents
  - Complex plans and resource allocation in ‘‘bandwidth market’’

Some recent applications

- TransDraw (UCL, PIRATES project)
  - Open, collaborative graphic editor (functionality similar to xfig)
  - Uses transactional protocol to overcome network delay
- Cow Disorder Expert System (CLAES, Egypt)
  - In use with farmers and veterinary doctors: CBR, 3000 cases, 428 disorders, being extended
  - Originally implemented in Prolog, recent translation to Mozart improved speed, distribution, robustness
- Friar Tuck (National University of Singapore, ReAlloc project)
  - Constraint-based round robin tournament planner
  - Used to schedule several sports tournaments (football, basketball, ...)
- Mozart Instant Messenger (SICS)
  - Similar functionality to ICQ
  - Dynamically extensible with new Mozart applications

Conclusions and perspectives

- This talk has given a high-level overview of the Mozart Programming System.
- In Mozart, any distributed application behaves exactly as if it were centralized.
- This vastly simplifies the development of distributed applications including agent-based ones.
- Current research includes construction of agent platforms that take advantage of Mozart's strengths, advanced constraint debugging, high-level abstractions for fault tolerance, adding security to the network layer, and implementations for devices with limited resources.
- Mozart is the result of a long-term research effort of the Mozart Consortium (DFKI, SICS, UCL, UdS). The project started in 1991. Work on distribution started in 1995. The first public release was in Jan. 1999 (http://www.mozart-oz.org) .
- At UCL, the PIRATES project does Mozart research. We are currently working on a global fault-tolerant transactional store, on high-level abstractions for user-interface design, and on formalisms for defining and reasoning about protocols.