Interactive constraints computer-aided composition

ICMC 2017

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

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

19th October 2017

Computer-aided composition

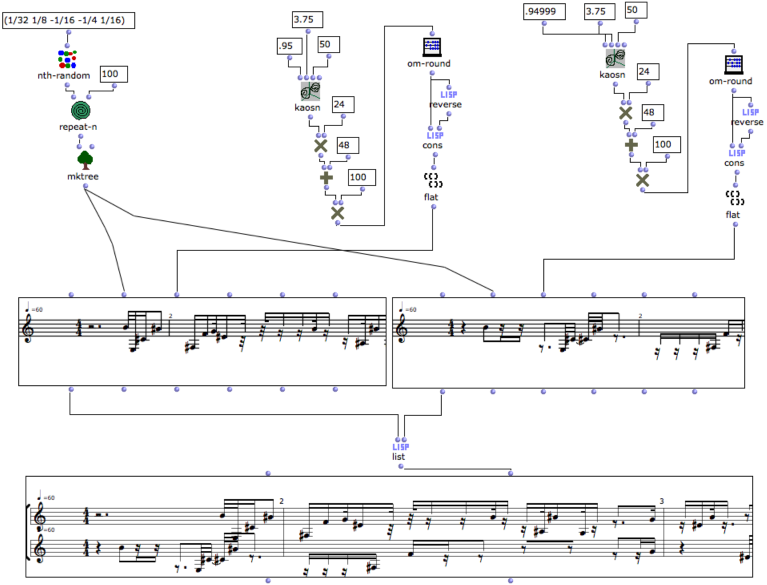
Goals

- Delegating tedious computations to the machine.
- Parametrizing the patch with values to quickly try-and-test.
- . . .

How does the composer interact with the machine?

- Mostly visual and dataflow programming languages: OpenMusic, PureData, Max,. . .
- Functional programming languages for the specifics: Lisp mostly.

Dataflow: a patch in OpenMusic



Constraints in computer-aided composition

Constraint programming

- Declarative paradigm for solving combinatorial problems.
- We state the problem and let the system solve it for us.
- Example: pitches must form a decreasing sequence (from highest to lowest).

Some examples of attempts to add constraints into CAC softwares:

- PWConstraints on top of PatchWork: constraints over the pitches, grouping the pitches together (modelling aspects).
- OMCloud on top of OpenMusic is based on a different constraint solving paradigm—local search—aiming at the ease of use.

Problem

- CAC softwares extended with constraints work in **black box**: one solution gets out of the box.
- But constraints are relations, not functions.
- Therefore, a constraint problem can have zero, one or many solutions.

By functionalizing the constraint process, we miss a key point:

Constraints are useful to describe a class of solutions

but how to work with many solutions?

Proposal: Interactivity

Experiment with an interactive constraint score editor.

- Bring the composer at the level of the solving process.
- He can consciously choose a solution.
- Development of an interactive search strategy to navigate in the **solution space**.

Proposal: Spacetime programming

Interactivity and search strategies is a deeper problem: constraint solvers also work in a “black box” mode.

We propose the process calculi spacetime programming.
SP = constraint programming + synchronous paradigm.

Spacetime programming

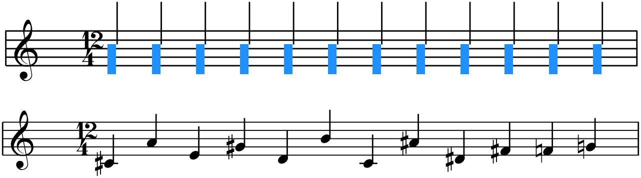
- Synchronous programming for *interactive computing*.
- A search strategy is viewed as a process: abstraction over the constraint solver.

Menu

- Introduction
- Interactivity in solvers
- Interactivity in CAC
- Conclusion

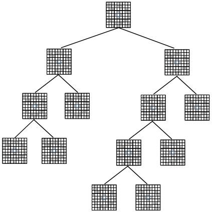
All-interval series: a MiniZinc model

```
int: n = 12;
array[1..n] of var 1..n: pitches;
array[1..n-1] of var 1..n - 1: intervals;
constraint forall(i in 1..n - 1)
                    ( intervals [i] = abs(pitches[i+1] - pitches[i]));
constraint alldifferent (pitches);
constraint alldifferent ( intervals );

solve satisfy;
```



All-interval series: a MiniZinc model

```
int: n = 12;
array[1..n] of var 1..n: pitches ;
array[1..n−1] of var 1..n − 1: intervals;
constraint forall(i in 1..n − 1)
                  ( intervals [ i ] = abs(pitches[ i+1] − pitches[i ]));
constraint alldifferent ( pitches );
constraint alldifferent ( intervals );

solve satisfy;
```

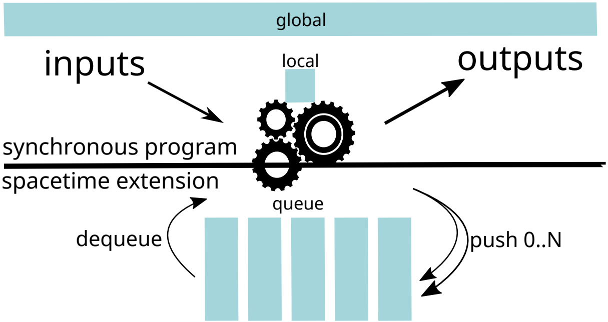


Synchronous paradigm

- Invented in the 80s to deal with reactive system subject to many (simultaneous) inputs.
- Continuous interaction with the environment.
- Mainly used in embedded systems.

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.



Spacetime programming

Syntax

$\langle p, q, \ldots \rangle$ ::=		**communication fragment**
$\mid$	*spacetime Type* $x = e$	(variable declaration)
$\mid$	**when** *cond* **then** *p* **end**	(ask)
$\mid$	*x* **<-** *e*	(tell)
$\mid$	$x.m(\ldots)$	(method call)
$\mid$		**synchronous fragment**
$\mid$	**par** *p* **||** *q* **end**	(parallel composition)
$\mid$	*p* **;** *q*	(sequential composition)
$\mid$	**suspend when** *cond* **in** *p* **end**	(suspension)
$\mid$	**loop** *p* **end**	(infinite loop)
$\mid$	**pause**	(delay)
$\mid$		**search tree fragment**
$\mid$	**space** *p* **end**	(branch creation)
$\mid$	**prune**	(branch pruning)

Spacetime attribute

Problem

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

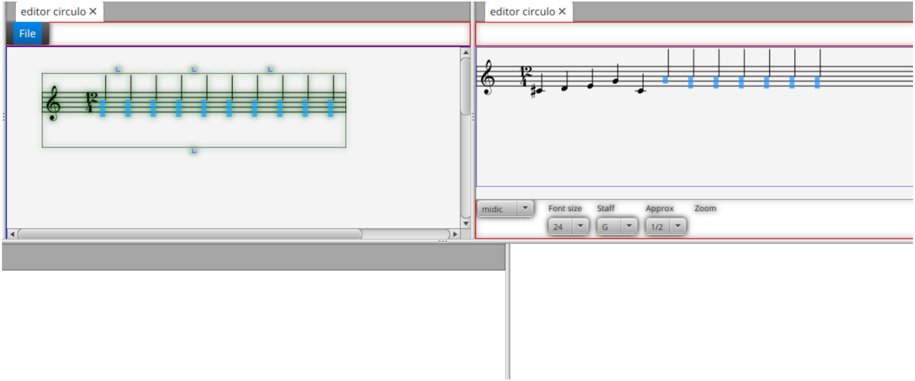
We use a spacetime attribute to situate a variable in space and time.

- `single_space`: variable **global** to the search tree.
- `single_time`: variable **local** to one instant.
- `world_line`: **backtrackable** variable in the queue of nodes.

Menu

- Introduction
- Interactivity in solvers
- Interactivity in CAC
- Conclusion

Score editor: overview



Font size 24

Staff G

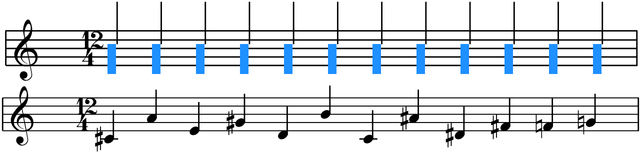
Approx 1/2

Constraint solving zone for the interactions with the composer.

A first interactive strategy

The strategy usually implemented in CAC with constraints: stop at each solution. In practice: click on “space” to jump to the next solution.

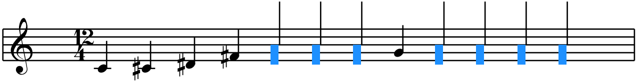
```
class EachSolution {
  world_line VStore domains = bot;
  world_line CStore constraints = bot;
  proc stop_at_solution =
    loop
      par
      || when domains |= constraints then stop end
      || pause
      end
    end
}
```



Interaction with the composer

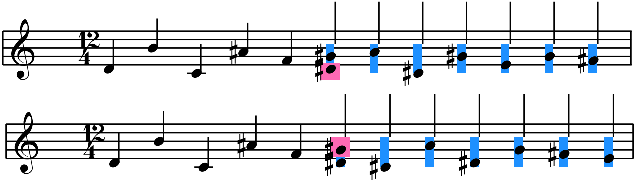
The composer interacts with the search in-between instants.
The spacetime attributes enable interactions with the search in two main ways: globally or only for the current search path.

```
class PSolver {
  world_line CStore constraints = bot;
  single_space CStore cpersistent = bot;
   ...
}
```



Lazily navigating the solution space

The next two scores represent a choice between ♯*D* and ♯*G* on the sixth note:



```
SubSolver<RBinary, Model> left = new SubSolver();
SubSolver<Binary, Model> right = new SubSolver();
single_time L<Boolean> choice = bot;
choice <- top;
par
|| suspend when choice |= true then right.search() end
|| suspend when choice |= false then left.search() end
end
```

Menu

- Introduction
- Interactivity in solvers
- Interactivity in CAC
- Conclusion

Constraints in music

From a computer scientist perspective

- Probably not for generating music: machine learning methods do it better.
- Reasoning on a class of scores satisfying some properties.
  Example: we are not forced to write a particular pitch but a class of pitches satisfying some rules.
- Constraints do not force the composer to make any choice!

Conclusion

- Constraints are relational: interactive search helps to use them in this way.
- To program interactive search strategies, we use spacetime programming.

Future work

- Current prototype with AIS only; enabling any MiniZinc model.
- This would allow composers to try the system and to develop more strategies.

Stay tuned! `github.com/ptal/bonsai`
`github.com/ptal/repmus`

Thank you for your attention.

Stay tuned! github.com/ptal/bonsai
github.com/ptal/repmus