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

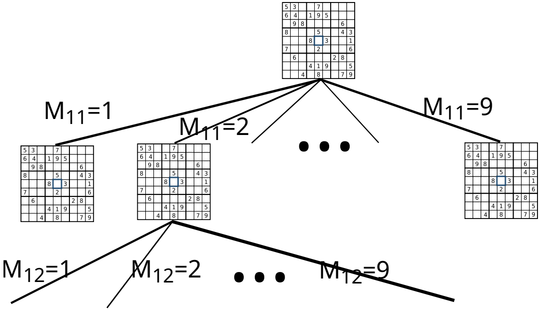
Applications

It has a lot of different applications ranging from Sudoku solving, scheduling, packing, musical orchestration...

How to find a solution?

NP-complete nature

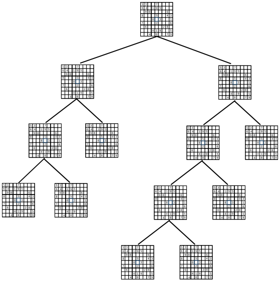
- Try every combination until we find a solution.
- The possible combinations are represented in a tree.



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.

Proposal

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

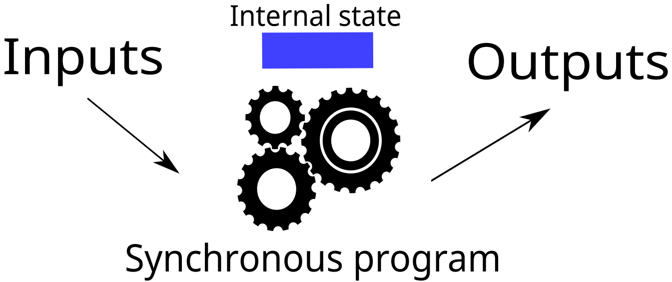
Spacetime programming

Spacetime programming = Synchronous programming + Search strategy.

- Search strategies as synchronous processes.
- **Composition** of strategies with the parallel operator.
  - `par` s_1 `||` s_2 `end`
  - Easy experiment: plugging in and out strategies.
- **Communication** between strategies in the deterministic framework of the synchronous paradigm.

Synchronous programming

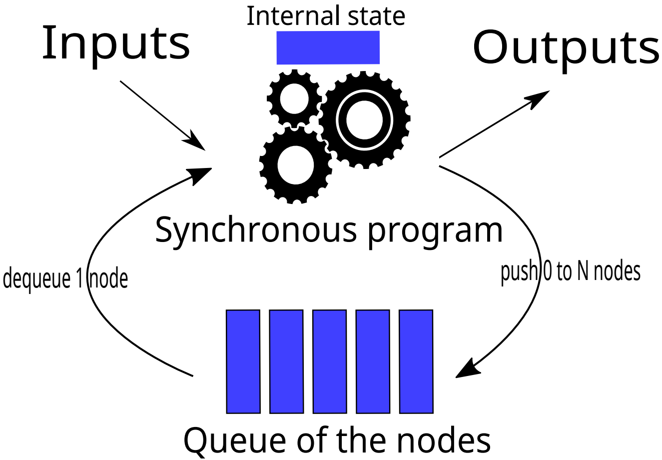
- In one instant, a synchronous program reacts to inputs and emits outputs.
- It keeps an internal state of variables and program status.



How to link the synchronous model and search tree?

Spacetime execution scheme

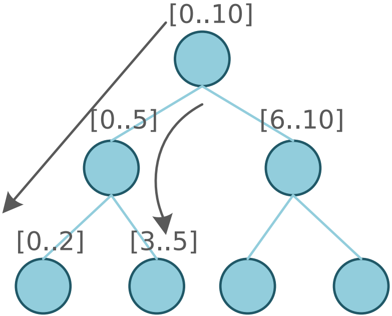
- The search tree is represented as a queue of nodes.
- We feed the program with **one node of the tree per instant**.
- The synchronous program fuels the queue with new nodes.



Space: Creating the tree

- **space** p || q **end** for creating two branches where *p* and *q* describes children nodes.

```
let x = [0..10];
loop
  let mid = middle_value(x);
  space
   || x ← [lb(x)..mid−1]
   || x ← [mid..ub(x)]
  end
  pause
end
```



Internal state

- 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.

```
count_nodes $\equiv$
  nodes $\leftarrow$ 1;
  loop
    pause;
    nodes $\leftarrow$ (pre nodes) + 1;
  end
```

Spacetime attribute

Problem

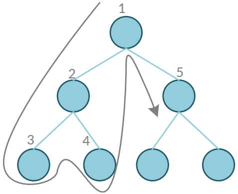
How to differentiate between variables in internal state and onto the queue?

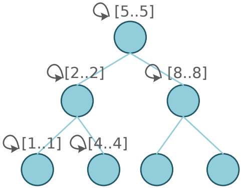
We use a spacetime attribute to situate a variable in space and time.

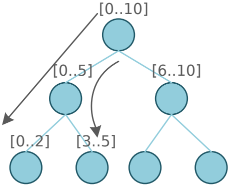
- **Global**: Variable in one location, global to the search tree (attribute `single_space`).
- **Local**: Variable in one time, local to one instant (attribute `single_time`).
- **Backtrackable**: Variable in the queue of nodes (attribute `world_line`).

Spacetime attribute

```
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
```





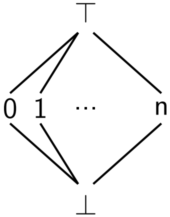


Variables are complete lattices

- Every variable is a complete lattice where ← 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).

```
let transient nodes = bot;

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



Menu

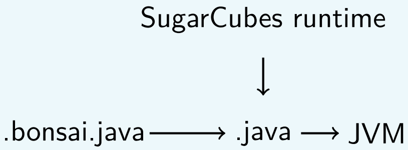
- Introduction
- Spacetime programming
- Implementation
- Conclusion

Implementation: Bonsai

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

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, `loop { pause; }` is compiled to `new Loop(new Pause())`.
- Method `activate()` called at each instant on the combinators.

Bonsai syntax

- Must inherits from `Executable` and have a process named `execute` (entry point).
- Java method call with `~method`.

```
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 )
  {  ...  }
}
```

Compilation

- A runtime environment contains all the variables.
- Programs are created at runtime.

```
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

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

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

Menu

- Introduction
- Spacetime programming
- Implementation
- Conclusion

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.