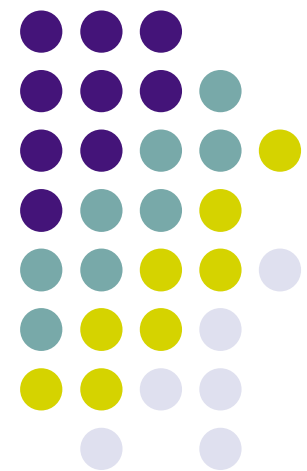


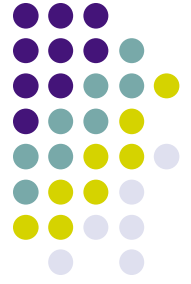
# General Overview of Mozart/Oz

---

**Peter Van Roy**

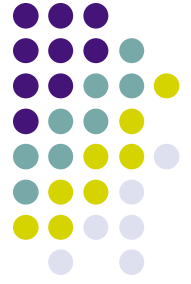
[pvr@info.ucl.ac.be](mailto:pvr@info.ucl.ac.be)





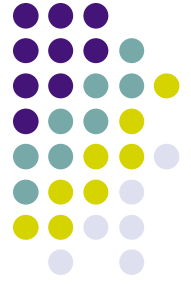
# At a Glance

- Oz language
  - Dataflow concurrent, compositional, state-aware, object-oriented language
  - First-class software components and resources
- Language strengths
  - **Concurrency**: ultralightweight threads, dataflow
  - **Inferencing**: constraint and logic programming, well factored
  - **Distribution**: network transparent, open, fault tolerant
  - **Graphical user interfaces**: dynamic, adaptable
- System properties
  - Mozart: **efficient** implementation, portable, Open Source
  - Simple **formal semantics**
  - **Flexibility** with few limits, first-class compiler



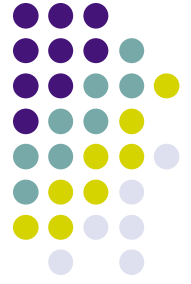
# Mozart's Synergy

- Mozart has the synergy that comes from combining different areas in a deep way
  - Concurrency - constraints - distribution - flexibility - ...
- We have just started to explore the consequences of this synergy. Here are a few examples (certainly a very personal nonexhaustive list!):
  - C. Schulte's parallel search engine
  - D. Grolaux's FlexClock with flexible user interface
  - Component programming in the ICITIES project
  - Community panel in the PEPITO project (peer-to-peer)
- I encourage you to take advantage of MOZ 2004 to learn those parts you don't know yet!



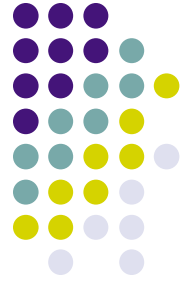
# Coming to Grips with Oz

- Oz supports many programming styles, in isolation and in mixes
- Possibly you want to use more of its possibilities? Then read the following books:
  - *Concepts, Techniques, and Models of Computer Programming*, by Peter Van Roy and Seif Haridi
  - *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, by Christian Schulte
  - *Concurrent Constraint Programming in Oz for Natural Language Processing*, by Denys Duchier, Claire Gardent, and Joachim Niehren



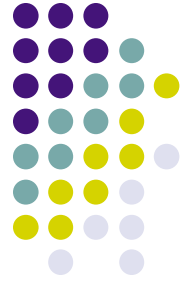
# Areas Covered by MOZ 2004

- Computer science education
- Software engineering
- Human-computer interfaces and the Web
- Distributed programming
- Grammars and natural language
- Constraint research
- Constraint applications



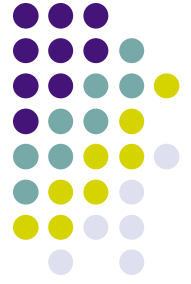
# Invited Talks

- *The Development of Oz/Mozart*  
by Gert Smolka
  
- *The Structure of Authority: Why Security is Not a Separable Concern*  
by Mark S. Miller (speaker), Bill Tulloh, and Jonathan Shapiro



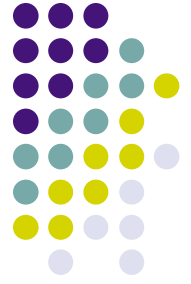
# Strengths

- Strengths
  - Concurrency
  - Inferencing
  - Distribution
  - Graphical user interfaces
  - Components and agents
  - Formal semantics
- Let's take a look at some of these strengths



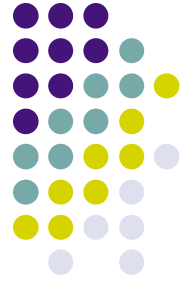
# Concurrency (1)

- Lightweight threads
  - Preemptive scheduling with guaranteed CPU share
  - More than 100000 active threads on standard PC
- Dataflow synchronization makes concurrency easy
- Better programming techniques are possible
  - Create a thread whenever the design requires it
  - For example, a thread can be created for each active entity in a multi-agent system



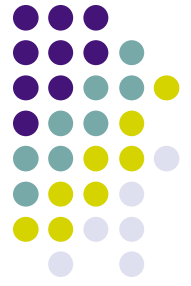
# Concurrency (2)

- Declarative concurrency
  - No race conditions (deterministic)!
  - Programs calculate as purely functional programs, but incrementally
  - In most programs, the nondeterministic part can be limited to a small part of the program, keeping the advantages of declarative concurrency for the most part
  - The default execution strategy is eager evaluation, but lazy evaluation is also supported with explicit declarations
- Message-passing concurrency
  - Ideal for programming multi-agent systems
  - Programming techniques of Erlang can be used



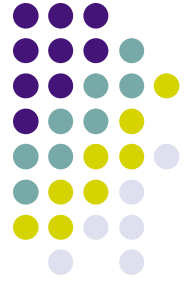
# Inferencing

- Constraint programming factorizes three concepts
  - **Distribution strategy**: what constraints define the search tree (problem dependent)
  - **Search strategy**: how the search tree is traversed (problem independent)
  - **Constraint domain**
- Supports full compositionality, concurrency, laziness, and higher-order programming techniques
  - Important for structuring large programs
- Supports four constraint domains: integers (finite domains), finite sets, rational trees, real intervals
- Logic programming
  - A special case of constraint programming; supports rational trees, Andorra style nondeterminism, and deep guards
  - In my view, Oz is the de facto Next Logic Programming Language

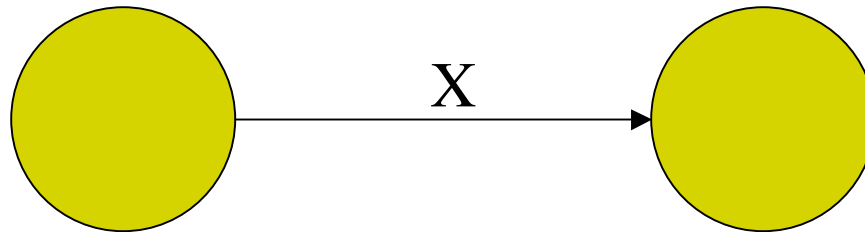


# Distribution

- **Network transparent:** same language semantics obeyed independent of distribution structure
  - Program can be distributed in various ways without rewriting the bulk of the code
- **Network aware:** efficient distributed protocols implement distributed entities
  - Language classifies entities as stateful, single assignment, and stateless
  - Language is stateless by default; using state requires explicit declaration
- **Open:** any language reference can be given to any process through the concept of “ticket”
- **Robust:** failure detection within the language



# Stream Communication with Dataflow Concurrency



$X = \text{all} \mid \text{roads} \mid Y$

$Y = \text{lead} \mid \text{to} \mid Z$

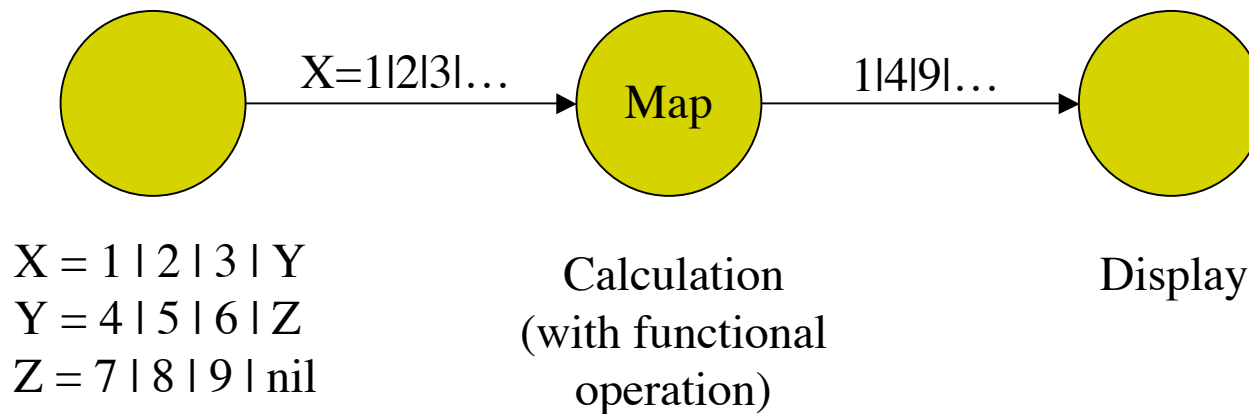
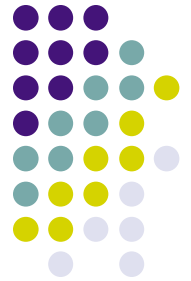
$Z = \text{alexandria} \mid \text{nil}$

Display

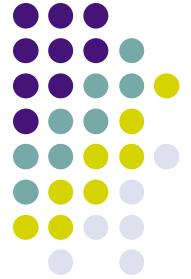
(with Browse tool)

- There are two threads
- The first thread creates the stream  $X$  incrementally
- The second thread displays it using dataflow
- Transmission is **asynchronous** (like a pipe)

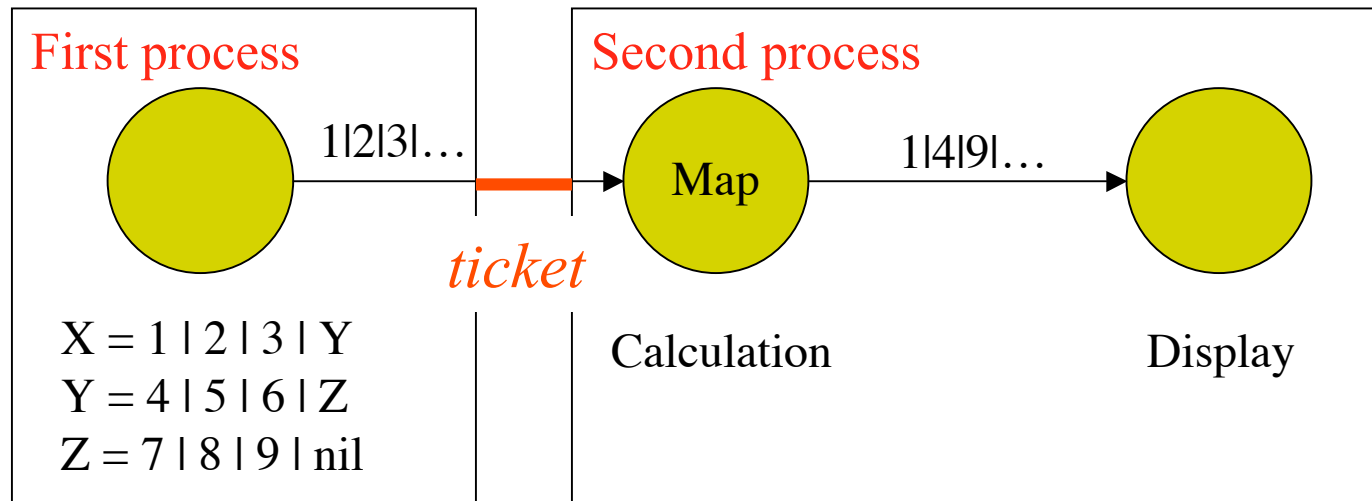
# Stream Communication with Dataflow Concurrency



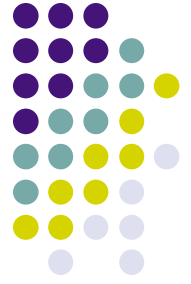
- There are three threads
- The first thread creates a stream of data
- The second thread does a calculation
- The third thread displays the results



# Stream Communication with Dataflow Concurrency



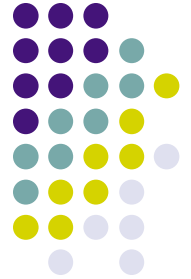
- Exactly the same thing, but *distributed*
- The processes connect through a *ticket*
  - A ticket is a reference that can exist outside of a process (since it is coded as an Ascii string)
- Except for the ticket, the program is unchanged



# Graphical User Interfaces

- Mixed declarative/imperative approach to GUI design, as realized in the QtK tool
  - Declarative specification is a data structure; can be manipulated easily
    - Interface structure, widget types, initial state, resize behavior
  - Imperative specification is a program; has maximum expressiveness
    - Action procedures and handler objects embedded in the data structure
- This makes creating **dynamic interfaces** easy

# Example User Interface



*Nested record with handler object E and action procedure P*

```
W=td(lr(label(text:«Enter your name»)
        entry(handle:E) )
      button(text:«Ok» action:P))
```

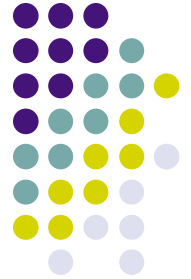
*Construct interface (window & handler)*

```
...
{Qt.build W}
```

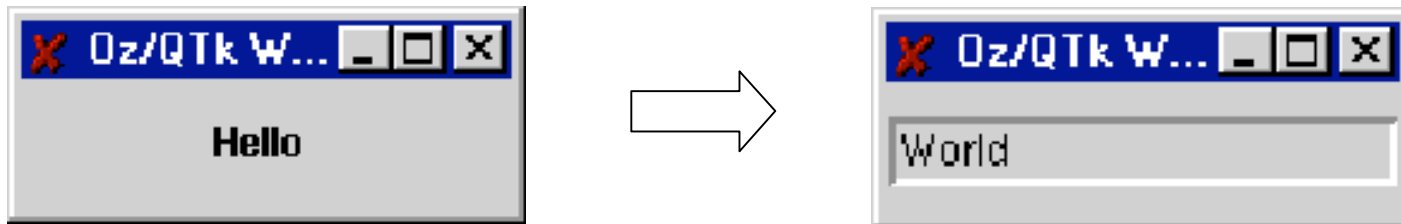
```
...
{E set(text:«Type here») }
```

*Call handler*

```
...
Result={E get(text:$) }
```



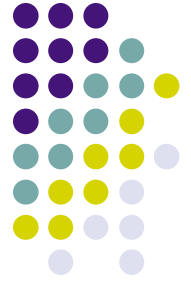
# Example Dynamic Interface



```
W=placeholder(handle:P)
...
{P set(label(text:«Hello»)}
...
{P set(entry(text:«World»)}
}
```

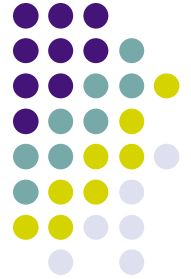
- Any GUI specification can be put in the placeholder at run-time
- GUIs specifications are data structures; can be calculated

# The Right Default: Concurrent Components



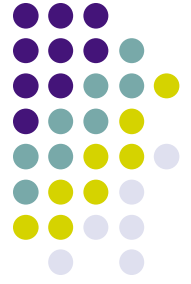
- In our experience, the right default for structuring programs is as concurrent components that communicate through asynchronous message passing
  - Not a new idea; Carl Hewitt anticipated it **30 years ago!** But now we understand it very well.
- Unfortunately, this is almost universally ignored by mainstream languages and books on program design
  - Object-oriented programming and shared-state concurrency are given priority, even though they are the **wrong default**

# Mozart Support for Concurrent Components



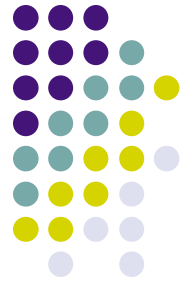
- The Mozart platform supports this default well
  - Ultralightweight threads
  - Support for message-passing concurrency
  - Note that there are no built-in monitors in Oz!
    - Isn't it strange? Not at all!
- We strongly recommend that you take advantage of this support to structure your applications!

# Three Reasons Why Concurrent Components are the Right Default

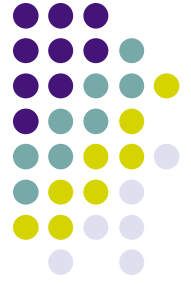


1. The simplest paradigms for concurrent programming are declarative concurrency and message-passing concurrency
  - Compare *Concurrent Programming in Erlang* with *Concurrent Programming in Java*
2. The **Erlang** language and system is used successfully for building highly available systems; for fundamental reasons it uses this model
3. The **E** language and system is used successfully for building secure distributed systems; again for fundamental reasons it uses this model

# The Exaggerated Importance of Object-oriented Programming



- Consider for example the task of building robust telecommunications systems
- Ericsson has developed an extremely reliable ATM switch (the AXD 301) using a message-passing architecture
- The important concepts are **isolation**, **concurrency**, and **higher-order programming**
- Not used: **inheritance**, **classes**, **methods**, **UML diagrams**, and **monitors**
  - Maybe these concepts are not the essential ones?



# Formal Semantics

- Oz has a simple formal semantics
  - Well factored Structural Operational Semantics for the complete Oz kernel language (e.g., chapter 13 of Oz book)
  - Operational semantics is subsettable for all paradigms
  - Other semantics (denotational, axiomatic, logical) for particular subsets that make this easy
- Mozart is designed to implement the semantics
  - The mere existence of a simple semantics, even if ignored by the programmer, means that there are no unexpected surprises
  - The system is both simple and highly expressive

# Programming Languages and Paradigms



## Declarative programming

*strict functional programming, e.g., Scheme, ML*  
*deterministic logic programming*

+ **concurrency**  
+ **by-need synchronization**  
*declarative concurrency*  
*lazy functional programming, e.g., Haskell*

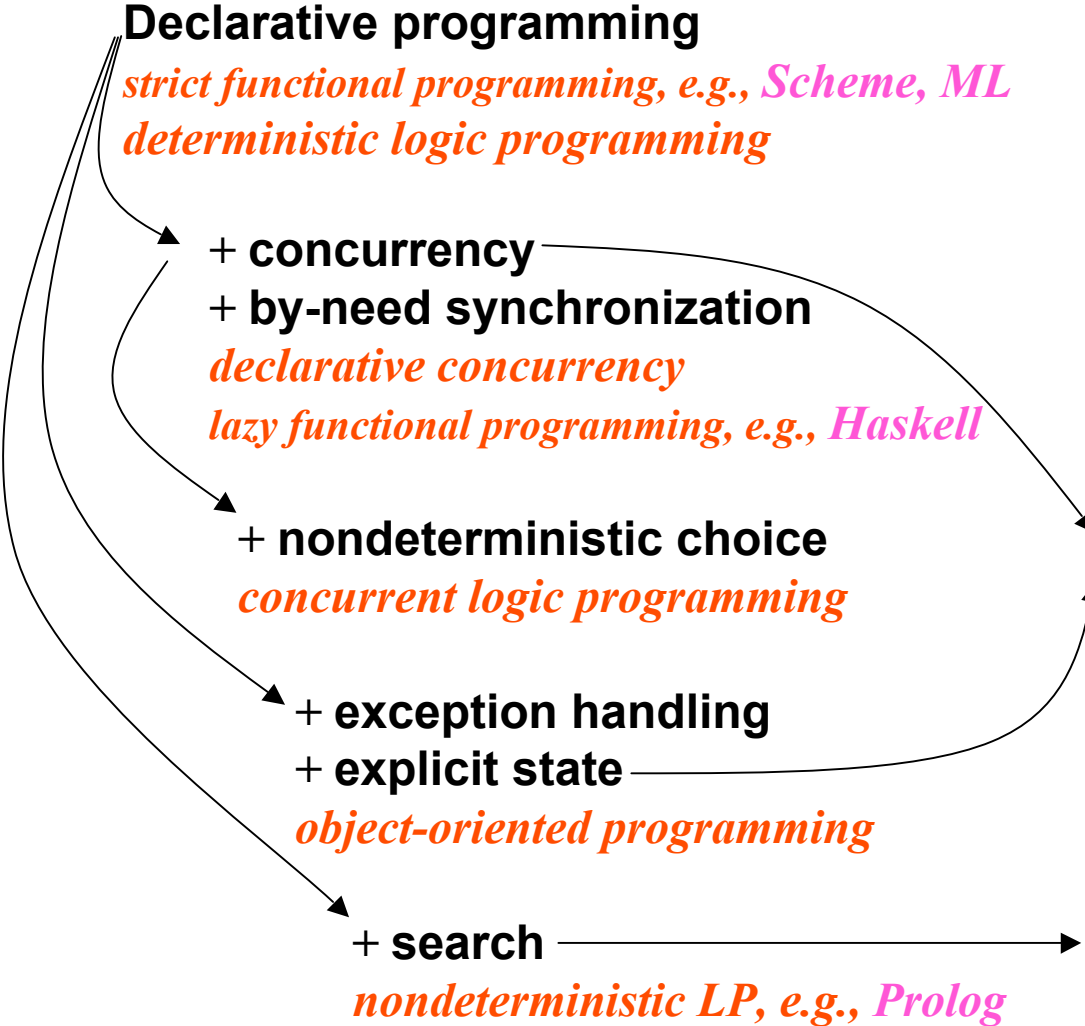
+ **nondeterministic choice**  
*concurrent logic programming*

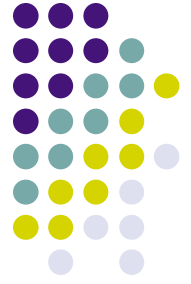
+ **exception handling**  
+ **explicit state**  
*object-oriented programming*

+ **search**  
*nondeterministic LP, e.g., Prolog*

*concurrent OOP*  
*(active object style, e.g., Erlang)*  
*(shared state style, e.g., Java)*

+ **computation spaces**  
*constraint programming*





# Conclusions

- The Mozart/Oz platform covers a lot of ground in a coherent way
- We hope that MOZ 2004 inspires you to cover more of this ground and take advantage of what Mozart/Oz offers
- Enjoy the conference!