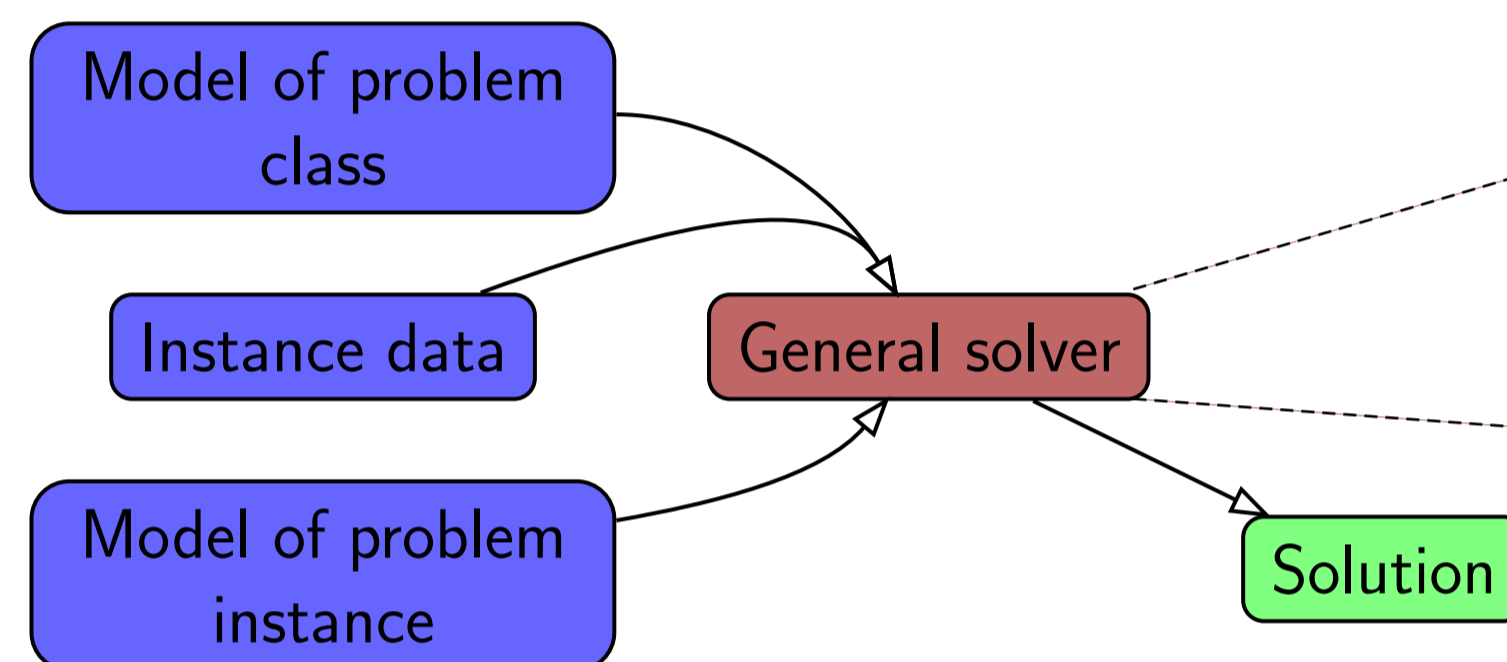


## Constraint solvers – current situation

- ▷ **monolithic** solver (e.g. Minion, Gecode, Choco, Eclipse)
- ▷ **fixed** design decisions
- ▷ **general versions** of propagation algorithms, queue implementation, search, memory management...
- ▷ **limited** amount of tailoring for a problem[7]
- ▷ manual specialisation **difficult** and **very expensive**
- ▷ automatic specialisation **very difficult**



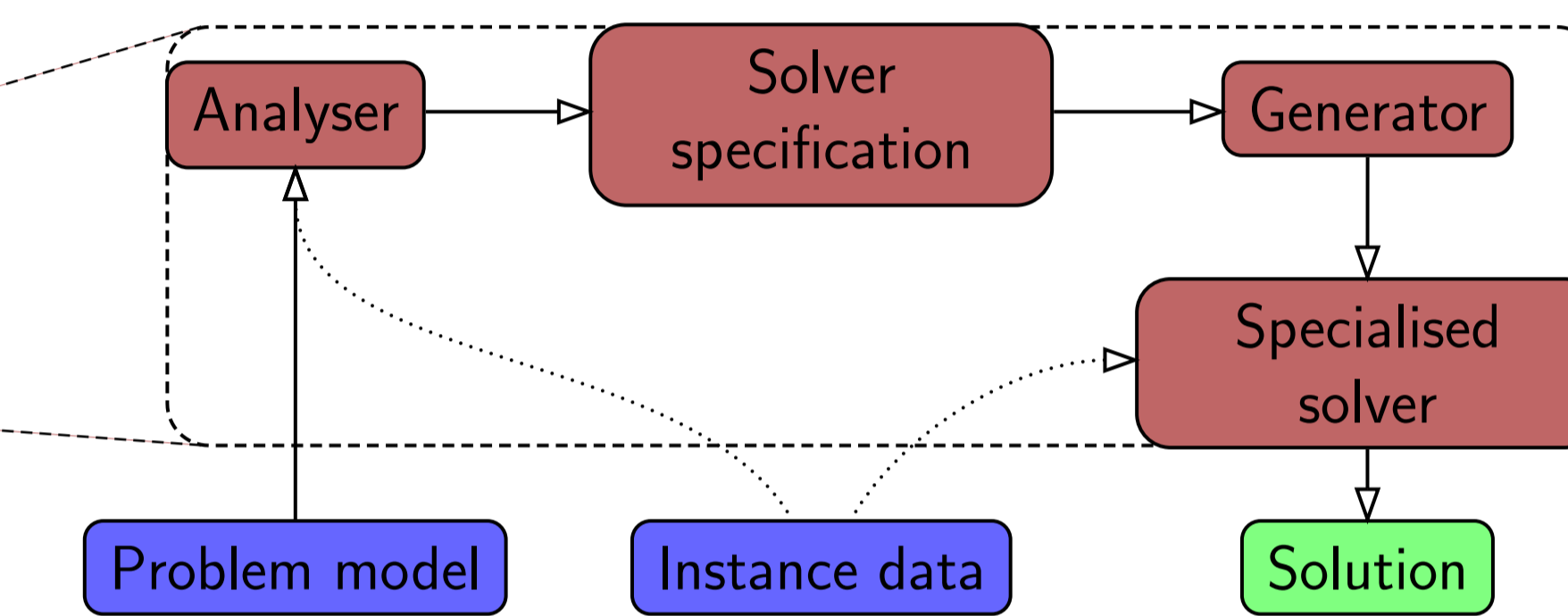
- ▷ design considerations when implementing a solver are for example
  - ▷ trailing vs. copying,
  - ▷ specialised propagator implementations for Boolean variables,
  - ▷ propagator queue order
  - ▷ levels of consistency
  - ▷ search order
- ▷ most of them are **crucial** for the performance on a given problem[5]
- ▷ they can make the difference between being able to solve the problem and not being able to do so

## Idea – constraint solver synthesis

- ▷ instead of using a general solver, synthesise a **specialised** one for the problem
- ▷ choose the **most suitable** algorithms and data structures
- ▷ employ **special techniques** which are usually not implemented in general solvers, e.g.
  - ▷ conflict recording[4],
  - ▷ backjumping[6],
  - ▷ singleton arc consistency[1],
  - ▷ neighbourhood inverse consistency[2]
- ▷ the specialised solver will have a **better performance** than a general solver for the problem it was specialised for

## Dominion architecture

- ▷ two main parts – **Analyser** and **Generator**
- ▷ the two parts are linked by the **Solver specification**, which contains the result of the analysis and the information the generator needs to synthesise a solver
- ▷ solvers can be synthesised for **problem classes** (e.g. n-Queens) as well as **problem instances** (e.g. 3-Queens) with a higher level of specialisation

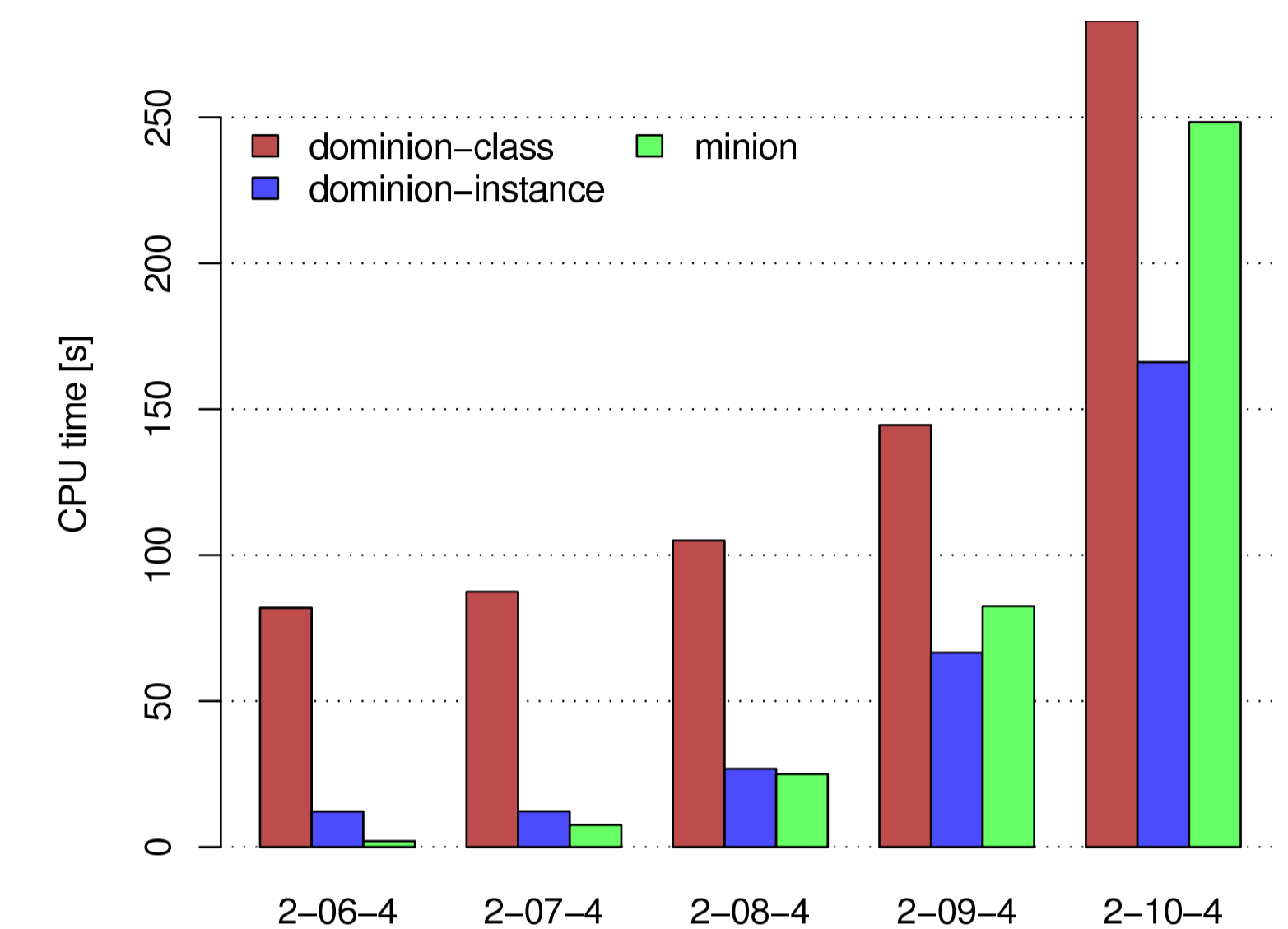


- ▷ the instance data is required by the Analyser or the specialised solver depending on whether an instance solver or a class solver is generated

## Preliminary implementation

- ▷ implementation based on the **Minion**[3] constraint solver
- ▷ the **Analyser** determines the constraints and variable types the problem model requires
- ▷ the **Solver specification** encodes this knowledge and the problem model
- ▷ the **Generator** modifies the Minion source code –
  - ▷ removes modules and infrastructure code which is not required,
  - ▷ generates new variable types for every domain size,
  - ▷ encodes the problem model in file which can be compiled into the solver
- ▷ the modified source can be compiled into
  - ▷ a **class solver**, which is specialised for the analysed problem model and similar ones and retains the input file parser
  - ▷ or an **instance solver**, which, when run, solves the analysed problem
- ▷ the modified solver still finds the correct solutions and compiles several orders of magnitude faster than standard Minion
- ▷ both analysis of the problem model and modifications of Minion are still very limited
- ▷ directions for future work include synthesising a solver from components, investigation of heuristics...

## Experimental results



- ▷ graph shows results for instances of the Social golfers problem
- ▷ numbers include generation time and solve time
- ▷ results for other problem classes are similar

## Conclusions

- ▷ initial results are **very promising**
- ▷ specialised solver is **significantly faster** than standard Minion
- ▷ for large problems, Dominion **wins** even taking the time it takes to generate the specialised solver into account

## References

- [1] Christian Bessière and Romauld Debruyne, **Theoretical analysis of singleton arc consistency and its extensions**, Artificial Intelligence 172 (2008), no. 1, 29–41.
- [2] Eugene C. Freuder and Charles D. Elfe, **Neighborhood inverse consistency preprocessing**, AAAI 1996, pp. 202–208.
- [3] Ian P. Gent, Chris Jefferson, and Ian Miguel, **MINION: A fast scalable constraint solver**, ECAI 2006, pp. 98–102.
- [4] George Katsirelos and Fahiem Bacchus, **Generalized nogoods in CSPs**, AAAI 2005, pp. 390–396.
- [5] Lars Kotthoff, **Constraint solvers: An empirical evaluation of design decisions**, CIRCA preprint, 2009, <http://www-circa.mcs.st-and.ac.uk/Preprints/solver-design.pdf>.
- [6] Patrick Prosser, **Hybrid algorithms for the constraint satisfaction problem**, Computational Intelligence 9 (1993), no. 3, 268–299.
- [7] Andrea Rendl, Ian P. Gent, and Ian Miguel, **Tailoring solver-independent constraint models: A case study with Essence' and Minion**, SARA 2007, pp. 184–199.