Spacetime Programming

A Synchronous Language for Composable Search Strategies

Pierre Talbot
Université de Nantes
Nantes, France
pierre.talbot@univ-nantes.fr

ABSTRACT

Search strategies are crucial to efficiently solve constraint satisfaction problems. However, programming search strategies in the existing constraint solvers is a daunting task and constraint-based languages usually have compositionality issues. We propose spacetime programming, a paradigm extending the synchronous language Esterel and *timed concurrent constraint programming* with backtracking, for creating and composing search strategies. In this formalism, the search strategies are composed in the same way as we compose concurrent processes. Our contributions include the design and behavioral semantics of spacetime programming, and the proofs that spacetime programs are deterministic, reactive and extensive functions. Moreover, spacetime programming provides a bridge between the theoretical foundations of constraint-based concurrency and the practical aspects of constraint solving. We developed a prototype of the compiler that produces search strategies with a small overhead compared to the hard-coded ones.

KEYWORDS

synchronous programming, concurrent constraint programming, constraint satisfaction problem, search strategy

1 INTRODUCTION

Constraint programming is a powerful paradigm to model problems in terms of constraints over variables. This declarative paradigm solves many practical problems including scheduling, vehicle routing or biology problems [33], as well as more unusual problems such as in musical composition [55]. Constraint programming describes *what* the problem is, whereas procedural approaches describe *how* a problem is solved. The programmer declares the constraints of its problem, and relies on a generic constraint solver to obtain a solution.

A constraint satisfaction problem (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 constraints on these variables. The goal is to find a solution: a set of singleton domains such that every constraint is satisfied. For example, given the CSP $\langle \{x \mapsto \{1,2,3\}, y \mapsto \{1,2,3\}\}, \{x > y, x \neq 2\}\rangle$, a solution is $\{x \mapsto 3, y \mapsto 1\}$.

PPDP '19, October 7–9, 2019, Porto, Portugal

The solving procedure usually interleaves two steps: propagation and search. Propagation removes values from the domains that do not satisfy at least one constraint. The search step makes a choice when propagation cannot infer more information and backtracks to another choice if the former one did not lead to a solution. The successive interleaving of choices and backtracks lead to the construction of a search tree that can be explored with various search strategies. In this paper, the term "search strategy" takes the broad sense of any procedure that describes how a CSP is solved.

In order to attain reasonable efficiency, the programmer must often customize the search strategy *per problem* [3, 47, 54]. However, to program a search strategy in a constraint solver is a daunting task that requires expertise and good understanding on the solver's intrinsics. This is why various language abstractions emerged to ease the development of search strategies [21, 26, 57, 60].

One of the remaining problems of search languages is the compositionality of search strategies: how can we easily combine two strategies and form a third one? Compositionality is important to build a collection of search strategies reusable across problems. To cope with this compositionality issue, we witness a growing number of proposals based on functional programming [41], constraint logic programming [40], and search combinators [42]. However, a recurring issue in these approaches is the difficulty to share information among strategies; we discuss this drawback and others in Section 8.

We propose *spacetime programming* (or "spacetime" for short) to tackle this compositionality issue. Spacetime is a language based on the imperative synchronous language Esterel [5] and timed concurrent constraint programming (TCC) [34, 35]. Spacetime extends the synchronous model of computation of Esterel with backtracking, and refines the interprocess communication mechanism of TCC with lattice-based variables. We introduce these features in the following two paragraphs.

Synchronous Programming with Backtracking. The synchronous paradigm [15] proposes a notion of logical time dividing the execution of a program into a sequence of discrete instants. A synchronous program is composed of processes that wait for one another before the end of each instant. Operationally, we can view a synchronous program as a coroutine: a function that can be called multiple times and that maintains its state between two successive calls. One call to this coroutine represents one instant that elapsed. The main goal of logical time is to coordinate concurrent processes while avoiding typical issues of parallelism, such as deadlock or indeterminism [22].

Spacetime inherits most of the temporal statements of TCC, and more specifically those of the synchronous language Esterel [5], including the delay, sequence, parallel, loop and conditional statements. The novelty of spacetime is to connect the search tree generated by a CSP and linear logical time of synchronous programming. Our proposal is captured in the following principle:

A node of the search tree is explored in exactly one logical instant.

A corollary to this first principle is:

A search strategy is a synchronous process.

These two principles are illustrated in Sections 4 and 6 with well-known search strategies.

Deterministic Interprocess Communication. The second characteristic of spacetime is inherited from concurrent constraint programming (CCP) [38]. CCP defines a shared memory as a global constraint store accumulating partial information. The CCP processes communicate and synchronize through this constraint store with two primitives: $tell(c)$ for adding a constraint c into the store, and $ask(c)$ for asking if c can be deduced from the store. Concurrency is treated by requiring the store to grow *monotonically* and *extensively*, which implies that removal of information is not permitted. An important result is that any CCP program is a closure operator over its constraint store (a function that is idempotent, extensive and monotone).

TCC embeds CCP in the synchronous paradigm [34, 35] such that an instant is guaranteed to be a closure operator over its store; however information can be lost between two instants. There are two main differences between spacetime and TCC.

Firstly, instead of a central and shared constraint store, variables in spacetime are defined over lattice structures. The tell and ask operations are thus defined on lattices, where *tell* relies on the least upper bound operation and *ask* on the order of the lattice. In Section 3, we formalize a CSP as a lattice that we later manipulate as a variable in spacetime programs.

Secondly, unlike TCC programs, spacetime programs are not closure operators by construction. This stems from the negative ask statement (testing the absence of information) which is not monotone, and the presence of external functions which are not necessarily idempotent and monotone. As in Esterel, we focus instead on proving that the computation is deterministic and reactive. In addition, we also prove that spacetime programs are extensive functions *within* and *across* instants (Section 5.6).

Contributions. In summary, this paper includes the following contributions:

- We provide a language tackling the compositionality issue of search strategies. We illustrate this claim in Sections 4 and 6 by reconstructing and combining well-known search strategies.
- We extend the behavioral semantics of Esterel to backtracking and variables defined over lattices with proofs of determinism, reactivity and extensiveness (Section 5).
- We implement a prototype of the compiler[1], and integrate spacetime into the Java language (Section 7). The evaluation of the search strategies presented in this paper shows a small overhead compared to the hard-coded ones of Choco [31].
- Spacetime is the first language that unifies constraint-based concurrency, synchronous programming and backtracking. This unification bridges a gap between the theoretical foundations of CCP and the practical aspects of constraint solving.

[1] Open source compiler available at https://github.com/ptal/bonsai/tree/PPDP19.

2 DEFINITIONS

To keep this paper self-contained, we expose necessary definitions on lattice theory which are then used to define constraint programming. Given an ordered set $\langle L, \leq \rangle$ and $S \subseteq L$, $x \in L$ is a *lower bound* of S if $\forall y \in S,\ x \leq y$. We denote the set of all the lower bounds of S by S^ℓ. The element $x \in L$ is the *greatest lower bound* of S if $\forall y \in S^\ell,\ x \geq y$. The *least upper bound* is defined dually by reversing the order.

Definition 2.1 (Lattice). An ordered set $\langle L, \leq \rangle$ is a lattice if every pair of elements $x, y \in L$ has both a least upper bound and a greatest lower bound. We write $x \sqcup y$ (called join) the least upper bound of the set $\{x, y\}$ and $x \sqcap y$ (called meet) its greatest lower bound. A *bounded lattice* has a top element $\top \in L$ such that $\forall x \in L,\ x \leq \top$ and a bottom element $\bot \in L$ such that $\forall x \in L, \bot \leq x$.

As a matter of convenience and when no ambiguity arises, we simply write L instead of $\langle L, \leq \rangle$ when referring to ordered structures. Also, we refer to the ordering of the lattice L as $\leq_L$ and similarly for any operation defined on L.

An example is the lattice *LMax* of increasing integers $\langle N, \geq, max \rangle$ where $N \subset \mathbb{N}$, $\geq$ is the natural order on $\mathbb{N}$ and max is the join operator. Dually, we also have *LMin* with the order $\leq$ and join min.

The Cartesian product $P \times Q$ is defined by the lattice $\langle \{(x, y) \mid x \in P, y \in Q\}, \leq_\times \rangle$ such that $(x_1, y_1) \leq_\times (x_2, y_2)$ if $x_1 \leq_P x_2 \land y_1 \leq_Q y_2$. Given the lattice $L_1 \times L_2$, it is useful to define the following projection functions, for $i \in \{1, 2\}$ and $x_i \in L_i$ we have $\pi_i((x_1, x_2)) \mapsto x_i$. For the sake of readability, we also extend the projection over any subset $S \subseteq L_1 \times L_2$ as $\pi_i'(S) = \{\pi_i(x) \mid x \in S\}$.

Given a lattice $\langle L, \leq \rangle$, a function $f : L \to L$ is extensive if for all $x \in L$, we have $x \leq f(x)$. This property is important in language semantics because it guarantees that a program does not lose information. More background on lattice theory can be found in [7, 11].

3 LATTICE VIEW OF CONSTRAINT PROGRAMMING

As we will see shortly, a spacetime program is a function exploring a state space defined over a lattice structure. To illustrate this paradigm, we choose in this paper to focus on the state space generated by constraint satisfaction problems (CSPs). Hence, we describe the lattice of CSPs and the lattice of its state space, called a *search tree*.

3.1 Lattice of CSPs

Following various works [1, 13, 28, 46], we introduce constraint programming through the prism of lattice theory. The main observation is that the hierarchical structure of constraint programming can be defined by a series of lifts. We incrementally construct the lattice of CSPs.

First of all, we define the domain of a variable as an element of a lattice structure. In the case of finite domains, an example is the powerset lattice $\langle \mathcal{P}(N), \supseteq \rangle$ with the finite set $N \subset \mathbb{N}$ and ordered by superset inclusion. For instance, a variable x in $\{0, 1, 2\} \in \mathcal{P}(N)$ is less informative than a singleton domain $\{0\}$, i.e. $\{0, 1, 2\} \leq \{0\}$. Other lattices can be used (see e.g. [13]), so we abstract the lattice of variable's domains as $\langle D, \leq \rangle$.

Let *Loc* be an unordered set of variable's names. We lift the lattice of domains D to the lattice of partial functions $Loc \rightharpoonup D$. In operational terms, a partial function represents a store of variables.

Definition 3.1 (Store of variables). We write the set of all partial functions from *Loc* to D as $[Loc \rightharpoonup D]$. Let $\sigma, \tau \in [Loc \rightharpoonup D]$. We write $\pi_1'(\sigma)$ the subset of *Loc* on which σ is defined. The set of variables stores is a lattice defined as:

$$SV = \langle [Loc \rightharpoonup D], \tau \leq \sigma \text{ if } \forall \ell \in \pi_1'(\tau),\ \tau(\ell) \leq_D \sigma(\ell) \rangle$$

We find convenient to turn a partial function σ into a set, called its graph, defined by $\{(x, \sigma(x)) \mid x \in \pi_1'(\sigma)\}$. Given a lattice L, the lattice $Store(Loc, L)$ is the set of the graphs of all partial functions from *Loc* to L. In comparison to SV, we parametrize the lattice $Store(Loc, L)$ by its set of locations *Loc* and underlying lattice L, so we can reuse it later. Notice that $Store(Loc, D)$ is isomorphic to SV.

We turn a logical constraint $c \in C$ into an extensive function $p : SV \to SV$, called *propagator*, over the store of variables. For example, given the store $d = \{x \mapsto \{1, 2\}, y \mapsto \{2, 3\}\}$ and the constraint $x \geq y$, a propagator $p_\geq$ associated to $\geq$ gives $p_\geq(d) = \{x \mapsto \{2\}, y \mapsto \{2, 3\}\}$. We notice that this propagation step is extensive, e.g. $d \leq p_\geq(d)$. Beyond extensiveness, a propagator must also be sound, i.e. it does not remove solutions of the induced constraint, to guarantee the correctness of the solving algorithm.

We now define the lattice of all propagators $SC = \langle \mathcal{P}(Prop), \subseteq \rangle$ where *Prop* is the set of all propagators (extensive and sound functions). The order is given by set inclusion: additional propagators bring more information to the CSP. We call an element of this lattice a *constraint store*. The lattice of all CSPs—with propagators instead of logical constraints—is given by the Cartesian product $CSP = SV \times SC$.

Given a CSP $\langle d, \{p_1, \ldots, p_n\} \rangle \in CSP$, the *propagation step* is realized by computing the fixpoint of $p_1(p_2(..p_n(d)))$. We note $propagate : CSP \to CSP$ the function computing this fixpoint. In practice, this function is one crucial ingredient to obtain good performance, and this is part of the theory of constraint propagation (e.g. see [1, 44, 51]). In the rest of this paper, we keep this propagation step abstract, and we delegate it to specialized solvers when needed.

Once propagation is at a fixpoint, and if the domain d is not a solution yet, a search step must be performed. Search consists in splitting the state space with a branching function $branch : CSP \to Store(\mathbb{N}, CSP)$ and exploring successively the sub-problems created. We call an element of the lattice $Store(\mathbb{N}, CSP)$ the *branches*. The indices of the branches serve to order the child nodes. For instance, a standard branching function consists in selecting the first non-instantiated variable and to divide its domain into two halves—one explored in each sub-problem. If the branching strategy is strictly extensive ($x < f(x)$) over each branch $b_i \in branch(\langle d, P \rangle)$, and does not add variables into d, then this solving procedure is guaranteed to terminate on finite domains. This solving algorithm is called *propagate and search*.

3.2 Lattice of Search Trees

A novel aspect of this lattice framework is to view the search tree as a lattice as well. It relies on the *antichain completion* which derives a lattice to the antichain subsets of its powerset.[2]

Definition 3.2 (Antichain completion). The antichain completion of a lattice L, written $\mathscr{A}(L)$, is a lattice defined as:

$$\mathscr{A}(L) = \langle \{S \subseteq \mathcal{P}(L) \mid \forall x, y \in S,\ x \leq y \implies x = y\}, \\ S \leq Q \text{ if } \forall y \in Q,\ \exists x \in S,\ x \leq_L y \rangle$$

It is equipped with the Smyth order [48].

The lattice of the search trees is defined as $ST = \mathscr{A}(CSP)$. Intuitively, an element $q \in ST$ represents the frontier of the search tree being explored. The antichain completion accurately models the fact that parents' nodes are not stored in q. Operationally, we view q as a queue of nodes[3], which is central to backtracking algorithms.

The missing piece to build and explore the CSP state space is the *queueing strategy* which allows us to pop and push nodes onto the queue.

Definition 3.3 (Queueing strategy). Let L be a lattice and $\mathscr{A}(L)$ be its antichain completion. The pair of functions

$$pop : \mathscr{A}(L) \to \mathscr{A}(L) \times L \\ push : \mathscr{A}(L) \times Store(\mathbb{N}, L) \to \mathscr{A}(L)$$

is a queueing strategy if, for any extensive function $f : \mathscr{A}(L) \times L \to \mathscr{A}(L) \times Store(\mathbb{N}, L)$, the function composition $push \circ f \circ pop$ is extensive over $\mathscr{A}(L)$.

In the context of CSP solving, we have $L = CSP$ and $\mathscr{A}(L) = ST$. As examples of queueing strategies, we have depth-first search (DFS), breadth-first search (BFS) and best-first search.

The state space of a CSP $\langle d, P \rangle$ is explored by computing the fixpoint of the function $solve(\{\langle d, P \rangle\})$ which is defined as:

$$solve : ST \to ST \\ solve = push \circ (id \times (branch \circ propagate)) \circ pop$$

This function formalizes the usual steps when solving a constraint problem: pop a node from the queue, propagate it, divide it into several sub-problems, and push these sub-problems onto the queue. The output type of each function matches the input type of the next one—notice that we use the identity function *id* to avoid passing the search tree to *propagate* and *branch*. Reaching a fixpoint on *solve* means that we explored the full search tree, and explored all solutions if there is any.

3.3 The Issue of Compositionality

The *solve* function is parametrized by a branching and queuing strategies. However, this does not suffice to program every search strategy. For example, the depth-bounded search strategy—further developed in the next section—consists in exploring the search tree until a given depth is reached. To program this strategy in the current framework, we must extend the definition of a CSP with a depth counter defined over *LMax* (given in Section 2). The resulting

[2] In the finite case, the antichain completion of a lattice L is isomorphic to the set of ideals of L as shown by Crampton and Loizou [10]. We prefer the antichain formulation because it is closer to the data structure of a queue.

[3] Despite the name, this terminology of "queue" does not imply a particular queueing strategy, i.e. the order in which the nodes are explored.

search tree is defined as $ST_2 = \mathscr{A}(CSP \times LMax)$. We also extend *solve* with two functions: *inc* for increasing the counter of the child nodes, and *prune* for pruning the nodes at the given depth:

$$solve2 : ST_2 \to ST_2$$
$$solve2 = push \circ (id \times (inc \circ prune \circ branch \circ propagate)) \circ pop$$

Although orthogonal to the depth counter, the types of the *propagate* and *branch* functions must be modified to work over $CSP \times LMax$. Another solution would be to project elements of $CSP \times LMax$ with additional *id* functions. A more elaborated version of this idea, relying on monads to encapsulate data, is investigated in *monadic constraint programming* [41]. The search strategies defined in this framework require the users to have substantial knowledge in functional language theory. Similarly, constraint solving libraries are made extensible through software engineering techniques such as design patterns. In all cases, a drawback is that it complicates the code base, which is hard to understand and extend with new search strategies. Moreover, such software architecture varies substantially across solvers.

The problem is that we need to either *modify existing structures* or integrate the strategies into some predefined software architecture in order to program new search strategies. We call this problem the *compositionality issue*. Our proposal is to rely on *language abstractions* instead of software abstractions to program search strategies.

4 LANGUAGE OVERVIEW

We give a tour of the spacetime model of computation and syntax by incrementally building the iterative-deepening search strategy [19]. A key insight is that this search strategy is developed generically with regard to the state space.

4.1 Model of Computation

The model of computation of spacetime is inspired by those of (timed) concurrent constraint programming (CCP) and Esterel.

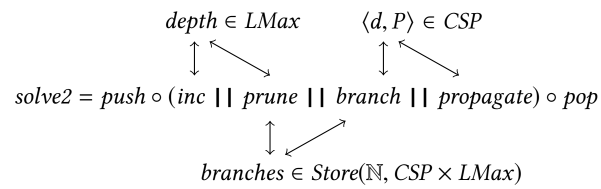
CCP model of computation. We view the structure of a CCP program as a lattice $\langle L, \models, \sqcup \rangle$ where $\models$ is called the *entailment*. The entailment is the order of the lattice defined as $a \models b \equiv b \leq a$. Following Scott's information systems [45], CCP views the bottom element $\bot$ as the lack of information, the top element $\top$ as all the information, the tell operator $x \sqcup y$ as the join of the information in x and y, and the ask operator $x \models y$ as an expression that is true if we can deduce y from x.

CCP processes communicate through this lattice by querying for information with the entailment, or adding information with join. For example, consider the following definitions of `prune` and `inc`:

(**when** depth $\models$ 4 **then** "prune the subtree") || (depth = depth $\sqcup$ (depth + 1))

with || the parallel composition. The first process is suspended on `depth` $\models$ `4` until `depth` becomes greater than or equal to 4. Hence, the second process is completed first if we initially have `depth < 4`. The limitation of CCP is that it is not possible to write a process for the statement "prune the subtree". This is because a CCP process computes over a fixed lattice, such as *CSP*, but it is not possible to compute over its antichain completion, which is necessary for creating and exploring its state space.

Space component of spacetime. The approach envisioned with the spacetime paradigm is to view a search algorithm as a set of concurrent processes exploring collaboratively a state space. In this model, we rewrite *solve*2 as a parallel composition of processes as follows (the arrows indicate read/write operations):



Firstly, we pop a node from the queue which contains the variables *depth* and $\langle d, P\rangle$. Then, similarly to CCP, the processes communicate by reading and writing into these variables. The Cartesian product of the variables, called the *space* of the program, is automatically synthesised by the spacetime semantics. This is reflected in the type $CSP \times LMax$ of *branches*. The processes only manipulate *branches* through dedicated statements, namely `space` and `prune` (that we introduce below).

Time component of spacetime. One remaining question is how to synchronize processes so that every process waits for each other before the next node is popped? Our proposal is to rely on the notion of synchronous time of Esterel. During each instant, a process is executed until it encounters a special statement called `pause`.[4] Once `pause` is reached, the process waits for all other processes to be paused or terminated. The next instant is then started.

The novelty in spacetime is to connect the passing of time to the expansion of the search tree. Concretely, an instant consists in performing three consecutive steps: pop a node, execute the processes until they are all paused, and push the resulting branches onto the queue. We repeat these steps until the queue is empty or all processes are terminated.

We now detail this model of computation through several examples, notably by programming the *inc* and *prune* processes. We delay the presentation of *propagate* and *branch* to Section 6.

4.2 Binary Search Tree

A spacetime program is a set of Java classes augmented with *spacetime class fields* (prefixed by the `single_space`, `world_line` or `single_time` keywords) and *processes* (prefixed by `proc` or `flow` keywords). The type of a spacetime field or local variable is a Java class that implements a lattice interface providing the entailment and join operators. A process does not return a value; it acts as a coroutine mutating the spacetime variables in each instant. In contrast, Java method calls are viewed as atomic operations in a spacetime process.

One of the simplest process in spacetime is to generate an infinite binary search tree:

```
class Tree {
  public proc binary =
    loop
      space nothing end;
```

[4] To ensure cooperative behavior among processes, the amount of work to perform during an instant must be bounded in time.

```
    space nothing end;
    pause;
  end }
```

This process generates a binary tree in which every node is empty; we will decorate these nodes with data later. A branch is created with the statement `space p end` where the process p describes the differences between the current node and the child node. In the example, the difference is given by `nothing` which is the *empty process* terminating immediately without effect, thus all generated nodes will be the same.

In each instant, four actions are realized (we connect these actions to the model of computation in parenthesis):

(1) A node is popped from the queue (function *pop*).
(2) The process is executed until we reach a `pause` statement (process between *pop* and *push*).
(3) We retrieve the sequence of branches, duplicate the backtrackable state[5] for each `space p end` statement, and execute each p on a distinct copy of the state to obtain the child nodes (writing into the variable *branches*).
(4) The child nodes are pushed onto the queue (function *push*).

These actions are repeated in the statement `loop`. Since the process `binary` never terminates and the queue is never empty, the state space generated is infinite. In summary, a process generates a sequence of branches during an instant, and a search tree across instants.

Now, we illustrate the use of spacetime variables by introducing a node and depth counters:

```
class Node {
  public single_space LMax node = new LMax(0);
  public flow count = readwrite node.inc()  }
class Depth {
  public world_line LMax depth = new LMax(−1);
  public flow count = readwrite depth.inc ()  }
```

A *flow process* executes its body p in each instant, the keyword `flow` is a syntactic sugar for `loop p; pause end`. Both classes work similarly: we increase by one their counters in each instant with the method `inc` on `LMax`. We discuss two kinds of annotations appearing in these examples: *read/write annotations* and *spacetime annotations*.

Read/write annotations indicate how a variable is manipulated inside a host function. It comes in three flavors: `read x` indicates that x is only read by the function, `write x` that the function only writes more information in x without reading it, and `readwrite x` that the value written in x depends on the initial value of x. Every write in x must respect its lattice order and this verification is left to the programmer of the lattice. For example, the method `x.inc()` is defined as $x = x + 1$, and thus x must be annotated by `readwrite`. These attributes are essential to ensure determinism when variables are shared among processes, and for correctly scheduling processes.

Spacetime annotations indicate how a variable evolves in memory through time. For this purpose, a spacetime program has three distinct memories in which the variables can be stored:

(i) Global memory (keyword `single_space`) for variables evolving globally to the search tree. A `single_space` variable has a unique location in memory throughout the execution. For example, the counter `node` is a `single_space` variable: since we explore one node in every instant, we increase its value by one in each instant.
(ii) Backtrackable memory (keyword `world_line`) for variables local to a path in the search tree. The queue of nodes is the backtrackable memory. For example, the value of the counter `depth` must be restored on backtrack in the search tree.
(iii) Local memory (keyword `single_time`) for variables local to an instant and reallocated in each node. A `single_time` variable only exists in one instant. We will encounter this last annotation later on.

Another feature of interest is the support of *modular programming* by assembling processes defined in different classes. As an example, we combine `Tree.binary` and `Depth.count` with the parallel statement:

```
public proc binary_stats  =
  module Tree generator  = new Tree ();
  module Depth depth = new Depth();
  par run generator . binary ()  ||  run depth.count() end
end
```

The variables `generator` and `depth` are annotated with `module` to distinguish them from spacetime variables. We use the keyword `run` to disambiguate between process calls and method calls.

Last but not least, the *disjunctive parallel* statement `par p || q end` executes two processes in lockstep. It terminates once *both* processes have terminated. Dually, we have the *conjunctive parallel* statement `par p <> q end` which terminates (i) in the next instant if one of p or q terminates, or (ii) in the current instant if both p and q terminate. The condition (i) implements a form of *weak preemption*. An instant terminates once every process is paused or terminated. In this respect, `pause` can be seen as a synchronization barrier among processes.

4.3 Depth-bounded Search

Now we are ready to program a search strategy in spacetime. We consider the strategy `BoundedDepth` which bounds the exploration of the search tree to a depth limit:

```
public class BoundedDepth {
  single_space LMax limit;
  public BoundedDepth(LMax limit) {  this . limit  =  limit ;  }
  public proc bound_depth =
    module Depth counter = new Depth ();
    par
    <> run counter . count ()
    <> flow
        when counter . depth  |=  limit  then prune end
        end
    end
  end }
```

Whenever `depth` is greater than or equal to `limit` we prune the remaining search subtree. The construction of the search tree through time is illustrated in Figure 1 with `limit` set at 2. The black dots are

[5] The backtrackable state is the Cartesian product of the variables prefixed by `world_line` (see below).

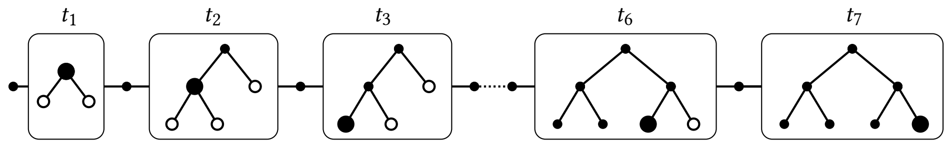


Figure 1: Progression of bounded depth search in each instant with maximum depth equals to 2.

the nodes already visited, the large one is the one currently being visited and the white ones are those pushed onto the queue.

The disjunctive parallel composes two search trees by union, whereas the conjunctive parallel composes them by intersection. For example, if we have `binary() || bound_depth()`, the search tree obtained is exactly the one of `binary()`, while `binary() <> bound_depth()` prunes the search tree at some depth limit. Over two branches, the statement `prune || space` p creates a single branch `space` p, while `prune <> space` p creates a pruned branch. This is made clear in Section 5.2 where we formalize these composition rules.

4.4 A Glimpse of the Runtime

The class `Tree` is processed by the spacetime compiler which compiles every process into a regular Java method. For example, the process `binary` is compiled into the following Java method:

```
public Statement binary () {
  return new Loop(
    new Sequence(Arrays. asList (
      new SpaceStmt(new Nothing ()),
      new SpaceStmt(new Nothing ()),
      new Delay(CompletionCode.PAUSE)))); }
```

The compiled method returns the abstract syntax tree (AST) of the process. This AST is then interpreted by the runtime engine `SpaceMachine`:

```
public static void main(String[] args) {
  Tree tree = new Tree ();
  StackLR queue = new StackLR ();
  SpaceMachine machine = new SpaceMachine(tree.binary (), queue);
  machine.execute (); }
```

We parametrize the runtime engine by the queue `StackLR`: a traditional stack exploring the tree in depth-first search from left to right. Importantly, it means that the spacetime program is generic with regard to the queueing strategy. The method `execute` returns either when the spacetime program terminates, the queue becomes empty or we reach a `stop` statement. This latest statement offers a way to stop and resume a spacetime program outside of the spacetime world, which is handy for interacting with the external world. In contrast, a `pause` statement is resumed automatically by the runtime engine as long as the queue is not empty.

Being aware of the runtime mechanism is helpful to extend `BoundedDepth` to the restart-based strategy *iterative depth-first search* (IDS) [19]. IDS successively restarts the exploration of the same search tree by increasing the depth limit. This strategy combines the advantages of breadth-first search (diversifying the search) and depth-first search (weak memory consumption). Assuming we have a class `BoundedTree` combining `BoundedDepth` and `Tree`, we program IDS in the host language as follows:

```
public static void main(String[] args) {
  for(int limit =0; limit < max_depth(); limit ++) {
    BoundedTree tree = new BoundedTree(new LMax(limit));
    StackLR queue = new StackLR ();
    SpaceMachine machine = new SpaceMachine(tree.search (), queue);
    machine.execute (); }}
```

We introduce additional examples of search strategies in Section 6, and show how to combine two restart-based strategies in spacetime.

5 SEMANTICS OF SPACETIME

We develop the semantics of spacetime independently from the host language (Java in the previous section). To achieve that, we suppose the program is flattened: every module definition and process call are inlined, and no recursion is allowed in processes. We obtain a lighter abstract syntax of the spacetime statements formalized as follows (p, q are processes, x, y are identifiers, and T is a host type):

$$\begin{aligned}\langle p, q\rangle ::= \; & T\ x^{\rightarrow|\circlearrowleft|\downarrow} \mid \texttt{when}\ x\ \texttt{|=}\ y\ \texttt{then}\ p\ \texttt{else}\ q \\ \mid\; & f(x_1^{w|r|rw}, \ldots, x_n^{w|r|rw}) \\ \mid\; & \texttt{nothing} \mid \texttt{pause} \mid \texttt{stop} \mid \texttt{loop}\ p \mid p\ ;\ q \mid p\ \texttt{||}\ q \mid p \texttt{<>} q \\ \mid\; & \texttt{space}\ p \mid \texttt{prune}\end{aligned}$$

Spacetime annotations are shorten as follows: $\rightarrow$ stands for `single_space`, $\circlearrowleft$ for `single_time` and $\downarrow$ for `world_line`.[6] Read/write annotations are given by w for `write`, r for `read` and rw for `readwrite`. Without loss of generality, we encapsulate the interactions between spacetime and its host language in function calls.

5.1 Behavioral Semantics

The semantics of spacetime is inspired by the logical behavioral semantics of Esterel, a big-step semantics, as defined in [6, 29]. The semantic rules of spacetime defining the control flow of processes (for example `loop` or `pause`) are similar to those in Esterel. We adapt these rules to match the two novel aspects of spacetime:

(i) Storing lattice-based variables in one of the three memories (instead of Esterel's Boolean signals).
(ii) Defining a structure to collect and compose the (pruned) branches created during an instant.

The rules proper to spacetime are specific to either (i) or (ii).

Given the set of outputs produced by a program, a derivation in the behavioral semantics is a proof that a program transition is valid. The behavioral transition rule is given as:

$$Q, L \vdash p \xrightarrow[I \sqcup O]{O'} p'$$

[6] These symbols reflect how the variables evolve in the search tree. For example, $\downarrow$ depicts an evolution from the root to a leaf of the tree along a path.

where the program p is rewritten into the program p' under (i) the queue Q equipped with a queueing strategy $(pop, push)$, (ii) the set of locations $L \subset Loc$ providing a unique identifier to every declaration of variable, (iii) the input I, and (iv) the outputs O and O'. We denote the set of syntactic variable names (as appearing in the source code) with *Name*, such that $Name \cap Loc = \emptyset$. We write $L \dot{\cup} \{\ell\}$ the disjoint union, which is useful to extract a fresh location ℓ from L.

The goal of behavioral semantics is not to compute an output O but to prove that a transition is valid if we already know O. We obtain a valid derivation if the output O' derived by the semantics is equal to the provided output O. Conceptually, the behavioral semantics allows processes to instantaneously broadcast information. In the following, we call the input and output structures *universe* and we write U' for the output O', and $U = I \sqcup O$ for the input/output provided.

Space structure. The variable environment of a program, called its *space*, stores the spacetime variables. The spacetime annotations are given by the set $spacetime = \{\rightarrow, \circlearrowleft, \downarrow\}$. The set of values of a variable is given by its type in the host language, which must be a lattice structure. From the spacetime perspective, we erase the types in the set *Value* which is the disjoint union of all types, and we delegate typing issues to the host language. Putting all the pieces together, the set of spacetime variables *Var* is the poset $\{\top\} \cup (spacetime \times Value)$. We need a distinct top element $\top$ for representing variables that are merged with a different spacetime or type—this can be checked at compile-time.

Given a set of locations *Loc*, the lattice of the spaces of the program is defined as $Space = Store(Loc, Var)$. The element $\bot$ is the empty space. Given a space $S \in Space$, we define the subsets of the single space variables with $S^{\rightarrow}$, the single time variables with $S^{\circlearrowleft}$ and the world line variables with $S^{\downarrow}$. In addition, given a variable $(st, v) \in S(\ell)$ at location ℓ, we define the projections $S^{st}(\ell) = st$ and $S^{V}(\ell) = v$ to respectively extract the spacetime and the value of the variable. $S^{V}(\ell)$ maps to $\bot$ if ℓ is undefined in S.

Universe structure. A universe incorporates all the information produced during an instant including the space, the completion code and the sequence of branches. The completion code models the state of a process at the end of an instant: normally terminated (code 0), paused in the current instant with `pause` (code 1) or stopped in the user environment with `stop` (code 2). We denote the set of completion codes with $Compl = \langle \{0, 1, 2\}, \leq_{\mathbb{N}} \rangle$. We describe the sequence of branches B^* in the next section. The universe structure is defined as follows:

$$Universe = Space \times Compl \times B^*$$

Given $U \in Universe$, we define the projections U^S, U^k and U^B respectively mapping to the space, completion code and the sequence of branches. We also write U^V instead of U^{S^V}, $U^{\rightarrow}$ instead of $U^{S^{\rightarrow}}$ and similarly for $\circlearrowleft$ and $\downarrow$.

5.2 Search Semantics

In this section, we use the following relevant subset of spacetime:

$$\langle p, q \rangle ::= p \;;\; q \mid p \;||\; q \mid p <> q \mid \texttt{space}\ p \mid \texttt{prune} \mid \alpha$$

where $p, q \in Proc$ with *Proc* the set of all the processes, and α is an atomic statement which is not composed of other statements. We can extend the definitions given below to the full spacetime language without compositional issues.

We give the semantics of the search tree statements with a branch algebra. We have a set of all branches defined as $B = \{\texttt{space}\ w \mid w \in Space^{\downarrow}\} \cup \{\texttt{prune}\}$. That is to say, a branch is either labelled by a `world_line` space or pruned.

Definition 5.1 (Branch algebra). The branch algebra is defined over a sequence of branches $\langle B^*, \circ, \vee, \wedge \rangle$ where all operators are associative, $\circ$ is noncommutative, and $\vee$ and $\wedge$ are commutative. The empty sequence $\langle \rangle$ is the identity element of the three operators.

The operators $\circ$, $\vee$ and $\wedge$ match the commutative and associative laws of the semantics of the operators `;`, `||` and `<>` respectively.

Sequence composition. Given $b_i, b_j \in B$ with $1 \leq i \leq n$ and $1 \leq j \leq m$, the sequence operator $\circ$ performs the concatenation of two sequences of branches as follows:

$$\langle b_1, \ldots, b_n \rangle \circ \langle b'_1, \ldots, b'_m \rangle = \langle\, b_1, \ldots, b_n, b'_1, \ldots, b'_m \,\rangle$$

Parallel compositions. We define the operators $\vee^1$ and $\wedge^1$ to combine two branches and then lift these operators to sequences of branches. Two sequences of branches are combined by repeating the last element of the shortest sequence when the sizes differ. Given $w, w' \in Space^{\downarrow}$ and $b \in B$, we define the disjunctive parallel operators $\vee^1$ between two branches and $\vee$ between two sequences of branches as follows:

$$\begin{aligned} &b \vee^1 \texttt{prune} && = b \\ &\texttt{space}\ w \vee^1 \texttt{space}\ w' && = \texttt{space}\ w \sqcup w' \\ &\langle b_1, \ldots, b_n \rangle \vee \langle b'_1, \ldots, b'_m \rangle && = \\ &\begin{cases} \langle\, b_1 \vee^1 b'_1,\ \ldots,\ b_{n-1} \vee^1 b'_{m-1},\ b_n \vee^1 b'_m \,\rangle & \text{if } n = m \\ \langle\, b_1 \vee^1 b'_1,\ \ldots,\ b_{n-1} \vee^1 b'_m,\ b_n \vee^1 b'_m \,\rangle & \text{if } n > m \end{cases} \end{aligned}$$

The case where $m > n$ is tackled by the commutativity of $\vee$. The conjunctive parallel operators $\wedge^1$ and $\wedge$ are defined similarly but for `prune`:

$$b \wedge^1 \texttt{prune} = \texttt{prune}$$

This algebra allows us to delete, replace or increase the information in a branch. For example, given a process p:

- p `<> (space nothing ; prune)` deletes every branch created by p but the first.
- p `<> (space nothing ; prune ; space nothing)` deletes the second branch.
- p `|| (prune ; space` q `; prune)` increases the information in the second branch by q.

We can also obtain any permutation of a sequence of branches with a suited *push* function. The only operation not supported is weakening the information of one branch. We have yet to find a use-case for such an operation.

5.3 Semantics Rules

The semantics rules of spacetime are given in Figure 2. We isolate host computations by relying on the *host transition rule* $e \xrightarrow[H]{H'}\!\!\!\!\rightarrow v$

NOTHING
$$Q, \{\} \vdash \texttt{nothing} \xrightarrow[U]{\bot,\, 0\, \langle\rangle} \texttt{nothing}$$

PAUSE
$$Q, \{\} \vdash \texttt{pause} \xrightarrow[U]{\bot,\, 1,\, \langle\rangle} \texttt{nothing}$$

STOP
$$Q, \{\} \vdash \texttt{stop} \xrightarrow[U]{\bot,\, 2,\, \langle\rangle} \texttt{nothing}$$

HCALL
$$\frac{f(\ell_1^{a_1}, \dots, \ell_n^{a_n}) \xrightarrow[host(U^S)]{H'} v}{Q, \{\} \vdash f(\ell_1^{a_1}, \dots, \ell_n^{a_n}) \xrightarrow[U]{(space(H'), 0, \langle\rangle)} \texttt{nothing}}$$

LOOP
$$\frac{Q, L \vdash p \xrightarrow[U]{U'} p' \qquad U^k \neq 0}{Q, L \vdash \texttt{loop}\ p \xrightarrow[U]{U'} p'\ ;\ \texttt{loop}\ p}$$

WHEN-TRUE
$$\frac{U^V(\ell_1) \models U^V(\ell_2) \twoheadrightarrow \mathit{true} \qquad Q, L \vdash p \xrightarrow[U]{U'} p'}{Q, L \vdash \texttt{when}\ \ell_1\ \texttt{|=}\ \ell_2\ \texttt{then}\ p\ \texttt{else}\ q \xrightarrow[U]{U'} p'}$$

WHEN-FALSE
$$\frac{U^V(\ell_1) \models U^V(\ell_2) \twoheadrightarrow v \quad v = \mathit{false} \lor v = \mathit{unknown} \qquad Q, L \vdash q \xrightarrow[U]{U'} q'}{Q, L \vdash \texttt{when}\ \ell_1\ \texttt{|=}\ \ell_2\ \texttt{then}\ p\ \texttt{else}\ q \xrightarrow[U]{U'} q'}$$

VAR-DECL↻
$$\frac{U' = (\{(\ell, (\circlearrowleft, \bot_T))\}, 0, \langle\rangle) \qquad Q, L \vdash p[x \to \ell] \xrightarrow[U]{U''} p'}{Q, L \mathbin{\dot\cup} \{\ell\} \vdash T\ x^{\circlearrowleft}\ ;\ p \xrightarrow[U]{U' \sqcup U''} T\ x^{\circlearrowleft}\ ;\ p'}$$

START-VAR-DECL→↓
$$\frac{st \neq \circlearrowleft \qquad x \in \mathit{Name} \qquad U' = (\{(\ell, (st, \bot_T))\}, 0, \langle\rangle) \qquad Q, L \vdash p[x \to \ell] \xrightarrow[U]{U''} p'}{Q, L \mathbin{\dot\cup} \{\ell\} \vdash T\ x^{st}\ ;\ p \xrightarrow[U]{U' \sqcup U''} T\ \ell^{st}\ ;\ p'}$$

PRUNE
$$Q, \{\} \vdash \texttt{prune} \xrightarrow[U]{(\bot, 0, \langle \texttt{prune} \rangle)} \texttt{nothing}$$

RESUME-VAR-DECL→↓
$$\frac{\ell \in \mathit{Loc} \qquad v = \begin{cases} (\rightarrow, \bot_T) & \text{if } st = \rightarrow \\ (\downarrow, \pi_2(pop(Q))(\ell)) & \text{if } st = \downarrow \end{cases} \qquad U' = (\{(\ell, v)\}, 0, \langle\rangle) \qquad Q, L \vdash p \xrightarrow[U]{U''} p'}{Q, L \vdash T\ \ell^{st}\ ;\ p \xrightarrow[U]{U' \sqcup U''} T\ \ell^{st}\ ;\ p'}$$

SPACE-PRUNED
$$\frac{U^B \neq \langle \texttt{space}\ W \rangle}{Q, \{\} \vdash \texttt{space}\ p \xrightarrow[U]{(\bot, 0, \langle \texttt{space}\ \bot \rangle)} \texttt{nothing}}$$

SPACE
$$\frac{U^B = \langle \texttt{space}\ W \rangle \qquad \bot, \{\} \vdash p \xrightarrow[U \sqcup (W, 0, \langle\rangle)]{U'} p' \qquad U'^k = 0 \qquad U'^{\rightarrow} = U'^{\circlearrowleft} = \emptyset}{Q, \{\} \vdash \texttt{space}\ p \xrightarrow[U]{(\bot, 0, \langle \texttt{space}\ U'^{\downarrow} \rangle)} \texttt{nothing}}$$

ENTER-SEQ
$$\frac{Q, L \vdash p \xrightarrow[U]{U'} p' \qquad U'^k \neq 0}{Q, L \vdash p\ ;\ q \xrightarrow[U]{U'} p'\ ;\ q}$$

NEXT-SEQ
$$\frac{U^B = B \circ B' \qquad Q, L \vdash p \xrightarrow[(U^S, U^k, B)]{U'} p' \qquad U'^k = 0 \qquad Q, L' \vdash q \xrightarrow[(U^S, U^k, B')]{U''} q'}{Q, L \mathbin{\dot\cup} L' \vdash p\ ;\ q \xrightarrow[U]{U' \sqcup^{\circ} U''} q'}$$

PAR$^{\lor}$
$$\frac{Q, L \vdash p \xrightarrow[U]{U'} p' \qquad Q, L' \vdash q \xrightarrow[U]{U''} q'}{Q, L \mathbin{\dot\cup} L' \vdash p\ \texttt{||}\ q \xrightarrow[U]{U' \sqcup^{\lor} U''} p'\ \texttt{||}\ q'}$$

PAR$^{\land}$
$$\frac{Q, L \vdash p \xrightarrow[U]{U'} p' \qquad Q, L' \vdash q \xrightarrow[U]{U''} q' \qquad U'^k \neq 0 \land U''^k \neq 0}{Q, L \mathbin{\dot\cup} L' \vdash p\ \texttt{<>}\ q \xrightarrow[U]{U' \sqcup^{\land} U''} p'\ \texttt{<>}\ q'}$$

EXIT-PAR$^{\land}$
$$\frac{Q, L \vdash p \xrightarrow[U]{U'} p' \qquad Q, L' \vdash q \xrightarrow[U]{U''} q' \qquad U'^k = 0 \lor U''^k = 0}{Q, L \mathbin{\dot\cup} L' \vdash p\ \texttt{<>}\ q \xrightarrow[U]{U' \sqcup^{\land} U''} \texttt{nothing}}$$

Figure 2: Behavioral semantics rules of spacetime.

which reduces the expression e into the value v with the input/output host environment H and the output environment H'. The interface between spacetime and the host language is realized by a pair of functions (*host*, *space*) such that *host* maps the space S into the host environment H and vice versa. We write $e \twoheadrightarrow v$ when the space of the program is not modified. We explain each fragment of the semantics in the following paragraphs.

The axioms NOTHING, PAUSE and STOP set the completion code respectively to terminated, paused and stopped. We leave the output space and branches empty.

The main interaction with the host language is given by the rule HCALL. The function f and its arguments are evaluated in the host version of the input/output space, written $host(U^S)$. The properties guaranteed by the spacetime semantics depend on the properties fulfilled by the host functions.

The rule LOOP simulates an iteration of the loop by extracting and executing the body p outside of the loop. We guarantee that p is not instantaneous by forbidding the completion code k to be equal to 0.

The conditional rules WHEN-TRUE and WHEN-FALSE evaluate the entailment result of $x \models y$ to execute either p or q. In case the entailment status is *unknown*, which happens if x and y are not ordered, we promote *unknown* to *false*. This is reminiscent of the closed world assumption in logic programming: "what we do not know is *false*".

$$\textsc{hcall}\ \dfrac{inc(\ell_0^{rw}) \xrightarrow[host(S_2)]{H'}\!\!\!\!\rightarrow v}{Q,\{\} \vdash \mathtt{inc}(\ell_0^{rw}) \xrightarrow[(S_2,0,\langle\rangle)]{(space(H'),0,\langle\rangle)} \mathtt{nothing}}$$

$$\textsc{space}\ \dfrac{\cdots}{Q,\{\} \vdash \mathtt{space\ inc}(\ell_0^{rw}) \xrightarrow[(S_1,0,\langle \mathtt{space}\ S_2\rangle)]{(\{\},0,\langle \mathtt{space}\ S_2\rangle)} \mathtt{nothing}}$$

$$\textsc{when-true}\ \dfrac{U^V(\ell_0) \vDash 1 \twoheadrightarrow true \qquad \cdots}{Q,\{\} \vdash \mathtt{when}\ \ell_0\ \mathtt{|=\ 1\ then\ space\ inc}(\ell_0^{rw}) \xrightarrow[(S_1,0,\langle \mathtt{space}\ S_2\rangle)]{(\{\},0,\langle \mathtt{space}\ S_2\rangle)} \mathtt{nothing}}$$

$$\textsc{hcall}\ \dfrac{inc(\ell_0^{rw}) \xrightarrow[host(S_1)]{H'}\!\!\!\!\rightarrow v}{Q,\{\} \vdash \mathtt{inc}(\ell_0^{rw}) \xrightarrow[(S_1,0,\langle \mathtt{space}\ S_2\rangle)]{(space(H'),0,\langle\rangle)} \mathtt{nothing}}$$

$$\textsc{exit-par-}\vee\ \dfrac{\cdots \qquad \cdots}{Q,\{\} \vdash (\mathtt{when}\ \ell_0\ \mathtt{|=\ 1\ then\ space\ inc}(\ell_0^{rw}))\ \mathtt{<>}\ \mathtt{inc}(\ell_0^{rw}) \xrightarrow[(S_1,0,\langle \mathtt{space}\ S_2\rangle)]{(S_1,0,\langle \mathtt{space}\ S_2\rangle)} \mathtt{nothing} \qquad U' = (\{(\ell_0,(\downarrow,0))\},0,\langle\rangle)}$$

$$\textsc{start-var-decl}{\rightarrow}{\downarrow}\ \dfrac{\cdots}{Q,\{\ell_0\} \vdash \mathtt{LMax}\ \mathtt{x}^{\downarrow};\ ((\mathtt{when\ x\ |=\ 1\ then\ space\ inc}(\mathtt{x}^{rw}))\ \mathtt{<>}\ \mathtt{inc}(\mathtt{x}^{rw})) \xrightarrow[(\{(\ell_0,(\downarrow,1))\},0,\langle \mathtt{space}\ \{(\ell_0,(\downarrow,2))\}\rangle)]{U' \sqcup (\{(\ell_0,(\downarrow,1))\},0,\langle \mathtt{space}\ \{(\ell_0,(\downarrow,2))\}\rangle)} \mathtt{nothing}}$$

Figure 3: An example of derivation in the behavioral semantics.

5.3.1 Semantics of spacetime variables. The variable declaration rules register the variables in the space or queue memory. A variable's name x must be substituted to a unique location ℓ. Locations are necessary to distinguish variables with the same name in the space and queue—this is possible if the scope of the variable is re-entered several times during[7] and across instants. In the rules VAR-DECL↻ and START-VAR-DECL→↓, we extract a fresh location ℓ from L and substitute x for ℓ in the program p, which is written $p[x \rightarrow \ell]$.[8] The substitution function is defined inductively over the structure of the program p. We give its two most important rules:

$$y[x \rightarrow \ell] \mapsto \begin{cases} \ell & \text{if } x = y \\ y & \text{if } x \neq y \end{cases}$$

$$(T\ y^{st}\ ; p)[x \rightarrow \ell] \mapsto \begin{cases} T\ y^{st}\ ; p & \text{if } x = y \\ T\ y^{st}\ ; p[x \rightarrow \ell] & \text{if } x \neq y \end{cases}$$

It replaces any identifier equals to x by ℓ, and stops when it reaches a variable declaration with the same name.

For `single_time` variables, we create a new location in each instant (VAR-DECL↻). For `single_space` and `world_line` variables, we create a new location only during the first instant of the statement (START-VAR-DECL→↓), and the next instants reuse the same location (RESUME-VAR-DECL→↓).

In the first instant, the values are initialized to the bottom element $\bot_T$ of the lattice T. In the next instants, we retrieve the value of a `world_line` variable in the queue by popping one node, and then extracting the value at location ℓ from that node. The values of `single_space` variables are transferred from one instant to the next by the reaction rules introduced in the next section.

5.3.2 Semantics of search statements. The statement `prune` is an axiom creating a single pruned branch. For `space` p, we have two cases: either we execute p under the input/output branch $\langle \mathtt{space}\ W\rangle$ (rule SPACE), or if another process prunes this branch, we avoid executing p (rule SPACE-PRUNED). The execution of the `space` statement does not impact the variables in the current instant, which is materialized by setting the space to $\bot$ in the output universe. In addition, we require that p terminates instantaneously, only writes into `world_line` variables and does not create nested branches.

To specify the sequential and parallel statements, we extend join over *Universe* with a branch operator. We have $(S, k, B) \sqcup^{\wedge} (S', k', B')$ equals to $(S \sqcup S', k \sqcup k', B \wedge B')$, and similarly for $\circ$ and $\vee$.

To formalize the sequence $p\ ;\ q$, we have the rule ENTER-SEQ which tackles the case where p does not terminate during the current instant, and the rule NEXT-SEQ where p terminates and q is executed. The disjunctive parallel statement $p\ ||\ q$ derives p and q concurrently and merges their output universes with $\sqcup^{\vee}$ (rule PAR$^{\vee}$). Finally, the conjunctive parallel statement p <> q is similar to `||` when none of p or q terminates (rule PAR$^{\wedge}$). However, if one process terminates, we rewrite the statement to `nothing` which prevents this statement to be executed in future instants (rule EXIT-PAR$^{\wedge}$). Note that the semantics of composition in space of `||` and <> match their respective semantics of composition in time.

5.3.3 An example of derivation. We illustrate the mechanics of the behavioral semantics with a short example:

LMax $\mathtt{x}^{\downarrow}$; ((**when** x |= 1 **then space** inc($\mathtt{x}^{rw}$)) <> inc($\mathtt{x}^{rw}$))

Two processes communicate over the variable x. The first creates a branch incrementing x by one if it is greater than 1, while the second increments x in the current instant. To derive this process in the behavioral semantics, we set the input/output universe to $U = (\{(\ell_0, (\downarrow, 1))\}, 0, \langle \mathtt{space}\ \{(\ell_0, (\downarrow, 2))\}\rangle)$ and attempt to prove that the output universe (the structure above the arrow) is equal to U. For clarity, we set $S_1 = \{(\ell_0, (\downarrow, 1))\}$ and $S_2 = \{(\ell_0, (\downarrow, 2))\}$. The derivation is given in Figure 3. We notice that the statement `space` is derived with the input/output space S_2 instead of S_1. Operationally, it implies that the branch must be evaluated at the end of the current instant.

5.4 Semantics Across Instants

A spacetime program is automatically executed until it terminates, stops or its queue of nodes becomes empty. Therefore, we must lift the transition rule to succession of instants, which gives the

[7] This is a problem known as reincarnation in Esterel [6].
[8] The variable declaration must be evaluated with regard to its body, this is why the body p follows the declaration. We can transform any variable declaration *Type* x^{st} which is not followed by any statement to *Type* x^{st} ; `nothing`.

$$\textsc{react}\quad \frac{\begin{array}{c} causal(p) \qquad Q, \mathcal{L}_i \vdash p \xrightarrow[H_i]{U'} p' \qquad Q' = push(Q, U'^B) \\ U'^k = 1 \text{ and } Q' \text{ is not empty} \qquad i+1, \mathcal{L} \vdash \langle Q', p' \rangle \xhookrightarrow[H]{H'} \langle Q'', p'' \rangle \\ H'' = \{(j, U'' \sqcup (U'^{\rightarrow}, 0, \langle\rangle)) \mid (j, U'') \in H'\} \end{array}}{i, \mathcal{L} \vdash \langle Q, p \rangle \xhookrightarrow[H]{\{(i,U')\} \sqcup H''} \langle Q'', p'' \rangle}$$

$$\textsc{exit-react}\quad \frac{\begin{array}{c} causal(p) \\ Q, \mathcal{L}_i \vdash p \xrightarrow[H_i]{U'} p' \qquad Q' = push(Q, U'^B) \qquad U'^k \neq 1 \text{ or } Q' \text{ is empty} \end{array}}{i, \mathcal{L} \vdash \langle Q, p \rangle \xhookrightarrow[H]{\{(i,U')\}} \langle Q', p' \rangle}$$

Figure 4: Reaction rules of spacetime.

following *reaction rule*:

$$i, \mathcal{L} \vdash \langle Q, p \rangle \xhookrightarrow[H]{H'} \langle Q', p' \rangle$$

where the state $\langle Q, p\rangle$ is rewritten into the state $\langle Q', p'\rangle$ with Q a queue with a queueing strategy (*pop*, *push*), and p a process. In addition, we have: (i) a counter of instants $i \in \mathbb{N}$, (ii) a sequence of sets of locations $\mathcal{L} \in Store(\mathbb{N}, Loc)$ where $\mathcal{L}_i \in \mathcal{L}$ is the set of locations at the instant i, (iii) the sequence of input/output universes $H \in Store(\mathbb{N}, Universe)$ where H_i is the input/output at the instant i, and (iv) the sequence of output universes $H' \in Store(\mathbb{N}, Universe)$. The lifting to sequence of universes is inspired by ReactiveML [24]. The reaction rules are defined in Figure 4. The rule REACT models the passing of time from one paused instant to the next. Of interest, we notice that the values of the `single_space` variables are joined into all of the future universes. We also observe that the two rules REACT and EXIT-REACT are exclusive on the termination condition. We now discuss the side condition $causal(p)$ which performs the causality analysis of the program in each instant.

5.5 Causality Analysis

Causality analysis is crucial to prove that spacetime programs are reactive, deterministic and extensive functions. An example of non-reactive program is `when x |= y then f(write y) end`. The problem is that if we add information in y, the condition `x |= y` might not be entailed anymore, which means that no derivation in the behavioral semantics is possible. This is similar to emitting a signal in Esterel after we tested its absence. Due to the lattice order on variables, we can however write on a value after an entailment condition, consider for example `when x |= y then f(write x) end`. Whenever `x |= y` is entailed, it will stay entailed even if we later write additional information on x, so this program should be accepted.

The causality analysis symbolically executes an instant of a process, yielding the set of all symbolic paths reachable in an instant. It also symbolically executes the paths of all branches generated in each instant. For space reason, we only show the most important part of the causality analysis: the properties that a path must fulfil to be causal. A path is a sequence of atomic statements $\langle a_1, \ldots, a_n\rangle$ where a_i is defined as:

$$\langle atom \rangle ::= x \vDash y \mid f(x_1^{w|r|rw}, \ldots, x_n^{w|r|rw})$$

For example, the process `when x |= y then` $f(x^r)$ `else` $g(x^r)$ generates two paths: $\langle x \vDash y, f(x^r)\rangle$ for the then-branch, and $\langle y \vDash x, g(x^r)\rangle$ for the else-branch. A path p is causal if for all atoms $a_i \in p$ the following two conditions hold.

First, for each entailment atom $a_i = x \vDash y$ we require:

$$\forall z^b \in Vars(p_{i+1..|p|}), z = y \implies b = \mathtt{r} \tag{1}$$

with $Vars(p)$ the set of all variables in the path p. It ensures all remaining accesses on y to be read-only.

Second, for each function call $a_i = f(x_1^{b_1}, ..., x_n^{b_n})$ and each argument $x_k^{b_k}$ of f we require:

$$\forall z^b \in Vars(p_{i+1..|p|}), x_k = z \wedge (b_k = \mathtt{r} \vee b_k = \mathtt{rw}) \implies b = \mathtt{r} \tag{2}$$

Whenever a variable is accessed with `read` or `readwrite`, it can only be read afterwards. A consequence is that a variable cannot be accessed by two `readwrite` during a same instant.

Definition 5.2 (Causal process). A process is causal if for all its instants i, every path p in the instant i is causal ((1) and (2) hold).

5.6 Reactivity, Determinism and Extensiveness

We now only consider causal spacetime programs. In this section, we sketch the proofs that the semantics of spacetime is deterministic, reactive and an extensive function during and across instants. Importantly, these properties only hold if the underlying host functions meet the same properties. The two first properties are typical of the synchronous paradigm and are defined as follows.

Definition 5.3 (Determinism and reactivity). For any state $\langle Q, p\rangle$, the derivation

$$0, \mathcal{L} \vdash \langle Q, p \rangle \xhookrightarrow[H]{H'} \langle Q', p' \rangle$$

is deterministic (resp. reactive) if there is at most (resp. at least) one proof tree of the derivation.

LEMMA 5.4. *The semantics of spacetime is reactive and deterministic.*

The proofs are given in Appendices A.1 and A.2. They essentially verify the completeness and disjointness of the rules.

LEMMA 5.5. *The semantics of spacetime is extensive over its space during an instant.*

PROOF. Any write in the space is done through a variable declaration or a host function. The declaration rules only add more information into the space by using the join operator $\sqcup$. Otherwise, this property depends on the extensiveness of the host functions. □

To define the extensiveness property of a program across instants, we rely on the notion of *observable space*. Given a space $S \in Space$, its observable subset $obs(S) \subseteq S$ is the set of variables that can still be used in a future instant.

LEMMA 5.6. *Every observable variable is stored in either the queue or in a* `single_space` *variable.*

PROOF. The `world_line` variables are stored in a queue of nodes when pushed (rule REACT). In the case of a pruned node, the `world_line` variables are not observable since no child node can ever used their values again. The single_time variables are reallocated in each instant, thus not observable in future instants. □

Lemma 5.7 (Extensiveness). *Given a sequence of universes H and two instant indices $i > j$, we have $H_i^S \models obs(H_j^S)$.*

Proof. By Lemma 5.6, it is sufficient to only look at the queue and single_space variables: (i) the queue is extensive by Definition 3.3 of the queueing strategy, and (ii) the single_space variables are joined with their previous values (rule react), thus single_space variables that exist in H_i^S and H_j^S are ordered by induction on the instant indices. Therefore our semantics is extensive with regard to the sequence of universes derived. □

6 CONSTRAINT PROGRAMMING IN SPACETIME

In Section 4, we defined a process generating an infinite binary search tree. As the underlying structure of the state space is a lattice, the "raw state space" can be programmed by the user. We demonstrate this fact by programming a process generating the state space of a constraint satisfaction problem (CSP).

A search strategy can be specialized or generic with regard to the state space. For example, the strategy IDS (introduced in Section 4.4) can be reused on the CSP state space without modification. As an additional example of generic search strategy, we consider limited discrepancy search (LDS) and its variants. It can be combined effortlessly with IDS and the CSP state space generator. We also introduce a branch and bound strategy which is bound to the CSP state space. Overall, the goal is to show that search strategies can be developed independently from the state space while retaining their compositionality.

6.1 Generating the CSP State Space

We consider a basic but practical solver using the propagate and search algorithm presented in Section 3.

```
class Solver {
  single_time ES consistent = unknown;
  ref world_line VStore domains;
  ref world_line CStore constraints;
  public Solver(VStore domains, CStore constraints) {...}
  public proc search = par run propagation() <> run branch() end
  flow propagation =
    consistent <- constraints.propagate(readwrite domains);
    when consistent |= true then prune end
  end
  flow branch =
    when unknown |= consistent then
      single_time IntVar x = failFirstVar (domains);
      single_time Integer v = middleValue(x);
      space constraints <- x.le(v) end; // x ≤ v
      space constraints <- x.gt(v) end // x > v
    end
  // Interface to the Choco solver.
  private IntVar failFirstVar (VStore domains) { ... }
  private Integer middleValue(IntVar x) { ... } }
```

This example introduces new elements of syntax: (i) the ref keyword which indicates that the variable name is an alias to a spacetime variable declared in another class, (ii) the tell operator x <- e which is a syntactic sugar for write x.join(e), the join operation $x = x \sqcup e$, and (iii) the keywords true, false and unknown that are elements of the lattice *ES* explained below. We also remark that read annotations apply by default when not specified on variables.

The lattices VStore and CStore are respectively the variable store and the constraint store. The constraint solver Choco [31] is abstracted behind these two lattices and provides the main operations to propagate and branch on the state space. The branching strategy is usually a combination of a function selecting a variable in the store (here failFirstVar) and selecting a value in the domain of the variable (here middleValue). The two variables storing these results are annotated with single_time since they are only useful in the current instant. We split the state space with the constraints $x \leq v$ and $x > v$. In the implementation, this code is organized in a more modular way so we can assemble various parts of the branching strategies.

The lattice ES is defined as $\{true, false, unknown\}$ with the total order $false \models true \models unknown$. It is used to detect if the current node of the CSP is a solution (*true*), a failed node (*false*) or if we do not know yet (*unknown*). In the process propagation, we prune the current subtree if we reached a solution or failed node.

6.2 Branch and Bound Search

Branch and bound (BAB) is an algorithm to find the optimal solution of a CSP according to an objective function. BAB reasons over the whole search tree by keeping track of the *best solution* obtained so far, in contrast to propagation which operates on a single node at a time. It is implemented in the following class MinimizeBAB.

```
public class MinimizeBAB {
  ref world_line VStore domains;
  ref single_time ES consistent;
  ref single_space IntVar x;
  single_space LMin obj = bot;
  public MinimizeBAB(VStore domains, ES consistent, IntVar x) {...}
  public proc solve = par run minimize() <> run yield_objective () end
  proc minimize =
    loop
      when consistent == true then
        single_space LMin pre_obj = new LMin(x.getLB());
        pause;
        obj <− pre_obj;
      else pause end
    end
  flow yield_objective =
    consistent <− updateBound(write domains, write x, read obj)
  static ES updateBound(VarStore domains, IntVar x, LMin obj) { ... }
```

Along with the current variable store, we have the variable x to be minimized and its current best bound obj of type LMin. The single_space attribute indicates that the bound obj is global to the search tree and will not be backtracked. The class has two main processes: (i) minimize strengthens the bound obj with the value obtained in the previous solution node, and (ii) yield_objective,

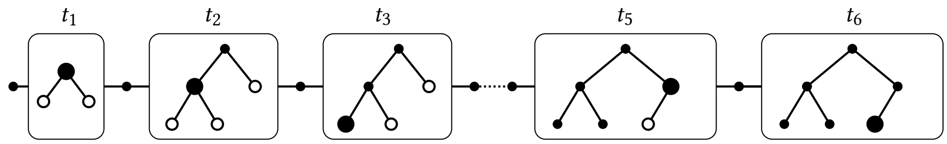


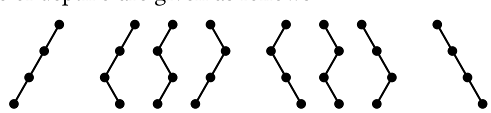
Figure 5: Combination of bounded depth and bounded discrepancy search.

through the function `updateBound`, interfaces with Choco to update `domains` with $x < obj$, so the next solution we find have a better bound.

There is an important detail to notice: we use a temporary variable `pre_obj` to store the latest bound instead of updating `obj` directly. Interestingly, if we do not, the causality analysis will fail since we have a cyclic dependency in the data: `obj` depends on `domains` and vice versa. Fortunately, the causality analysis prevents us from having a bug: adding the current bound in the CSP would turn a solution node into a failed node.

6.3 Limited Discrepancy Search and Variants

For some problems, the branching strategy can order the branches such that the left one is more likely to reach a solution first. Limited discrepancy search (LDS) was introduced to take advantage of this ordering property. It is based on the notion of *discrepancies* which is the number of right branches taken to reach a leaf node. In its original version [16], LDS successively increases the number of discrepancies a branching strategy can take by restarting the exploration of the full tree. The paths with 0, 1, 2 and 3 discrepancies in a tree of depth 3 are given as follows:



The first iteration generates the leftmost path, the second iteration allows one discrepancy, and so on. The search is complete if the discrepancy limit is not reached during one iteration. An iteration of LDS is programmed in spacetime as follows:

```
public class BoundedDiscrepancy {
  single_space LMax limit;
  world_line LMax dis = new LMax(0);
  public BoundedDiscrepancy(LMax limit) { ... }
  public flow bound =
    space nothing end;
    when dis |= limit then prune
    else space readwrite dis.inc () end end
  end }
```

Initially, the discrepancy counter `dis` is set to 0. The left branch is always taken, which we represent with a neutral `space nothing end` statement. The right branch is taken only if the discrepancies counter is less than the limit, otherwise we prune this branch. We can restart this search with the same technique as the one used for IDS (Section 4.4).

A drawback of LDS is that at each iteration k, it re-explores all paths with *k or less* discrepancies. In [20], Korf proposes an improved version of LDS (ILDS) where only paths with exactly k discrepancies are explored. We provide a library of reusable improved LDS strategies including ILDS, depth-bounded discrepancy search (DDS) [62] and LDS variants [30] in the implementation.

In addition to creating a search strategy from scratch, we often need to assemble existing strategies to obtain the best of two approaches. For example, the combination of LDS with IDS is discussed in [16], as well as the combination of DDS with IDS in [62]. These combinations can be easily programmed in spacetime; we obtain the first by combining `BoundedDepth` and `BoundedDiscrepancy`:

```
module BoundedDepth bd = new BoundedDepth(new LMax(2));
module BoundedDiscrepancy bdis =
  new BoundedDiscrepancy(new LMax(1));
par run db.bound() <> run bdis.bound() end
```

The result of this combination is shown in Figure 5. Similarly we can use the disjunctive parallel operator `||` to obtain their union. What's more, we can apply this strategy to the CSP state space, possibly augmented with the BAB process, in the very same way.

7 IMPLEMENTATION

The compiler of spacetime performs static analyses to ensure well-formedness of the program. It includes common analyses and transformations on synchronous programs such as causality analysis, detection of instantaneous loop and reincarnation [29, 52]. Specifically in spacetime, we ensure that every statement `space` p has an instantaneous body and does not contain nested `space` or `prune` statements. In addition, we provide several analyses to integrate Java and spacetime in a coherent way, especially for initializing objects with existing spacetime variables (keyword `ref`). These analyses are out of scope in this paper, but we provide a comprehensive list of the analyses in the file `src/errors.rs` of the implementation.

As shown in Section 4.4, every spacetime statement is mapped to a *synchronous combinator* encoding its behavior at runtime. Synchronous combinators are also used in the context of synchronous reactive programming—basically Esterel without reaction to absence—in the Java library SugarCubes [8, 50]. In this section, we overview how these combinators are scheduled in the runtime.

Replicating. Every spacetime program presented in this paper, as well as the experiments below, are available in the repository https://github.com/ptal/bonsai/tree/PPDP19.

7.1 Scheduling Algorithm

The main purposes of the runtime are to dynamically schedule concurrent processes, to retain the state of the program from an instant to the next, and to push and pop variables onto the queue. To achieve these goals, we extend the structures introduced in the behavioral semantics (Section 5.1) to incorporate *access counters* and a *suspended completion code.*

Problem	Spacetime	Choco	Factor
13-Queens	16.4s (62946n/s)	5.3s (194304n/s)	3.1
14-Queens	89.9s (62020n/s)	30.6s (182218n/s)	2.9
15-Queens	528.2s (60972n/s)	185.2s (173816n/s)	2.85
Golomb Ruler 10	1.8s (17407n/s)	1s (31154n/s)	1.8
Golomb Ruler 11	40.1s (14186n/s)	27.2s (20888n/s)	1.47
Golomb Ruler 12	425.8s (10871n/s)	279.8s (16541n/s)	1.52
Latin Square 60	19s (155n/s)	17.1s (172n/s)	1.10
Latin Square 75	61.2s (73n/s)	57.9s (77n/s)	1.06
Latin Square 90	150.3s (44n/s)	147.8s (45n/s)	1.02

Table 1: Comparison of spacetime and Choco on the resolution time and nodes-per-second (n/s).

Firstly, we equip every variable with an *access counter* $(w, rw, r) \in LMin^3$ where w is the numbers of `write`, rw of `readwrite` and r of `read` accesses that can still happen on a variable in the current instant. As suggested by the lattice *LMin*, these counters are decreased whenever the corresponding access is performed. We extend the poset *Var* to access counters: $\{\top\} \cup (spacetime \times Value \times LMin^3)$.

Secondly, given a variable x and its access counter (w, rw, r), we say that a process is *suspended* if it needs to perform a `readwrite` on x when $w > 0$, or to read x when $w > 0$ or $rw > 0$. A process cannot be suspended on a write access. Whenever a process is stuck, the flow of control is given to another process. We add this additional stuck status in the set of completion codes *Compl* with the code 3.

In order to schedule processes, the runtime performs a *can* and *cannot* analyses over the program. The *can analysis* computes an upper bound on the counters: the numbers of accesses that can still happen on each variable in the current instant. The *cannot analysis* decreases counters by invalidating parts of the program that cannot be executed.

Consider the following spacetime program ($x, y \in LMax$):

```
when x |= y then f(write x, read y) else g(read x, write y) end
```

Initially, the counters of x and y are both set to $(1, 0, 1)$. Therefore, we cannot decide the entailment of $x \vDash y$ because its result might change due to future writes on x or y. However, we observe that if $x \vDash y$ holds then we can only write on x, which cannot change the entailment result. Similarly if $x \nvDash y$ holds, we can only write on y. To unlock such a situation, the *cannot analysis* decreases the counters of the variables with unreachable read/write accesses. Thanks to the causality analysis, a deadlock situation cannot happen since every access in every path is well-ordered.

The algorithm scheduling an instant alternates between the execution of the process, and the decrement of access counters with the *cannot* analysis.[9] The mechanics of this scheduling algorithm is close to the one of SugarCubes [8] and ReactiveML [25].

7.2 Experiments

We terminate this section with a short experimental evaluation. The experiments were run on a 1.8GHz Intel(R) Core(TM) i7-8550U processor running GNU/Linux. A warm-up time of about 30*s* was performed on the JVM before any measure was recorded.

We select three CSPs to test the overhead of a spacetime strategy in comparison to the same hard-coded Choco strategy. The propagation engine is the one of Choco in both cases. As shown in Table 1, the overhead factor of spacetime varies from almost 1 to at most 3.1 depending on the problem to solve. To obtain a *search intensive* algorithm, we search for all solutions of the N-Queens problem which has only three constraints to propagate in each node. This is the worst-case scenario for spacetime since the number of nodes is directly linked to the number of reactions of a spacetime program, and thus its overhead factor. We also consider a *propagation intensive* algorithm by searching for a single solution of a Latin Square problem which has a large number of constraints. To find a solution, the search never backtracks so the number of nodes is few. This explains the small overhead factor of spacetime which is almost 1. Finally, we evaluate a branch and bound (BAB) search strategy on the Golomb Ruler problem. BAB finds the best solution of an optimization problem, and thus explores a large tree. In this case, the overhead factor of spacetime drops to 1.5 thanks to the more realistic balance between search and propagation.

As for the correctness, spacetime always finds the same number of nodes, solutions and failures than Choco, as well as the same lower bounds for optimization problems (Golomb ruler). It indicates that the exact same search tree is explored.

8 RELATED WORK

We review two families of search languages: constraint logic programming and combinator-based search languages. Afterwards, we discuss the independent issue of integrating arbitrary data into imperative synchronous languages.

8.1 Constraint Logic Programming

Constraint logic programming (CLP) [17] is a paradigm extending logic programming with constraints. We can program search strategies by using the backtracking capabilities of logic programming. CLP systems such as GNU-Prolog [9, 12] and Eclipse [2, 39] propose various built-in blocks to construct a customized search strategy. Although CLP is an elegant formalism, it suffers from three drawbacks:

(1) There is no mechanism to compose search strategies.
(2) Global state, such as a node counter, is programmed via system dependent non-backtrackable mutable state libraries.
(3) It is bound to the evaluation strategy of Prolog, which for example means that LDS with highest-occurrence discrepancies cannot be easily implemented.[10]

The `tor/2` predicate [40] tackles the compositionality issue of CLP systems. It proposes to replace the disjunctive Prolog predicate `;/2` by a `tor/2` predicate which, in addition to creating two branches in the search tree, is a synchronization point. Two search strategies defined with `tor/2` can be merged with the predicate `tor_merge/2`. This extension allows the user to program various strategies independently and to assemble them. However, the search predicates are not executed concurrently, thus two search strategies cannot be interleaved and communicate over a shared variable. For example, the processes `Solver.search` and `MinimizeBAB.solve`

[9] A sketch of this algorithm is available in Appendix A.3.

[10] See the documentation of Eclipse at http://eclipseclp.org/doc/bips/lib/fd_search/search-6.html.

must be interleaved because they communicate over the variables `consistent` and `domains`.

8.2 Search Combinators

Early constraint search languages appeared around 1998 with Localizer [27], Salsa [21] and OPL [58, 61]. More recent approaches include Comet [59] (successor of Localizer), the search combinators [42] and its subset MiniSearch [32]. Comet and Localizer are specialized to local search, a non-exhaustive form of constraint solving. Local search languages differ because their programs are not necessarily extensive and are not always based on backtracking search. Search combinators mostly focus on the control part of search and it is interesting to take an example (from [42]):

$$\begin{aligned} &id(s) \stackrel{\text{def}}{=} ir(depth, 0, +, 1, \infty, s) \\ &ir(p, l, \oplus, i, u, s) \stackrel{\text{def}}{=} let(n, l, restart(n \leq u, \\ &\qquad and([assign(n, n \oplus i), limit(p \leq n, s)]))) \end{aligned}$$

The combinator *id* is an iterative depth-first search (IDS) [19] that restarts a strategy s by increasing the depth limit. The pattern of iteratively restarting the search is encapsulated in a combinator *ir* where the strategy s is restarted until we reach a limit $n \leq u$. To summarize, n is an internal counter initialized at l, and increased by $n \oplus i$ on each restart. They show that LDS is just another case of the combinator *ir* with discrepancies.

In search combinators, the search strategy is written *vertically*: each strategy is encapsulated in another strategy. In spacetime, we compose search strategy *horizontally*: each strategy is executed concurrently ("next to") another strategy. We believe that both vertical and horizontal compositionality is required in order to achieve high re-usability of search strategies.

A drawback of combinators-based languages is that they rely on data from the constraint solver, and the interactions with the host language are not formalized. In particular, it is not possible that two search strategies safely communicate over shared variables.

8.3 Arbitrary Data in Synchronous Languages

Signals in Esterel are Boolean values, which are limited when processes need to communicate more complex information. This is why they bring the notions of valued signals and variables for storing non-Boolean values [4, 53]. However, they are more restricted than pure signal: testing the value of a signal is only possible when all emissions have been performed, and variables must not be shared for writing across processes. Sequentially constructive Esterel (SCEst) [49] brings variables to Esterel that can be used across processes. The main idea is that any value must be manipulated following an *init-update-read* cycle within an instant. This is similar to our way to schedule *write-readwrite-read*, but there is no notion of order between values in SCEst. Therefore we can use destructive assignment similarly to sequential languages. In spacetime, the choice of lattices as the underlying data model comes from CCP and is more suited for constraint programming. In this respect, lattice-based variables unify the notions of signals, valued signals and variables of Esterel.

ReactiveML merges the imperative synchronous and functional paradigms without negative ask [25]. An advantage is that we can manipulate arbitrary functional data. Note that the addition of mutable states to ReactiveML is not deterministic [23].

Default TCC [36] is TCC with negative ask. It views an instant as a set of closure operators, one for each assumption on the result of the ask statements. A weakness of default TCC is to speculate on the result of the negative asks, which is implemented by backtracking inside an instant if its guess was wrong [37]. This is also problematic for external functions that produce side-effects.

9 CONCLUSION

Concurrent constraint programming (CCP) is a theoretical paradigm that formalizes concurrent logic programming inspired by constraint logic programming [56]. Unfortunately, this marriage is incomplete since backtracking, available in constraint logic programming, is not incorporated in CCP. We believe that the missing piece is the notion of logical time, as it appears in the synchronous paradigm, and it fostered the development of spacetime.

In the first part of this paper, we argued that logical time is a suitable device to conciliate concurrency and backtracking. The main underlying idea is captured as follows: a search strategy explores one node of the search tree per logical instant. In particular, we took the example of constraint solving in which designing search strategies is crucial to solve a CSP efficiently. We developed several search strategies in a modular way, and showed that they can be composed to obtain a new one. As a result, spacetime improves on the compositionality issues faced by developers of search strategies.

In the second part of this paper, we developed the foundations of spacetime by extending the behavioral semantics of Esterel to lattice-based variables and backtracking. We proved that the semantics is deterministic, reactive and that a spacetime program only accumulates more and more information during and across instants (extensiveness).

Further developments of spacetime include static compilation such as in Esterel [29] to improve efficiency, development in a proof assistant of the reactivity, determinism and extensiveness proofs, and formalization of a precise connection between the operational semantics (runtime) and the behavioral semantics. Furthermore, a natural extension of spacetime is to reify the queue inside the language itself instead of relying on the host language. The key idea of this extension is to merge the time hierarchy of synchronous languages [14, 24] and the space hierarchy induced by deep guards in logic programming [18] and Oz computation spaces [43]. First-class queue will allow users to program restart-based search strategies directly in spacetime instead of partly relying on the host language. Preliminary extension of the compiler indicates that this approach is feasible. Finally, although we applied spacetime to constraint programming, the notion of constraints is not built-in since we rely on lattice abstractions. Therefore, we firmly believe that spacetime is suitable to express strategies in other fields tackling combinatorial exploration such as in satisfiability modulo theories (SMT), model checking and rewriting systems.

ACKNOWLEDGMENTS

I would like to thank the anonymous reviewers for their useful comments. I am indebted to Carlos Agon who gave me the freedom to pursue this research and for fruitful discussions about the spacetime paradigm. My special thanks go to Clément Poncelet, David Cachera, Charlotte Truchet and Yinghan Ling for their helpful comments and suggestions on this paper. This work was partially supported by Agence Nationale de la Recherche under Grant No. 15-CE25-0002 (Coverif). A substantial part of this work was done at the Institute for Research and Coordination in Acoustics/Music (IRCAM) and Sorbonne University, Paris.

REFERENCES

[1] Krzysztof R. Apt. 1999. The essence of constraint propagation. *Theoretical computer science* 221, 1-2 (1999), 179–210. https://doi.org/10.1016/S0304-3975(99)00032-8

[2] Krzysztof R. Apt and M Wallace. 2007. *Constraint logic programming using ECLiPSe.* Cambridge University Press.

[3] J. Christopher Beck. 2007. Solution-guided multi-point constructive search for job shop scheduling. *Journal of Artificial Intelligence Research* 29 (2007), 49–77. https://doi.org/10.1613/jair.2169

[4] Gérard Berry. 2000. *The Esterel v5 language primer: version v5_91.* Centre de mathématiques appliquées, Ecole des mines and INRIA. ftp://ftp-sop.inria.fr/marelle/Laurent.Thery/esterel/esterel.pdf

[5] Gérard Berry. 2000. The Foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte (Eds.). MIT Press, Cambridge, MA, USA, 425–454.

[6] Gerard Berry. 2002. The Constructive Semantics of Pure Esterel. Draft version 3. (2002).

[7] Garrett Birkhoff. 1967. *Lattice Theory* (3rd ed.). AMS Colloquium Publications, Vol. XXV. American Mathematical Society.

[8] Frédéric Boussinot and Jean-Ferdy Susini. 2000. Java threads and SugarCubes. *Software: Practice and Experience* 30, 5 (2000), 545–566. https://doi.org/10.1002/(SICI)1097-024X(20000425)30:5<545::AID-SPE308>3.0.CO;2-Q

[9] Philippe Codognet and Daniel Diaz. 1996. Compiling constraints in clp(FD). *The Journal of Logic Programming* 27, 3 (1996), 185 – 226. https://doi.org/10.1016/0743-1066(95)00121-2

[10] Jason Crampton and George Loizou. 2001. The completion of a poset in a lattice of antichains. *International Mathematical Journal* 1, 3 (2001), 223–238. http://www.isg.rhul.ac.uk/~jason/Pubs/imj.pdf

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

[12] Daniel Diaz, Salvador Abreu, and Philippe Codognet. 2012. On the implementation of GNU Prolog. *Theory and Practice of Logic Programming* 12, 1 (2012), 253–282. https://doi.org/10.1017/S1471068411000470

[13] Antonio J. Fernández and Patricia M. Hill. 2004. An Interval Constraint System for Lattice Domains. *ACM Trans. Program. Lang. Syst.* 26, 1 (Jan. 2004), 1–46. https://doi.org/10.1145/963778.963779

[14] Mike Gemünde, Jens Brandt, and Klaus Schneider. 2013. Clock refinement in imperative synchronous languages. *EURASIP Journal on Embedded Systems* 2013, 1 (10 Apr 2013), 3. https://doi.org/10.1186/1687-3963-2013-3

[15] Nicolas Halbwachs. 1993. *Synchronous programming of reactive systems.* Springer Science & Business Media. https://doi.org/10.1007/978-1-4757-2231-4

[16] William D. Harvey and Matthew L. Ginsberg. 1995. Limited discrepancy search. In *Proceedings of the 14th International Conference on Artificial Intelligence (IJCAI '95).* 607–615. https://www.ijcai.org/Proceedings/95-1/Papers/080.pdf

[17] J. Jaffar and J.-L. Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87).* ACM, New York, NY, USA, 111–119. https://doi.org/10.1145/41625.41635

[18] Sverker Janson and Seif Haridi. 1991. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Logic Programming Symposium.* MIT Press, San Diego, California. http://eprints.sics.se/2097/ (Revised version of SICS Research Report R91:08).

[19] Richard E. Korf. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27, 1 (1985), 97 – 109. https://doi.org/10.1016/0004-3702(85)90084-0

[20] Richard E. Korf. 1996. Improved Limited Discrepancy Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1 (AAAI'96).* AAAI Press, 286–291. http://new.aaai.org/Papers/AAAI/1996/AAAI96-043.pdf

[21] François Laburthe and Yves Caseau. 2002. SALSA: A Language for Search Algorithms. *Constraints* 7, 3 (01 Jul 2002), 255–288. https://doi.org/10.1023/A:1020565317875

[22] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (May 2006), 33–42. https://doi.org/10.1109/MC.2006.180

[23] Louis Mandel. 2006. Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive.

[24] Louis Mandel, Cédric Pasteur, and Marc Pouzet. 2015. Time refinement in a functional synchronous language. *Science of Computer Programming* 111 (2015), 190–211. https://doi.org/10.1016/j.scico.2015.07.002

[25] Louis Mandel and Marc Pouzet. 2005. ReactiveML: A Reactive Extension to ML. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05).* ACM, New York, NY, USA, 82–93. https://doi.org/10.1145/1069774.1069782

[26] Thierry Martinez, François Fages, and Sylvain Soliman. 2015. Search by constraint propagation. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15).* ACM Press, 173–183. https://doi.org/10.1145/2790449.2790527

[27] Laurent Michel and Pascal Van Hentenryck. 1999. Localizer: A modeling language for local search. *INFORMS Journal on Computing* 11, 1 (1999), 1–14. https://doi.org/10.1287/ijoc.11.1.1

[28] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. 2013. A constraint solver based on abstract domains. In *Verification, Model Checking, and Abstract Interpretation.* Springer, 434–454. https://doi.org/10.1007/978-3-642-35873-9_26

[29] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2007. *Compiling Esterel* (1st ed.). Springer Publishing Company, Incorporated. https://doi.org/10.1007%2F978-0-387-70628-3

[30] Patrick Prosser and Chris Unsworth. 2011. Limited discrepancy search revisited. *Journal of Experimental Algorithmics* 16 (May 2011), 1.1. https://doi.org/10.1145/1963190.2019581

[31] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2017. *Choco Documentation.* TASC - LS2N CNRS UMR 6241, COSLING S.A.S. http://www.choco-solver.org

[32] Andrea Rendl, Tias Guns, Peter J. Stuckey, and Guido Tack. 2015. MiniSearch: a solver-independent meta-search language for MiniZinc. In *Principles and Practice of Constraint Programming.* Springer, 376–392. https://doi.org/10.1007/978-3-319-23219-5_27

[33] Francesca Rossi, Peter van Beek, and Toby Walsh. 2006. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc.

[34] Vijay Saraswat, Vineet Gupta, and Radha Jagadeesan. 2014. TCC, with History. In *Horizons of the Mind. A Tribute to Prakash Panangaden.* Springer, 458–475. https://doi.org/10.1007/978-3-319-06880-0_24

[35] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. 1994. Foundations of Timed Concurrent Constraint Programming. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science.* IEEE, 71–80. https://doi.org/10.1109/LICS.1994.316085

[36] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. 1996. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation* 22, 5 (1996), 475 – 520. https://doi.org/10.1006/jsco.1996.0064

[37] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. 2003. jcc: Integrating Timed Default Concurrent Constraint Programming into Java. In *Progress in Artificial Intelligence.* Springer, 156–170. https://doi.org/10.1007/978-3-540-24580-3_23

[38] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. 1991. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91).* ACM, New York, NY, USA, 333–352. https://doi.org/10.1145/99583.99627

[39] Joachim Schimpf and Kish Shen. 2012. ECLiPSe – From LP to CLP. *Theory and Practice of Logic Programming* 12, 1 (2012), 127–156. https://doi.org/10.1017/S1471068411000469

[40] Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. 2014. Tor: Modular Search with Hookable Disjunction. *Sci. Comput. Program.* 84 (May 2014), 101–120. https://doi.org/10.1016/j.scico.2013.05.008

[41] Tom Schrijvers, Peter Stuckey, and Philip Wadler. 2009. Monadic constraint programming. *Journal of Functional Programming* 19, 06 (Nov. 2009), 663–697. https://doi.org/10.1017/S0956796809990086

[42] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey. 2013. Search combinators. *Constraints* 18, 2 (2013), 269–305. https://doi.org/10.1007/s10601-012-9137-8

[43] Christian Schulte. 2002. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services.* Lecture Notes in Computer Science, Vol. 2302. Springer. https://doi.org/10.1007/3-540-45945-6

[44] Christian Schulte and Peter J. Stuckey. 2008. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.* 31, 1 (Dec. 2008), 2:1–2:43. https://doi.org/10.1145/1452044.1452046

[45] Dana S. Scott. 1982. Domains for denotational semantics. In *Automata, Languages and Programming*, Mogens Nielsen and Erik Meineche Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 577–610. https://doi.org/10.1007/BFb0012801

[46] Joseph Scott. 2016. *Other Things Besides Number: Abstraction, Constraint Propagation, and String Variable Types.* Ph.D. Dissertation. Acta Universitatis Upsaliensis, Uppsala. OCLC: 943721122.

[47] Helmut Simonis and Barry O'Sullivan. 2008. Search Strategies for Rectangle Packing. In *Proceedings of the 14th International Conference on Principles and*

Practice of Constraint Programming (CP '08). Springer-Verlag, Berlin, Heidelberg, 52–66. https://doi.org/10.1007/978-3-540-85958-1_4
[48] M. B. Smyth. 1978. Power domains. *J. Comput. System Sci.* 16, 1 (1978), 23 – 36. https://doi.org/10.1016/0022-0000(78)90048-X
[49] Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard Von Hanxleden, and Michael Mendler. 2017. SCEst: Sequentially Constructive Esterel. *ACM Trans. Embed. Comput. Syst.* 17, 2, Article 33 (Dec. 2017), 26 pages. https://doi.org/10.1145/3063129
[50] Jean-Ferdy Susini. 2006. The Reactive Programming Approach on Top of Java/J2ME. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '06)*. ACM, New York, NY, USA, 227–236. https://doi.org/10.1145/1167999.1168037
[51] Guido Tack. 2009. *Constraint Propagation – Models, Techniques, Implementation.* Ph.D. Dissertation. Saarland University.
[52] Olivier Tardieu and Robert de Simone. 2005. Loops in Esterel. *ACM Trans. Embed. Comput. Syst.* 4, 4 (Nov. 2005), 708–750. https://doi.org/10.1145/1113830.1113832
[53] Esterel Technologies. 2005. *The Esterel v7 Reference Manual Version v7 30 – initial IEEE standardization proposal.* Esterel Technologies.
[54] Erich C. Teppan, Gerhard Friedrich, and Andreas A. Falkner. 2012. QuickPup: A Heuristic Backtracking Algorithm for the Partner Units Configuration Problem. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI'12)*. AAAI Press, 2329–2334. https://www.aaai.org/ocs/index.php/IAAI/IAAI-12/paper/viewPaper/4793
[55] Charlotte Truchet and Gérard Assayag. 2011. *Constraint Programming in Music.* Wiley.
[56] Kazunori Ueda. 2017. Logic/Constraint Programming and Concurrency: The Hard-Won Lessons of the Fifth Generation Computer Project. *Science of Computer Programming* (2017). https://doi.org/10.1016/j.scico.2017.06.002
[57] Pascal Van Hentenryck. 1989. *Constraint Satisfaction in Logic Programming.* MIT Press.
[58] Pascal Van Hentenryck and Laurant Michel. 2000. OPL Script: Composing and Controlling Models. In *New Trends in Constraints*, Krzysztof R. Apt, Eric Monfroy, Antonis C. Kakas, and Francesca Rossi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 75–90. https://doi.org/10.1007/3-540-44654-0_4
[59] Pascal Van Hentenryck and Laurent Michel. 2005. *Constraint-Based Local Search.* The MIT Press.
[60] Pascal Van Hentenryck, Laurent Michel, and Liyuan Liu. 2005. Contraint-based combinators for local search. *Constraints* 10, 4 (2005), 363–384. https://doi.org/10.1007/s10601-005-2811-3
[61] Pascal Van Hentenryck, Laurent Perron, and Jean-François Puget. 2000. Search and Strategies in OPL. *ACM Trans. Comput. Logic* 1, 2 (Oct. 2000), 285–320. https://doi.org/10.1145/359496.359529
[62] Toby Walsh. 1997. Depth-bounded discrepancy search. In *Proceedings of the 15th International Conference on Artificial Intelligence (IJCAI '97)*, Vol. 97. 1388–1393. http://www.cse.unsw.edu.au/~tw/dds.pdf

A APPENDIX

A.1 Proof of Reactivity

LEMMA A.1. *The semantics of spacetime is reactive.*

PROOF. Given a program p, we can always choose a rule to apply, this is checked by verifying the completeness of the side conditions on rules applying to a same program.

- Axioms NOTHING, PAUSE, STOP and PRUNE are always reactive because they do not have side conditions.
- Axioms SPACE and SPACE-PRUNED derives both pruned and non-pruned branch. In SPACE, enforcing instantaneousness of the body and forbidding writes in `single_space` or `single_time` variables can be statically checked at compile-time.
- `loop` is reactive if the loop is not instantaneous, this can be statically checked at compile-time.
- WHEN-TRUE and WHEN-FALSE are reactive since the entailment operation only maps to *true*, *false* or *unknown* (last both are handled in WHEN-FALSE). Moreover, due to the causality analysis (property 1), the entailment result cannot further change during the derivation.
- Given $p;q$, ENTER-SEQ and NEXT-SEQ are complete on the completion code of p: $U'^k = 0 \vee \neg(U'^k = 0)$ is a tautology.
- PAR$^\vee$ is always applicable.
- Given $p \ || \ q$, PAR$^\wedge$ and EXIT-PAR$^\wedge$ are complete on the completion code of p and q. We have $(U'^k \neq 0 \wedge U''^k \neq 0) \vee (U'^k = 0 \vee U''^k = 0)$ a tautology.
- VAR-DECL↻ is always applicable.
- START-VAR-DECL→↓ and RESUME-VAR-DECL→↓ are complete (either we have a location or a variable name). For RESUME-VAR-DECL→↓, the function *pop* returns $\bot$ if the queue is empty, any variable not defined in a space is mapped to $\bot$ as well (cf. Section 5.1), so the initialization of a `world_line` variable is reactive.
- HCALL depends on the semantics of the host language. The causality analysis guarantees that the function is only called if all its variables can be safely accessed:
  - A `write` access is always possible.
  - For `read` access, we ensure this variable cannot be written anymore in the future (by property 2).
  - For `readwrite` access, only one of such access can happen in an instant (by property 2), and it must happen after every write on this variable.
- REACT and EXIT-REACT are complete on the termination condition. We have ($U'^k = 1$ and Q' is not empty) $\vee$ ($U'^k \neq 1$ or Q' is empty) a tautology.

□

A.2 Proof of Determinism

LEMMA A.2. *The semantics of spacetime is deterministic.*

PROOF. We check that for every rule, at most one rule can be applied to any process p, this is checked by verifying that rules on a same statement are exclusive to each other.

- Rules NOTHING, PAUSE, STOP, PRUNE, LOOP, VAR-DECL↻ and PAR$^\vee$ are deterministic because only one rule can apply.
- Axioms SPACE and SPACE-PRUNED are exclusive on the kind of branch, so it is deterministic.
- WHEN-TRUE and WHEN-FALSE are deterministic since the side conditions on the entailment are exclusive.
- Given $p;q$, ENTER-SEQ and NEXT-SEQ are exclusive on the completion code of p.
- Given $p \ || \ q$, PAR$^\wedge$ and EXIT-PAR$^\wedge$ are exclusive on the completion code of p and q.
- Due to the disjointness of the sets *Name* and *Loc*, we can only apply either START-VAR-DECL→↓ or RESUME-VAR-DECL→↓.
- HCALL is deterministic if the semantics of the host language is deterministic.
- REACT and EXIT-REACT are exclusive on the termination condition.

□

A.3 Scheduling Algorithm

We divide the runtime algorithm into two parts: the execution of several instants in Algorithm 1 and the execution of an instant in Algorithm 2.

Algorithm 1 Runtime engine

Input: A spacetime program p, a space $S \in \mathit{Space}$ and a queue Q.
Output: The triple $\langle p, S, Q\rangle$ such that either p is stopped or terminated, or Q is empty.

procedure EXECUTE(p, S, Q)
    $k \leftarrow 1$ ▷ Completion code initialized to pause.
    if First instant **then**
        $Q \leftarrow push(Q, \{\bot\})$ ▷ Bootstrap the queue with a single element.
    end if
    while $k = 1 \land Q$ is not empty **do**
        $\langle Q, S^{\downarrow}\rangle \leftarrow pop(Q)$
        $S \leftarrow can(p, S)$ ▷ We compute an upper bound on the access counters.
        $\langle p, S, B, k\rangle \leftarrow executeInstant(p, S)$
        $Q \leftarrow push(Q, B)$
    end while
    return (p, S, Q)
end procedure

Algorithm 2 Runtime execution of one instant

Input: A spacetime program p and a space $S \in \mathit{Space}$.
Output: The tuple $\langle p, S, B, k\rangle$ such that B is the set of branches and k the completion code.

procedure EXECUTEINSTANT(p, S)
    $k \leftarrow 3$ ▷ Completion code initialized to stuck.
    while $k = 3$ **do**
        $\langle p, S, B, k\rangle \leftarrow executeProcess(p, S)$
        if $k = 3$ **then**
            $\langle p, S\rangle \leftarrow cannot(p, S)$ ▷ We decrease the upper bound on the access counters
        end if
    end while
    return (p, S, B, k)
end procedure

The first algorithm implements the rules REACT and EXIT-REACT of the behavioral semantics. In addition, it initializes the access counters before each instant with the *can* function.

The second algorithm is the scheduler of the processes inside an instant. It alternates between *executeProcess* and *cannot* until the process is not suspended anymore. Consequently, this function never returns a suspended completion code. The function *executeProcess* is implemented following the same mechanics than SugarCubes [8] and some ideas from ReactiveML [25].