Mozart: A Programming System for Agent Applications

Peter Van Roy
Université catholique de Louvain and SICS
pvr@info.ucl.ac.be

Seif Haridi
Swedish Institute of Computer Science
seif@sics.se

November 3, 1999

Abstract

The Mozart Programming System is a development platform designed for distributed programming, symbolic computation, and constraint-based inferencing. This article gives a survey of the abilities of Mozart for open, concurrent, resource-aware distributed computing. We show by example how easy it is to develop applications with these properties. This makes Mozart particularly well-suited for building agent applications. We give a summary of some current agent-based projects in Mozart.

1 Introduction

The Mozart Programming System is a general-purpose development platform that was designed specifically to support concurrency, distribution, resource-aware computation, and symbolic computation and inferencing [4]. This makes it well-suited for agent-based programming.

Mozart implements Oz, a multiparadigm language that resists easy classification. Oz has a simple underlying model based on concurrent constraints extended with state and higher-orderness [2]. Oz can appear to the programmer as a concurrent object-oriented and functional language.

This article is structured as follows. Section 2 talks about symbolic computation and constraint-based inferencing. Section 3 talks about concurrency and explains why dataflow synchronization is important. Section 4 shows transparent distribution with small code examples and explains why it is efficient. Section 5 explains how to do resource-aware computation. Section 6 lists some current agent-based projects using Mozart as their development platform. Finally, Section 7 gives perspectives on the future evolution of Mozart.

Mozart was developed by the Mozart Consortium, a loose collaboration between the German Research Center for Artificial Intelligence (DFKI), Germany, the Swedish Institute of Computer Science, Sweden, the Université catholique de Louvain, Belgium, and the Universität des Saarlandes, Germany.

2 Symbolic computation and inferencing

The Oz language provides the abilities of constraint, logic, and lazy functional languages. There are higher-order functions with lexical scoping. Functions can be declared as lazy. There are three constraint systems, namely rational trees (as in modern Prologs), finite domains, and finite sets of integers. Performance is competitive with commercial systems (including ILOG Solver, CHIP, SICStus Prolog, and Prolog IV), but Oz is much more expressive and flexible, providing first-class computation spaces, programmable search strategies, a GUI for the interactive exploration of search trees, parallel search engines that exploit networked computers, and a programming interface to add new, efficient constraint systems. Parallel search engines make it easy for ordinary users to vastly increase the performance of compute-intensive applications, without having to rewrite them.

3 Concurrency and dataflow synchronization

Mozart provides lightweight threads and dataflow synchronization. These help the programmer structure his or her application in a modular way. Threads are so cheap that one can afford to create them in large numbers. On most machines at least 100000 simultaneous active threads are possible; more if main memory is very large. Dataflow synchronization means that concurrent programming becomes very easy. For example, synchronizing on data availability is completely invisible. This is one of the most common concurrent operations.

Because of dataflow synchronization, many concurrent and distributed programming idioms become very simple [3]. They are also efficient. For example, we have measured a producer/consumer example that generates a stream of 1000000 integers and sums them [1]. This does asynchronous FIFO communication between the producer and consumer. We compared Java 2 (JDK 1.2) with Mozart 1.0.1 on an UltraSPARC 2 running Solaris. For Mozart, the centralized and distributed solutions both require 32 lines of code (the code is identical in both cases, see Section 4), running in 4 and 8 seconds, respectively. For Java, the centralized and distributed solutions are quite different (108 and 220 lines of code; distribution uses RMI), running in 18 and 3600 seconds, respectively. The distributed solutions run the producer and consumer on different machines on the same LAN.

We put this example in perspective. For general computations, Mozart and Java 2 have comparable performance. However, in concurrent and distributed programming and in symbolic computation, Mozart outperforms Java 2 significantly.

4 Distribution

In Mozart, a distributed program is a program that is partitioned between a set of "sites," where a site is just an operating system process. Sites can be on the same machine or on different machines.

Distributed programming in Mozart is ridiculously easy because it is network transparent. That is, the same program can be spread out over more than one site, and it will still do exactly the same computation. From the programmer's point of view, the network is invisible. Whether a computation happens here or there has no effect on what the program does. It only has an effect on how long it takes for the program to do it.

In our experience, new users find it hard to believe that Mozart is really network transparent and that this does not result in inefficient network operations. So we'll start off by showing exactly how this is done. In fact, Mozart also makes it easy to ensure fault tolerance; this is explained in the distribution tutorial [5].

4.1 Network transparency and openness

Network transparency means that a program will perform exactly the same computation independently of how it is partitioned over a set of sites. This means that a language entity used on one site has to behave in rigorously the same way if it is referenced from many sites. To illustrate this, we'll give self-contained code examples that can be run in Mozart's interactive user interface. In these examples, we will reference language entities from two sites.

In Mozart, openness is implemented by means of *tickets*. A ticket is a global reference into a Mozart store, represented as an ASCII string. A Mozart computation can create a ticket for any local reference. A second computation gets the reference by getting the ticket. Since the ticket is an ASCII string, there are a million and one ways that another process can get the ticket. It could be through a shared file, through a Web page, through email, etc.

Let's start with a simple example. The first process has a big data structure that it wants to share. It first creates the ticket:

```
declare
X=the_novel(
     text:"It was a dark and ..."
     author:"E.G.E. Bulwer-Lytton"
     year:1803)

{Show {Connection.offerUnlimited X}}
```

Creating and getting tickets are implemented through the module `Connection`. The code creates the ticket (with `Connection.offerUnlimited`) and displays it in the Mozart emulator window (with `Show`). Any other process that wants to get a copy of `X` just has to know the ticket. Here's what the other process does:

```
declare
X2={Connection.take
    '...ticket comes here...'}
```

(To make this work, replace the text `'...ticket comes here...'` by what was displayed by the first process.) That's it. The reference `X2` now points to the big data structure. Both `X` and `X2` behave identically.

This works for other data types as well. Let's say the first process has a function instead of a record:

```
declare
fun {MyEncoder X}
   (X*4449+1234) mod 33667 end

{Show {Connection.offerUnlimited
   MyEncoder}}
```

The second process can get the function easily:

```
declare
E2={Connection.take
    '...MyEncoder's ticket...'}

{Show {E2 10000}}  % Call function
```

In addition to records and functions, the ticket can also be used to pass unbound variables. Such variables can be bound to exactly one value. Any operation that needs the value will wait; this is how Mozart does dataflow synchronization [3]. The first process creates the variable and makes it globally accessible:

```
declare X

{Show {Connection.offerUnlimited X}}
```

But the process does not bind the variable yet. Other processes can get a reference to the variable:

```
declare
X={Connection.take
   '...X's ticket...'}

{Browse X}
```

Unlike `Show`, which just prints the current value, `Browse` is a concurrent tool that observes the store continuously and tells us when the variable is bound. If we bind the variable to something, then the binding will become visible at all sites that reference the variable. Any process can bind the variable, including a different process than the variable's creator.

With tickets, one can pass references to *any* data type, including objects, classes, functors, and ports. For example, here's how to distribute an object:

```
declare
class Coder
   attr seed
   meth init(S) seed<-S end
   meth get(X)
      X=@seed
      seed<-(@seed*1234+4449)mod 33667
   end
end

C={New Coder init(100)}

{Show {Connection.offerUnlimited C}}
```

This defines the class `Coder` and an object `C`. Any process that takes the object's ticket will reference it. The Mozart system guarantees that the object will behave exactly like a centralized object. For example, if the object raises an exception, then the exception will be raised in the thread calling the object.

As a final example, let's use a port to make an open server. A port is an Oz type; it is a channel with an asynchronous send operation. It guarantees that successive sends in the same thread will appear in the same order in the channel's stream. Let's create a port, make it globally accessible, and display the stream contents locally:

```
declare S P in
{NewPort S P}
thread
   for X in S do
      {Browse X}
   end
end

{Show {Connection.offerUnlimited P}}
```

This sets up a thread to display everything sent to the port. The `for` loop[1] causes dataflow synchronization to take place for elements appearing on the stream `S`. Each time a new element appears, the loop does a new iteration. Here's how a second process sends to the port:

```
declare
P={Connection.take
```

[1] The `for` syntax is supported starting from Mozart 1.1.0.

```
'...P's ticket...'}

thread {Send P 100} {Send P 200} end
thread {Send P foo} {Send P bar} end
```

In the first process, `100` will appear before `200` and `foo` will appear before `bar`.

4.2 Network awareness

These examples show us that the apparently simple module `Connection` is actually doing many different things. Its implementation uses a wide variety of distributed algorithms to provide a simple abstraction to the programmer.

We have left one big question unanswered. We've seen that `Connection` lets us build network-transparent connections between processes. But what price have we paid in terms of network operations? There are in fact two related questions:

- What are the network communications that the system uses to implement the transparency?
- Are the network communications *predictable*, i.e., is it possible to build applications that communicate in predictable ways?

As we will see, the network communications are both few and predictable; in most cases exactly what would be achieved by explicit message passing. This property of the implementation is called *network awareness*. Here's a quick summary of what happens for the most-used language entities; for more details please see the distribution tutorial [5].

- **Records**, **procedures**, and **functions**. These are copied over immediately when the ticket is taken. This takes one round trip, i.e., two messages. At most one copy of a given procedure or function can exist on a site.
- **Dataflow variables**. When binding the variable, one message is sent to each site that references the variable.
- **Objects**. By default, objects execute locally on each site that calls them. A mobility protocol moves the object to a site that asks for it. This requires a maximum of three messages for each object move. This is optimized for the case when the object is updated frequently; it is also possible to optimize for other cases.
- **Ports**. Sending to a port is both asynchronous and FIFO. Each element sent causes one message to be sent to the port's home site. This kind of send is not possible with RMI, but it is important to have: in many cases, one can send things without having to wait for a result.

It is clear that the distributed behavior of these entities is both simple and well-defined. In a first approximation, we recommend that a developer just ignore it and assume that the system is being essentially as efficient as a human programmer doing explicit message passing. There are no hidden inefficiencies.

5 Resource-aware computing

By *resource* we mean any system ability that is restricted to a single site. An important part of any adaptive distributed computation is that it can specify dynamically what resources it needs, and link itself to those resources on the site it is initiated.

In Mozart, the basic unit of resource-aware computation is the *functor*. A functor is just a module specification that lists the resources the module needs. For example:

```
declare
functor F
   import OS
   export time:T
define
   S P={NewPort S}
   thread
      for X in S do {OS.time X} end
   end
   fun {T} X in {Send P X} X end
end
```

The functor `F` specifies a module that imports the resource `OS`, defines a server (i.e., the port and its thread), and exports the function `T`, which queries the server. The resource `OS` contains basic operating system functionality. The procedure `OS.time` returns the current time in seconds since Jan. 1, 1970.

There are several ways to install a functor on a site. The easiest is by using a compute server. A compute server takes a functor, links the site's resources to the functor, and executes the functor body, thus creating a module. Compute servers are defined in Oz [5]. For example, let's say that `CS` is a local reference to a remote compute server. Then the following

code will install the functor remotely and give local access to the function `T`:

```
declare
T={CS F}.time

{Browse {T}}
```

The function `T` is asynchronous; it can be made synchronous by adding `{Wait X}` to its definition. The call `{Wait X}` suspends until `X` is bound to a value.

6 Agent-based projects using Mozart

We mention some of the projects that are using Mozart as their main development platform:

- The COORD project's main goal is to develop methods to coordinate the actions of agents by using decentralized market-based models of interaction. The project will develop techniques to construct complex plans and resource allocations, and apply the results in a "bandwidth market." An original approach is to use derivatives to express more complex resource demands while maintaining the volatility of market resources. Mozart is used for agent simulation. The main contact is Lars Rasmusson (`lra@sics.se`).

- The DMS project's main goal is to develop a multi-agent platform on top of Mozart. The platform is inspired by FIPA, but uses the properties of Mozart to provide a more powerful yet more concise expression of agents. A secondary goal is to study interaction protocols and build a library of protocols useful for building real-world agent applications. The main contact is Fredrik Holmgren (`fredrikh@sics.se`).

- The ESPRIT project ToCEE's main goal is to develop a distributed environment for cooperative and concurrent engineering in the building and construction industry. Mozart was chosen for its fine-grained concurrency, metaprogramming abilities, constraint-based inferencing, and ability to efficiently construct user interfaces. The main contact is Rainer Wasserfuhr (`Rainer.Wasserfuhr@cib.bau.tu-dresden.de`).

- The InfoCities project (European Fifth Framework Programme) will study the evolution of information cities in the Internet. Mozart will be used as a platform to house large-scale agent simulations (millions of agents). The project will study the laws governing their evolution. The main contact is Seif Haridi (`seif@sics.se`).

7 Conclusions and perspectives

This article explains some of the strengths of the Mozart Programming System. In Mozart, any distributed application behaves exactly as if it were centralized, in contrast to Java-based platforms. 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, security, and implementations for devices with restricted resources.

References

[1] Per Brand. An example of programming in Mozart versus Java, October 1999. Available at `http://www.sics.se/~seif/JavaVSMozart.html`.

[2] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, May 1998.

[3] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, May 1999.

[4] Mozart Consortium. The Mozart Programming System, January 1999. Available at `http://www.mozart-oz.org/`.

[5] Peter Van Roy, Seif Haridi, and Per Brand. Distributed programming in Mozart – A tutorial introduction. Technical report, Mozart Consortium and Université catholique de Louvain, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.