Search Strategies as Synchronous Processes (Extended Abstract)

Pierre Talbot
Université Pierre et Marie Curie
IRCAM
CNRS (UMR 9912 STMS)
talbot@ircam.fr

Abstract

Solving constraint satisfaction problems (CSP) efficiently depends on the solver configuration and the search strategy. However, it is difficult to customize the constraint solvers because they are not modular enough, and it is hard to create new search strategies by composition. To solve these problems, we propose spacetime programming, a paradigm based on lattices and synchronous process calculi that views search strategies as processes working collaboratively towards the resolution of a CSP. We implement the compiler of the language and use it to replace the search module of Choco, a state of the art constraint solver, with an efficient spacetime program that offers better modularity and compositionality of search strategies.

1 Introduction

Backtracking algorithms are the corner stone of exhaustive methods for solving Constraint Satisfaction Problems (CSP). However, their implementations are often not modular or non-compositional which results in solvers hard to read, maintain and extend. It is important to bring modularity and compositionality inside solvers because most CSP are NP-complete in time. In this regard, acceptable performance on large problem instances is usually achieved after the design and test of several solver configurations and search strategies. Indeed, there is no general algorithm working well for every problem and sometimes even the shape of the data can impact the performances. Therefore, we need support to easily try and test the efficiency of new search strategies for a given problem.

Lack of modularity Constraint solvers have evolved through decades of software engineering and research in the field. Despite this considerable effort, their implementations are often monolithic and their core functionalities are entangled [Schrijvers *et al.*, 2013; Michel and Van Hentenryck, 2016]. This lack of modularity prevents the easy selection of the solver components such as no-goods learning, propagation strength or the state restoration policy.

Lack of compositionality Solvers are generally engineered to be open for user-defined search strategies. However, it is difficult to compose two existing strategies without hard-coding the composition by hand. The problem is that search strategies can be composed in various ways that are not easily captured by software abstractions. Examples include sequential composition ; concurrent composition by intersection or by union of the nodes of their search trees; or in an interleaved fashion. Search combinators [Schrijvers *et al.*, 2013] is a high-level language to address this problem but was deemed hard to implement in existing solvers [Rendl *et al.*, 2015]. Moreover, in current approaches, the search strategy is isolated and cannot be concurrently composed. Also, it is not convenient to use and share data across search strategies. Hence, current abstractions in modern solvers fall short to address compositionality of search strategies.

2 Spacetime Programming

Synchronous programming is a paradigm for modeling systems reacting to simultaneous events of the environment—different inputs can arrive at the same time—while avoiding typical issues of parallelism, such as deadlock or indeterminism. In this paper, we focus on the Esterel [Berry, 2000] synchronous language. We demonstrate that this notion of time is ideal to program search strategies over a state-space. To this end, we propose spacetime programming[1], a language that extends the synchronous paradigm to data over complete lattices and with backtracking. We give an overview of the statements in spacetime, and then provide an example. The following statements are standard in the Esterel synchronous language:

- s_1 `;` s_2 is the sequential composition of the two statements s_1 and s_2 ;
- `par` s_1 `||` s_2 `end` is the parallel composition of the two statements s_1 and s_2—called processes ;
- `pause` delays the execution to the next instant ;
- `loop` s `end` executes indefinitely the body s. An instant must always be executed in bounded time so the body s must contain a `pause` statement.

[1]Compiler publicly available at github.com/ptal/bonsai.

The first extension to data over lattices is given by two instructions metaphorically called *ask* and *tell* [Saraswat and Rinard, 1989]. In addition, our language is embedded into the object-oriented Java programming language. Hence, any declared variable has a Java type and any monotonic Java method is callable on this variable.

- `when` a `|=` b `then` s `end` to execute the body s whenever we can deduce b from a in the current instant (entailment relation).
- x `<-` e to augment the information in x with e ; it performs the join operation $x = x \sqcup e$.
- $o.m(e_1, \ldots, e_n)$ to call the method m on the object o with the arguments $e_1 ... e_n$.

As a second extension, we provide a set of operators to dynamically create and prune the state-space. The spacetime paradigm helps to navigate in this state-space by imposing that one state is visited in each instant. Therefore, the processes are synchronized through time and must progress at the same rate. The backtracking extension is provided with the following statements:

- $spacetime\ Type\ var\ =\ e$ declares a variable var of type $Type$ initialized with the expression e. The attribute $spacetime$ refers to the memory of the variable and is explained below.
- `space` s_1 `||` s_2 `end` creates two child nodes of the current state. The statements s_1 and s_2 are executed when the relevant child node is instantiated.

The spacetime attribute specifies how a variable evolves through time and space. For this purpose, a spacetime program has three distinct memories in which the variables can be stored: (1) local memory (keyword `single_time`) for variables local to an instant and re-initialized in each node; (2) global memory (keyword `single_space`) for variables evolving across instants and (3) backtrackable memory (keyword `world_line`) for variables local to a path in the search tree. We exemplify this paradigm by programming a basic constraint solver where two processes collaborate to find the first solution of a CSP:

```
module Solver =
  world_line VStore domains;
  world_line CStore constraints;
  proc search = par propagation() || branch() end
  proc propagation = loop
    domains.propagate( constraints );
    pause;
  end
  proc branch = loop
    single_time IntVar x = domains. fail_first ();
    single_time Integer v = x.middle_value();
    space
    || constraints <- x.leq(v);
    || constraints <- x.ge(v);
    end
    pause;
  end
end
```

The module `Solver` is structured into two fields `domains` and `constraints`, and two processes `propagation` and `branch` implementing a "propagate-and-search" algorithm. Firstly, the types of the fields `VStore` and `CStore` are Java classes abstracting the constraint library Choco [Prud'homme *et al.*, 2015]. Along with the method `propagate` in the process `propagation`, they provide a usable abstraction in spacetime over a state-of-the-art constraint library. Secondly, the state space is generated by the process `branch`. It implements a branching strategy selecting a variable x with the smallest non-singleton domain (function `fail_first`) and the middle value v of this domain (function `middle_value`). The search space is then built with a binary `space` statement where the first branch describes a future where $x \leq v$ and the second branch describes a future where $x > v$. This contributes to improve the modularity of Choco since two individual processes—composed in parallel—work collaboratively to solve a constraint problem.

3 Conclusion

We identified two main problems in the mainstream constraint solvers: the lack of modularity and of compositionality. Spacetime is a paradigm extending the synchronous paradigm with lattices and backtracking. Modularity is achieved by lattice-based variables that only evolve monotonically and that can be safely shared among processes. The synchronous approach solves the compositionality problem by providing a notion of logical time for synchronizing the search strategies viewed as independent processes. This project is the first step to create highly customized constraint solvers. In the long run, it will open the door to machine learning methods to dynamically select the best solver configurations and search strategies for a given problem and from a library of existing strategies.

References

[Berry, 2000] Gérard Berry. The Foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. MIT Press, 2000.

[Michel and Van Hentenryck, 2016] Laurent Michel and Pascal Van Hentenryck. A microkernel architecture for constraint programming. *Constraints*, 2016.

[Prud'homme *et al.*, 2015] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*, 2015.

[Rendl *et al.*, 2015] Andrea Rendl, Tias Guns, Peter J. Stuckey, and Guido Tack. MiniSearch: a solver-independent meta-search language for MiniZinc. In *CP-2015*, pages 376–392. Springer, 2015.

[Saraswat and Rinard, 1989] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *POPL'90*, pages 232–245. ACM, 1989.

[Schrijvers *et al.*, 2013] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter J. Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013.