Spacetime Programming
Synchron 2016

Pierre Talbot    Carlos Agon    Philippe Esling
(talbot@ircam.fr)

Institute for Research and Coordination in Acoustics/Music (IRCAM)  
University Pierre et Marie Curie (UPMC)

5th December 2016
Menu

- Introduction
- Spacetime programming
- Implementation
- Conclusion
Constraint programming

Holy grail of computing

- Declarative paradigm for solving combinatorial problems.
- We state the problem and let the system solve it for us.
Successful paradigm

Applications

It has a lot of different applications ranging from Sudoku solving, scheduling, packing, musical orchestration...
How to find a solution?

**NP-complete nature**

- Try every combination until we find a solution.
- The possible combinations are represented in a tree.
Problem

Holy grail?

- Search tree is often too huge to find a solution in a reasonable time.
- **Search strategies are crucial** for describing how to create and prune the tree and improving efficiency.
- Search strategies are often problem-dependent so we need to try and test (empirical evaluation).
State-of-the-art

1. **Languages** (Prolog, MiniZinc,...): Clear and compact description but limited amount of pre-defined strategies.

2. **Libraries** (Choco, GeCode,...): Highly customizable and efficient but complex software, hard to understand and time-consuming.

   ▶ Composing strategies is impossible or hard in both cases.

**Lack of abstraction** for expressing, composing and extending search strategies.
Synchronous languages provide the needed abstraction!

- We propose *spacetime programming*, a language abstraction for expressing search strategies.
- Based on Esterel (without the reaction to absence).
- **Execution**: One node of the tree processed per instant.
- **Nondeterministic** operator for specifying the branches of the tree.
Menu

▶ Introduction

▶ Spacetime programming

▶ Implementation

▶ Conclusion
Spacetime programming

Spacetime programming = Synchronous programming + Search strategy.

- Search strategies as synchronous processes.
- **Composition** of strategies with the parallel operator.
  - `par s₁ || s₂ end`
  - Easy experiment: plugging in and out strategies.
- **Communication** between strategies in the deterministic framework of the synchronous paradigm.
Synchronous programming

- In one instant, a synchronous program reacts to inputs and emits outputs.
- It keeps an internal state of variables and program status.

How to link the synchronous model and search tree?
The search tree is represented as a queue of nodes. We feed the program with **one node of the tree per instant**. The synchronous program fuels the queue with new nodes.

**Inputs**
- dequeue 1 node

**Outputs**
- push 0 to N nodes

**Internal state**

**Synchronous program**

**Queue of the nodes**
space p || q end for creating two branches where \( p \) and \( q \) describes children nodes.

```plaintext
let x = [0..10];
loop
  let mid = middle_value(x);
space
  || x ← [lb(x)..mid−1]
  || x ← [mid..ub(x)]
end
pause
end
```
We can use the internal state for maintaining global information to the tree.
For example, for maintaining statistics such as the number of nodes explored.

```plaintext
count_nodes ≡
    nodes ← 1;
loop
    pause;
    nodes ← (pre nodes) + 1;
end
```
Spacetime attribute

Problem

How to differentiate between variables in internal state and onto the queue?

We use a spacetime attribute to situate a variable in space and time.

- **Global**: Variable in one location, global to the search tree (attribute single_space).
- **Local**: Variable in one time, local to one instant (attribute single_time).
- **Backtrackable**: Variable in the queue of nodes (attribute world_line).
let x in world_line = [0..10];
loop
  let mid in single_time = middle_value(x);
  space
  || x ← [lb(x)..mid−1]
  || x ← [mid..ub(x)]
end
pause
end
Variables are complete lattices

- Every variable is a complete lattice where $\leftarrow$ is the join operator and `bot` the bottom representing the lack of information.
- `transient` re-initializes the value to bottom between instants (persistent by default).

```plaintext
let transient nodes = bot;

count_nodes ≡
  nodes ← 1;
loop
  pause;
  nodes ← (pre nodes) + 1;
end
```

\[
\begin{array}{c}
\top \\
0 & 1 & \ldots & n \\
\bot
\end{array}
\]
Menu

- Introduction
- Spacetime programming
- Implementation
- Conclusion
Implementation: Bonsai

- Integration into object-oriented language (Java).
- Extend the Java syntax with processes and reactive attributes.
- [github.com/ptal/bonsai](https://github.com/ptal/bonsai)

Compilation

The compiler acts as a preprocessor from Bonsai to Java.

```
SugarCubes runtime

↓

.bonsai.java → .java → JVM
```
SugarCubes (Susini, ’01)

SugarCubes is a Java library to program reactive systems with the synchronous paradigm.

- It provides a set of class combinators for each synchronous instructions.
- For example, \texttt{loop \{ pause; \}} is compiled to \texttt{new Loop(new Pause())}.
- Method \texttt{activate()} called at each instant on the combinators.
Must inherits from Executable and have a process named execute (entry point).

Java method call with ~method.

```java
public class ConstraintProblem implements Executable {
    world_line VarStore domains = bot;
    world_line ConstraintStore constraints = bot;
    proc execute() {
        ~modelChoco(domains, constraints);
        par branching() || propagate() end
    }
    private static void modelChoco(VarStore domains,
        ConstraintStore constraints )
    { ... }
}
```
A runtime environment contains all the variables.

Programs are created at runtime.

```java
public Program execute() {
    return SC.seq(
        new JavaAtom((env) -> {
            VarStore domains = (VarStore) env.var("domains");
            ConstraintStore constraints = (ConstraintStore) env.var("constraints");
            modelChoco(domains, constraints);
        }),
        SC.par(
            branching(),
            propagate()
        )
    );
}
```
Experiments

We validate this approach by replacing the search module of the state-of-the-art constraint solver Choco and comparing the efficiency.

- We provide a small binding (200 loc) to be able to use Choco inside the language.
- We implemented the same search strategy in Choco and in Bonsai.
- Comparison on 3 different constraint problems.
## Experiments

<table>
<thead>
<tr>
<th></th>
<th>Choco</th>
<th>SP</th>
<th>$\frac{SP}{Choco}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First solution</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latin square (40)</td>
<td>3.42 s</td>
<td>3.45 s</td>
<td>1</td>
</tr>
<tr>
<td>Latin square (50)</td>
<td>8.26 s</td>
<td>9.66 s</td>
<td>1.17</td>
</tr>
<tr>
<td>Latin square (60)</td>
<td>19.49 s</td>
<td>23.20 s</td>
<td>1.19</td>
</tr>
<tr>
<td><strong>All solutions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-Queens (12)</td>
<td>1.44 s</td>
<td>3.62 s</td>
<td>2.51</td>
</tr>
<tr>
<td>N-Queens (13)</td>
<td>6.35 s</td>
<td>16.04 s</td>
<td>2.53</td>
</tr>
<tr>
<td>N-Queens (14)</td>
<td>32.10 s</td>
<td>147 s</td>
<td>4.58</td>
</tr>
<tr>
<td><strong>Best solution</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Golomb ruler (9)</td>
<td>0.57 s</td>
<td>1.61 s</td>
<td>2.83</td>
</tr>
<tr>
<td>Golomb ruler (10)</td>
<td>1.69 s</td>
<td>6.43 s</td>
<td>3.81</td>
</tr>
<tr>
<td>Golomb ruler (11)</td>
<td>24.89 s</td>
<td>135 s</td>
<td>5.42</td>
</tr>
</tbody>
</table>

> Almost no overhead for finding one solution, factor between 2 and 5 for all and best solution.
Conclusion

- Lack of an abstraction for expressing search strategies.
- Synchronous language is an ideal abstraction when extended with:
  - Partial information (lattice-based variable).
  - Nondeterminism.
- Working implementation available.
- Experiments show an acceptable overhead compared to state-of-the-art solvers.

github.com/ptal/bonsai
Future work

- **Static analysis** for avoiding the top value.
- **Interactive constraint system.**
  - Computer-aided composition with constraints.
- Queue of nodes directly accessible in the program.
  - Enables restart-based search strategies such as iterative deepening, limited discrepancy, ...
Thank you for your attention.