Interactive computer-aided composition with constraints

Pierre Talbot, Carlos Agon, Philippe Esling

Interactive computer-aided composition with constraints

Pierre Talbot **Carlos Agon** **Philippe Esling**
Institute for Research and Coordination in Acoustics/Music (IRCAM)
Université Pierre et Marie Curie (UPMC)
CNRS (UMR 9912 STMS)
{talbot,agonc,esling}@ircam.fr

ABSTRACT

Computer-aided composition systems enable composers to write programs to generate and transform musical scores. In this sense, constraint programming is appealing because of its declarative nature: the composer constrains the score and relies on a solver to automatically provide solutions. However, the existing constraint solvers often lack interactivity. Specifically, the composers do not participate in the selection of a particular solution. We propose an interactive search algorithm that enables the composer to alter and to navigate in the solution space. To this aim, we propose spacetime programming, a paradigm based on lattices and synchronous process calculi. The result is an interactive score editor with constraint support, and we experiment it on the all-interval series problem.

1. INTRODUCTION

Computer-aided composition is a routine for many composers, as attested by numerous tools including OpenMusic [1] and Max/MSP [2]. It enables the composer to delegate tedious computation to the machine, such as generating rhythms for non-overlapping voices of a score. The computation is usually displayed in visual programming languages based on the functional paradigm. In this paradigm, the data "flows" in a tree structure where nodes (named "boxes") encapsulate computation on data. If a functionality is missing in the available precoded boxes, the composer must implement it with a "lower-level" programming language, such as Lisp in OpenMusic. However, these programming languages are less intuitive for composers than visual languages. This is why other paradigm, such as constraint programming, are investigated.

Constraint programming is a declarative paradigm where programmers only need to declare the structure of a problem with constraints. The resolution process is left to a dedicated solver (see Section 2.2). It has been applied to model multiple aspects of music theory, such as harmony, rhythm and orchestration [3]. There are several systems integrating constraint programming in computer-aided composition softwares [4]. In particular, PWConstraint [5] is one of the first systems that integrates constraint solving under a visual composition environment. Another approach is OMCloud [6] that is based on a non-exhaustive constraint solving technique called *local search*. Generally, merging the constraint and functional paradigms is done by encapsulating constraint solving into a box where the parameters are inputs to the constraint satisfaction problem (CSP) and the output is a solution to this CSP.

Lack of interactivity A CSP can have from zero (in case of unsatisfiability) to multiple solutions. Despite the relational nature of constraints, the existing systems view a CSP as a function. Therefore, the solution chosen by the solver is unpredictable and the composer does not participate in the selection of this particular solution. Besides, this process is not replicable: the first solution may change with the solver search strategy, parameters or when the solver is simply updated.

On the contrary, a CSP can be over-constrained with no solution. In this case, a common method is to use soft constraints: the system tries to satisfy as many constraints as possible. It is similar to the problem with multiple solutions because many "soft solutions" are possible. In summary, current approaches lack interactivity between the composer and the constraint solver for selecting the solution.

Interactive score editor To solve this problem, we suggest a score editor in which the composer can visualize partially instantiated scores and steer the solving process toward a customized solution (Section 4). We propose several interactive strategies for navigating in the solution space to help the composer consciously select a solution (Section 5). However, the existing abstractions inside constraint solvers are not tailored for interactivity. To solve this problem, we introduce spacetime programming, a paradigm based on the synchronous paradigm, to facilitate interactions between the composer and the solver (Section 3). The synchronous paradigm is the key to support interactive solving (Section 2.1). We experiment the system [1] with the all-interval series problem (Section 4). The result is an interactive score editor with constraint solving as a part of the composition process.

[1] A prototype is available at github.com/ptal/repmus.

2. BACKGROUND

2.1 Synchronous programming

The synchronous paradigm [7] was initially designed for modeling systems reacting to simultaneous events of the environment (different inputs can arrive at the same time) while avoiding typical issues of parallelism, such as deadlock or indeterminism. The main idea of this paradigm is to divide the execution of a program into a sequence of discrete instants conceptually instantaneous (named as *synchrony hypothesis*). It enables strong static analysis to ensure the cooperative and correct behaviour of the processes reacting to external stimuli. A simple example is a watch: its state changes when the user presses buttons or when it receives a signal indicating that a second has elapsed.

In the rest of this paper, we consider the synchronous language Esterel [8]. An Esterel program reacts to boolean input signals and produces outputs. We introduce the ABO example (variant of the standard ABRO example [8]): when the signals (i.e. boolean events) A and B are received, the output O is emitted. The signal O carries an integer value indicating the number of times it has been emitted.

```
module ABO:
  input A, B;
  output O := 0: integer;
  loop
    [ await A || await B ];
    emit O(pre(?O) + 1);
    pause;
  end loop
end module
```

The statement `await A` indefinitely waits for the signal A and terminates when A occurred. The parallel composition P `||` Q allows the concurrent execution of two processes and terminates when both are terminated. This operator does not offer multithreading and is compiled into sequential code—the processes are statically interleaved to ensure determinism. In this example, the parallel statement terminates as soon as the signals A and B have occurred. Once terminated, the signal O is emitted and can be retrieved by the user to activate, for example, a real world command. We initialize the value of O to 0 and increment it when emitted; `pre(?O)` is used to retrieve the value of O in the previous instant. The primitive instruction `loop` p `end loop` executes indefinitely the process p. We forbid the body of the loop to be instantaneous due to the synchrony hypothesis and therefore, we delay each iteration to the next instant with the statement `pause`. The whole behavior is reset at each loop iteration.

A synchronous program is a coroutine: a function that can be called multiple times and that maintains its state between calls. When called, the code of the coroutine is resumed from the last `pause` statements reached. Also, due to this temporal dimension, it is composed of two memories: (i) the global memory for the values spanning several instants, such as the valued signal O, and (ii) the local memory for those only relevant to a single instant—the case of the signals A and B. As shown in the upper part of Figure 1, user inputs can be injected into the program in-between two calls. The external inputs are collected by a host language from which the synchronous program is called.

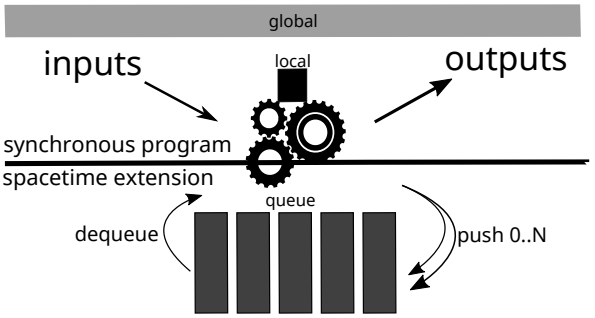


Figure 1: Spacetime extension of the synchronous model of computation.

The well-defined semantics of synchronous languages for the treatments of simultaneous events has a wide variety of applications encompassing interactive mixed music [9] and web programming [10], just to name a few of them.

2.2 Constraint programming

Constraint programming is a declarative paradigm for solving CSP [11]. A CSP is a couple $\langle d, C\rangle$ where d is a function mapping variables to sets of values (the domain) and C is a set of relations constraining these variables (the constraints). The goal is to find a solution: a set of variables reduced to a singleton domain such that every constraint is satisfied. In practice, it interleaves: (i) a propagation step, removing values from the domains that do not satisfy at least one constraint and (ii) a search step making a choice when "we don't know" and backtracking to another choice if the former did not lead to a solution. The choice made for creating the children nodes is referred to as the branching strategy—it describes the branches leading to these nodes. The successive interleaving of choices and backtracks leads to the construction of a search tree that can be explored with various search strategies. In this paper, we exclusively focus on the search step and delegate the propagation step to the specialized constraint solver Choco [12] (more information on the propagation theory can be found in e.g. [13]).

3. SPACETIME PROGRAMMING

We propose spacetime programming, a language where variables are complete lattices and the program statements are monotonic functions over these variables. The language semantics is based on lattice theory and we introduce several definitions based on [14] before describing the language.

3.1 Definitions

An ordered set $\langle D, \leq\rangle$ is a complete lattice if every subset of $S \subseteq D$ has both a least upper bound and a greatest lower bound. A complete lattice is always

```
class Solver<Brancher, Model> {
  world_line VStore domains = bot;
  world_line CStore constraints = bot;
  Brancher brancher = new Brancher();
  Model model = new Model();
  proc solve =
    par
    || model.define()
    || brancher.branch()
    || propagation()
    end
  proc propagation = loop
    constraints.propagate(domains);
    pause;
  end
}
```

```
class Binary {
  ref world_line VStore domains;
  ref world_line CStore constraints;
  proc branch =
    loop
      when not (domains |= constraints) then
        single_time IntVar x = input_order(domains);
        single_time Integer v = middle_value(x);
        space constraints <- x.gt(v) end
        space constraints <- x.le(v) end
      end
      pause;
    end
}
```

Figure 2: A minimal constraint solver in spacetime.

bounded: it exists a supremum $\top \in D$ such that $\forall x \in D.x \leq \top$ and an infimum $\bot \in D$ such that $\forall x \in D.\bot \leq x$. Alternatively, we can view a lattice as an algebraic structure $\langle L; \vee; \wedge \rangle$ where the binary operation $\vee$ is called the join and $\wedge$ the meet. The join $x \vee y$ is the lower upper bound of the set $\{x, y\}$ and the meet $x \wedge y$ its greatest lower bound. In the following, we use the entailment operation which is the inverse ordering defined as $x \models y \stackrel{\text{def}}{=} y \leq x$ and the in-place join operator $x \leftarrow y \stackrel{\text{def}}{=} x = x \vee y$. Finally, any set S can be turned into a *flat lattice* with the following order: $\forall x \in S.\bot \leq x \leq \top$.

3.2 Model of computation

We generalize the synchronous paradigm with data defined over complete lattices and extend it to support backtracking. In Figure 1, the synchronous model of computation is endowed with a third memory named the *queue*. This queue of nodes represents the remaining search tree to explore and enables the restoration of variables upon backtracking. In each instant, one node is extracted from the queue and instantiated but several new children nodes can be created. For this purpose, we introduce the `space` b `end` statement. It pushes the node described by the process b onto the queue. This process will be executed in a future instant when the node is popped out from the queue.

A spacetime program is a set of processes working collaboratively on the exploration of the tree by synchronizing on their `pause` statements. Indeed, a new instant can only be started once every process has reached a `pause` or is terminated. We can see the `pause` statement as a barrier that must be reached by all the processes before an new instant is started.

Finally, we have three distinct memories and we introduce *spacetime attributes* for explicitly situating variables in one of them: (1) `single_time` for variables in the local memory which are re-initialized between each instant; (2) `single_space` for variables in the global memory, and (3) `world_line` for variables in the queue that must be backtracked.

3.3 Minimal constraint solver

In Figure 2, we give a spacetime program implementing a minimal constraint solver. It is constituted of two classes: `Solver` for composing the main components of the CSP and `Binary` to implement a particular branching strategy. As suggested by the syntax, the host language is Java and therefore, all variables are Java objects. They must implement a lattice interface for being usable inside the spacetime paradigm. Also, the methods are assumed to be monotonic functions over the considered lattice [2] .

The class `Solver` faithfully implements the mathematical definition of a CSP (Section 2.2) with a pair of variables $\langle$`domains`, `constraints`$\rangle$ initialized to `bot`—the bottom element of their lattices. Their types, `VStore` and `CStore` respectively, are Java classes serving as abstraction of the constraint library `Choco` [12]. The class `Solver` is parametrized by a class `Model` containing the definition of the CSP and a class `Brancher` for the branching strategy. They are both attributes and are composed with the parallel statement in the process `solve`. The process `propagation` is implemented with a Java method called on the variable `constraints` and the *loop/pause* mechanism is used to propagate in each node. The propagation is implemented by the constraint solver Choco. Similarly, the model of the problem is also implemented in Choco and is encapsulated in the class `Model` (not shown).

The class `Binary` takes references to the CSP's variables with the keyword `ref`. It implements a branching strategy for selecting the first non-instantiated variable (method `input_order`) and the value in the middle of the variable's domain (method `middle_value`). The search tree is then built with two `space` statements. The first describes the left child node in which $x > v$ and the second describes the right child in which $x \leq v$. Using the statement `when`, we ensure that children nodes are only created when we did not reach a solution or a failed node yet. The expression `domains |= constraints` is true if we can deduce the constraints from the variables' domains—which indicates a solution. This result is negated with the `not` operator.

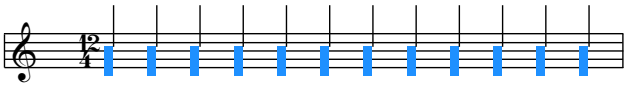
[2] Verifying that a Java method is actually monotonic is left to the implementer of the method.

4. SCORE EDITOR WITH CONSTRAINTS

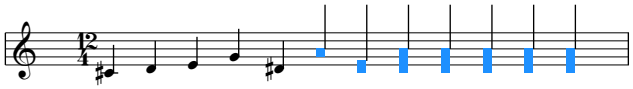
4.1 Visual constraint solving

We propose a new score editor programmed in Java. We particularly focus on the visual and interactive aspect of constraint solving. To illustrate the system, we use the all-interval series (AIS) musical constraint problem. It constrains the pitches to be all different as well as the intervals between two successive pitches. This is notably used to implement the twelve-tone technique in which every note of a pitch class has the same importance. This constraint comes built-in in our system and we leave apart its exact modeling which is, for example, covered in [3].

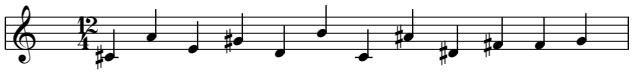
Initially, when the AIS problem is set in the editor, the pitches are initialized with domains in the interval [1..12] and are represented with rectangles:



Through the solving process, these rectangles become smaller and are displayed as a note when instantiated. For example, the following score is partially instantiated with four notes and the propagation reduces the domains of the rest of the score accordingly—the rectangles became smaller:



Experimentally, the 'space' key is pressed until a partial solution or a fully instantiated solution satisfies the composer. An example of solution given by our system to the all-interval series is:



These scores are displayed in a larger visual programming environment similar to OpenMusic. In this setting, a score is contained in a functional box which ensures the compatibility with existing methods.

4.2 Spacetime for composition

The model of the problem can be monotonically updated (adding constraints) throughout the solving process. Between instants, the composer is allowed to add new constraints into the model. We identify two different ways to add constraints interactively based on the spacetime attributes:

- *Persistent*: The constraints are added in a store with the spacetime `single_space`; they hold for the rest of the search. For example, a chord that the composer particularly likes and wants to be part of the final musical composition.
- *Contextual*: The constraints are added in a store with the spacetime `world_line` store; they hold only for the current subtree. For example, it can be an interval between two notes that only makes sense in the presence of the already instantiated notes.

To achieve this, we need to modify the solver presented in Section 3.3. We only highlight the changes here:

```
class PSolver {
  world_line CStore constraints = bot;
  single_space CStore cpersistent = bot;
  proc merge_cstore =
    loop
      constraints <- cpersistent;
      pause;
    end
}
```

We use an additional constraint store `cpersistent` that can be augmented by user constraints in between instants. In each instant, we impose these constraints in the initial `constraints` store, hence they are never "forgotten". For example, here, the composer interactively chooses to instantiate the eighth note to G:

G is added in the persistent constraint store, and will remain unchanged until the end of the search. Hence, every partial assignment or solution will contain this note.

5. INTERACTIVE SEARCH STRATEGIES

Using the spacetime paradigm, we investigate several search strategies from the most straightforward to the more complex but useful strategies.

5.1 Stop and resume the search

There are many ways to interact with a search tree during its traversal. Interacting in each node is not really interesting because the search tree is usually too large and we are not interested in every partial assignment. In most composition-aided systems, the user interacts with the search at solution nodes and, if needed, asks for the next solution. This behavior is programmed in spacetime with the following code:

```
loop
  par
  || when domains |= constraints then stop end
  || pause
  end
end
```

We introduce the statement `stop` which behaves similarly to `pause` but gives the control back to the host program. Indeed, `pause` automatically extracts a node from the queue and continue the execution without suspending the program. Hence, in this example, the search stops each time we reach a solution. Since we are in a loop, we must also ensure that we at least

```
class LazySearch<Model> {
  SubSolver<RBinary, Model> left = new SubSolver();
  SubSolver<Binary, Model> right = new SubSolver();
  // true if left, false if right.
  single_time L<Boolean> choice = bot;
  proc search =
    choice <- top;
    trap EndOfSearch in
      par
      || suspend when choice |= true then right.search() end
      || suspend when choice |= false then left.search() end
      || commit_user_choice()
      || exit_when_all_solutions()
      end
    end
  proc commit_user_choice =
    loop
      single_time Constraint c = select(left, right, choice);
      left.cpersistent <- c;
      right.cpersistent <- c;
      stop;
    end
  proc exit_when_all_solutions =
    loop
      when right.solver.domains |= left.solver.domains then
        exit EndOfSearch
      end
      stop;
    end
}
```

Figure 3: Lazy search algorithm.

pause in each iteration. When `pause` and `stop` occur at the same time, `stop` takes the priority over `pause`.

More generally, we can stop the search on any event. For example, we can be interested by a partial assignment in which a new variable has just been instantiated:

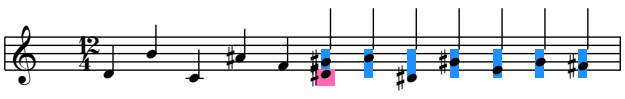
```
world_line LMax asn = new LMax(0);
loop
  par
  || asn <- domains.count_asn()
  || when asn > pre asn then stop end
  || pause
  end
end
```

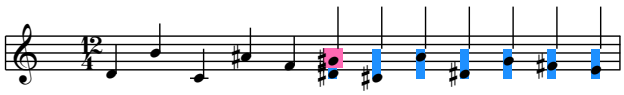
The variable `asn` represents the number of variables instantiated in `domains`. It is of type `LMax`: the lattice of increasing integers. In each node, we update this value by calling `count_asn()` on `domains`. We suspend the search whenever the current number of assignments is greater than the previous one.

5.2 Lazy search tree

A musical CSP can have many solutions, especially in the early composition of the musical piece because it is under-constrained. The solutions proposed by the strategies presented above form a catalogue in which the composer can pick one. However, their analysis by the composer is not practical and computing every solution can be time-consuming. We propose a search strategy interleaving solution generation and composer interaction. The goal is to obtain a solution that has been entirely chosen by the composer, but without exploring the full solution space.

We call it a lazy search strategy because it explores the solutions space *on-demand*. This strategy explores the score from left to right, and whenever a note can be instantiated to several pitches, the composer chooses one. For example, the next two scores represent a choice between $\sharp D$ and $\sharp G$ on the sixth note—framed with a red rectangle:





The strategy first computes a representative solution for each $\sharp D$ and $\sharp G$. It is mandatory if we want the composer to navigate in the solution space and not the full search space. To summarize, each time two or more solutions exist in a given note's domain, the system performs the following tasks: (i) it pauses the search strategy, (ii) it asks the composer for the note wanted, and (iii) it discards all the other propositions and resumes the search. The laziness comes from the fact that the search trees of the discarded solution will not be explored further.

Encapsulated search We introduce the `universe` class for encapsulating the search [15] using nested time scales [16]. A universe is a Java class with its own queue of nodes. Its full execution is nested in the current instant of the outer universe. This nested universe can interact with the outer universe with the following statements: (i) `stop` suspends the execution of the current universe in the outermost universe—which is the environment, and (ii) `pause up` suspends the current universe and gives the control back to the outer universe. For example, we can encapsulate the search performed in `PSolver` in the following class `SubSolver`; it is used to "pause up" each time we reach a solution.

```
universe class SubSolver<Brancher, Model> {
  PSolver<Brancher, Model> solver = new PSolver();
  proc search =
    par
    || solver.solve()
    || loop
       par
       || when solver.domains |= solver.constraints then
          pause up
          end
       || pause
       end
      end
    end
}
```

To implement the discussed lazy search strategy, we introduce the statement `suspend when` c `then` P. It suspends the execution of the process P whenever the

condition c is true. The algorithm is shown in Figure 3.

Firstly, we create two instances of `SubSolver`: `left` and `right` for exploring the search tree from left-to-right and right-to-left. For this purpose, we use the branching class `RBinary` which is the class `Binary` with the space statements reversed. Since we "pause up" when we find a solution, we have the left- and right-most solutions.

Secondly, the variable `choice` reflects the decision of the composer to explore the left or the right part of the solution space. Its type is `L<Boolean>` where `L<T>` is a Java class transforming any type `T` into a flat lattice. In the first instant, we need to explore both solutions, and this is why `choice` is initialized to `top` since we can deduce both `true` and `false` from top. After, we only need to explore either the left or right part of the search tree depending on the composer choice. This is implemented with the `suspend` statement which activates the left search strategy if the composer has chosen the right one, and the right one otherwise.

Third, the process `commit_user_choice` stores the composer's choices in the persistent constraint store. It has the effects of pruning the exploration of other solutions, and of preventing to backtrack beyond the user choice.

Finally, the termination criterion is implemented in the process `exit_when_all_solutions`. When it detects that both sub-solvers reached the same solution, it exits the trap `EndOfSearch`—a mechanism similar to exceptions.

6. CONCLUSION

Computer-aided composition with constraints is not often used due to the black box search process in constraint solvers. We introduced a score editor with an interactive search strategy allowing to navigate in the solution space. Hence, the composer knows clearly why a solution is chosen. With the spacetime paradigm, we are able to lazily explore the search tree, to pause and to resume the search with additional information from the composer. In addition, at any stage of the search, the partial solution can be visualized on the score and examples of possible solutions are given.

The modeling of musical constraint problems has been left apart. In the future, we want to incorporate visual modeling capabilities in our score editor that fits the interactivity of the search well. Last but not least, we will evaluate and experiment this editor with professional composers.

7. REFERENCES

[1] C. Agon, G. Assayag, O. Delerue, and C. Rueda, "Objects, time and constraints in OpenMusic," in *Proceedings of the 1998 International Computer Music Conference*, 1998.

[2] M. Puckette and D. Zicarelli, *MAX, Development Package*, Ircam and Opcode Systems, 1990.

[3] C. Truchet and G. Assayag, *Constraint Programming in Music*. Wiley, 2011.

[4] T. Anders and E. R. Miranda, "Constraint programming systems for modeling music theories and composition," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 30:1–30:38, Oct. 2011.

[5] M. Laurson, "PatchWork: A visual programming language and some musical applications," Ph.D. dissertation, 1996.

[6] C. Truchet and P. Codognet, "Musical constraint satisfaction problems solved with adaptive search," *Soft Computing*, vol. 8, no. 9, 2004.

[7] N. Halbwachs, *Synchronous Programming of Reactive Systems*, 1993.

[8] G. Berry, "The Foundations of Esterel," in *Proof, Language, and Interaction*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000, pp. 425–454.

[9] G. Baudart, F. Jacquemard, L. Mandel, and M. Pouzet, "A synchronous embedding of antescofo, a domain-specific language for interactive mixed music," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, 2013, pp. 1–12.

[10] G. Berry and M. Serrano, "Hop and HipHop: Multitier Web Orchestration," in *International Conference on Distributed Computing and Internet Technology*. Springer, 2014, pp. 1–13.

[11] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[12] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Documentation*, 2015.

[13] G. Tack, "Constraint Propagation – Models, Techniques, Implementation," Ph.D. dissertation, Saarland University, 2009.

[14] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*. Cambridge University Press, 2002.

[15] C. Schulte, *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2302.

[16] M. Gemünde, J. Brandt, and K. Schneider, "Clock refinement in imperative synchronous languages," *EURASIP Journal on Embedded Systems*, vol. 2013, no. 1, pp. 1–21, 2013.