# Advanced Programming Languages for Artificial Intelligence: Project

Toon Nolten r0258654, Joren Verspeurt r0258417

June 2014

## 1 Introduction

In this assignment we evaluate the usefulness of different constraint programming systems when applied to logic puzzles, specifically sudoku and slitherlink (both by Nikoli). Because of other concurrent projects we had very little time to work on this assigment limiting the quality and quantity of our results.

## 2 Sudoku

### 2.1 Discussion

***ECLiPSe*** Some useful aspects:

- *ECLiPSe* arrays are a very natural way of representing a sudoku grid. The representation of various possible values for one box as the domain of an array element is very elegant. Also indexing over them is very convenient, even more so because of the handy built in iteration mechanisms such as the multifor and the foreach.
- The ic(_global) library contains many constraints that allow the basic rules of the puzzle to be represented in just a couple of lines, such as alldifferent and occurrences.
- The search predicate is powerful and dynamic, allowing for an easy way to test out different search strategies almost instantaneously.

Some disadvantages:

- Continuously switching between arrays and lists gets confusing.
- We tried using structs as a data structure but this didn't fit well with the problem and got pretty confusing at times.
- It's not always clear which comparison operators and such are the correct ones. Also it seems like not every comparison operator has a reified version, which is a little inconsistent.

- Considering the tools that come with the language as a part of the system we'd like to remark that these tools aren't very developer-friendly. The commandline interpreter, for example, seems like it wasn't made to be used at all. And though the GUI version does offer some niceties with respect to tracing, much is left to be desired.
- There are libraries that contain predicates that clash with those in the standard set.

**CHR** Chr allows for constraints regarding the values on a sudoku grid to be easily expressed as chr rules. This is great for problems that can be solved by relying solely on constraints. However for problems like sudoku there is no known set of constraints that solves all puzzles. These kind of problems require search which is hard to do in chr and very hard to do well.

**Jess** Jess is a rule engine, therefore the rules regarding sudoku should be easily expressible in jess. However, like chr jess's weakpoint is problems like sudoku where rules alone do not suffice. Jess has even less support for search as does chr, the standard way of doing it is to delegate it to java, the host language.

## 2.2 Viewpoints and Programs

**Alternative Viewpoint** Instead of considering a map of coordinates to values, we considered a map of values to sets of coordinates. Each of these sets will have size nine and contain only coordinates that do not lie on the same row, in the same column or in the same box as any other in that set.

In chr this is accomplished through a constraint "val_set/3" which holds the value as an identifier, a list of positions that certainly belong to the set and a list of positions which may belong to the set.

In our *ECLiPSe* implementation we did something similar, we used an array where the row index corresponded to a value to be filled in, the column index to the actual column in the sudoku grid and the domain of the variables to the possible rows.

## 2.3 Experiments

In general *ECLiPSe* is much faster than CHR, we are unsure whether this is inherent to these two systems or due to our inexperience.

***ECLiPSe*** Experiments run with our *ECLiPSe* implementation revealed a couple of things:

- The second viewpoint and channeling constraints didn't help the performance much, at least not when using a simple labeling method as search. To the contrary the extra constraints actually slowed the solution down depending on the order of the constraints, doubling the calculation time from about 3.3 seconds to 7 seconds for the hardest problem (sudowiki_nb28 in this case).

- The search strategy matters a lot for the performance: the time for the puzzle that was the hardest for labeling went down from 3.3 seconds to 0.08 seconds when using the "most constrained" variable selection strategy. The value selection strategy matters as well: among the results for the experiments that used the "most constrained" strategy the times still range between 0.08 seconds and 1 second.

**CHR** Only the classical viewpoint has been properly implemented in chr, and attains an average performance around 6s for the sudoku from toledo (the easier ones like "verydifficult" and "expert" take below 0.3s while "extra2" takes 30s). The alternative viewpoint performs really slow, we had to add a much simpler puzzle for it to finish at all, this probably means there are still bugs in the implementation but we could not find these. An attempt has been made to implement the channeling constraints but we also failed doing this, which may be caused by a bug in the alternative viewpoint rules but is probably due to separate mistakes in how the channeling constraints interact in different directions.

## 3 Slitherlink

### 3.1 Implementation in *ECLiPSe*

The standard viewpoint uses 3 arrays: arrays of booleans for the horizontal and vertical edges and an array of integers between 0 and 6 indicating the state of the crossings between the edges. Every integer value represents a valid state of the crossing: 0 for no adjacent edges and the other numbers for 2 adjacent edges in various configurations. Using this representation has the benefit of reducing the amount of additional constraints that need to be placed on the variables but also has the downside of potentially becoming confusing. We planned to add an extra viewpoint that would represent walks between crossings as elements. Every 'certain' edge on the grid would have to belong to one of these walks. When walks meet they are merged into one. This allows for continuous checking of the single cycle constraint and can possibly also be used to detect when the loop is closed too soon.

### 3.2 Slitherlink in CHR

We did not have time to implement slitherlink in chr.

## 4 Appendix

We each worked on this project for 30 hours.