Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 10

Conclusions

In this thesis we have explored the possibilities for composing branch-and-propagate constraint solvers from a set of solver “building blocks.” We have taken a practical approach, and implemented OpenSolver, a configurable constraint solving engine that supports a wide range of relevant solver configurations. Performance comparison with state-of-the-art solvers, some of them commercially available systems, demonstrates that the approach leads to realistic and efficient solvers.

In this chapter we review our approach. In Section 10.1 we return to forms of solver composition identified in Section 2.4, and evaluate what has been achieved. In Section 10.2 we compare our approach with other systems, this time taking a broader view than in Section 3.4.2, were we considered only object-oriented frameworks. In Section 10.3 we discuss the limitations of our approach, and identify directions for further research and development. In Section 10.4 we summarize the contributions of our work.

10.1 Composing Constraint Solvers

Propagation

All forms of composition related to constraint propagation that we identified in Sections 2.4.1 and 2.4.2 are supported by OpenSolver. In particular:

- OpenSolver allows that a separate reduction operator is used for each inversion of an (atomic) arithmetic constraint. We can then define a schedule for the operator-based scheduler that respects the hierarchical dependencies between these rules, following from the decomposition of general arithmetic constraints. This way we do not need heavy-weight operators like HC4. Instead, comparable functionality can be composed from the facilities for a decomposition-based approach, and a scheduler.
• Instead of implementing reduction operators for “stronger” forms of local consistency, such as box consistency and shaving, we compose these from the ingredients of the weaker consistency notions plus an operator for nested search. The operator-based scheduler allows for the implementation of priority schemes that postpone the application of computation-intensive operators until no more “cheap” reductions can be made. It does not support the dynamic schemes suggested in [SS04], but OpenSolver itself does not prevent their implementation.

• Because for every variable, a different domain-type plug-in can be used, hybrid solvers are naturally supported.

Search

OpenSolver has rich facilities for the composition of search strategies, as described in Section 2.4.3. We have seen the following examples.

• In Section 4.3 we composed a best-first search strategy according to Warnsdorf’s heuristic for solving the knight’s tour problem from a container, a selector, and an annotation scheme, all having wider applicability than this particular heuristic.

• In Section 6.4 we discussed how the composition of branching strategies, as supported by the SALSA language, can be implemented in OpenSolver.

In retrospect, the proposed mechanism for composing branching strategies could also have been applied to the memory-bounded LDS traversal strategy of Section 4.1.2. We did not exploit the full potential of the adapter mechanism here, and implemented a dedicated container plug-in. Instead, we could have composed it as suggested in Section 2.4.3.

OpenSolver also supports multiple search probes, to simulate parallel search, as suggested in that same section. Compared to schemes like interleaved depth-first search it has the additional benefit that it supports the possibility to abort constraint propagation before a fixed point has been reached. This would prevent that slow convergence in one node of the search tree blocks progress in other nodes. We have not performed experiments in this area, though. Another option would be to actually run a parallel solver on a single processor. In this case the multiplexing between the search probes is taken care of by the operating system, at the cost of unnecessary inter-process communication. In this case load-balancing is not an issue, and the experiments in Chapter 8 suggest that for the benchmark problems used in that chapter, the communication overhead would range from 4% to 16%, depending on the size of the solver configurations that are being communicated.
10.1. Composing Constraint Solvers

Solver Cooperation

OpenSolver is not intended as a framework for solver cooperation. It does not support generic solver cooperation schemes, as do BALI [AM98] and the system of [HSG01].

Other solvers can be embedded in OpenSolver, but this should be the case for all object-oriented frameworks that can be extended with user constraints. Conversely, OpenSolver can also be embedded in other solvers, and we have seen an example of this in Chapter 7. Because the coordination-layer mechanism gives fine-grained control over the solving process, we expect that OpenSolver is more versatile than other solvers in this respect.

For optimization purposes, the coordination layer mechanism makes it easy to incorporate new bounds that are calculated by external solvers. Also the configuration language makes OpenSolver particularly well suited for cooperation with symbolic solvers that are applied as a preprocessing step. In fact, this is how we implemented the decomposition of arithmetic constraints into simple and atomic constraints for our experiments in Chapter 5. We expect that the ability to publish the search frontier, which is also the basis for parallel search in Chapter 8, opens up new possibilities for cooperation with symbolic solvers during the solving process, instead of only as a preprocessing step. Despite the flexibility offered by the coordination layer mechanism, we think that for general cooperation schemes, a proper API would be a valuable addition.

Composition of Parallel and Distributed Solvers

The coordination layer mechanism was specifically designed to support distributed constraint propagation and parallel search, and OpenSolver is well suited for these forms of solver composition. While we did not experiment with distributed constraint propagation after we moved from DICE to OpenSolver (see Section 3.1 and Chapter 9), the design of OpenSolver supports the basic ingredients of distributed constraint propagation algorithms: distributed termination detection, and the communication of domain updates.

For parallel search, we use a time-out mechanism to achieve an implicit load balancing. This gives a very simple implementation of parallel search, but it leads to slightly less user-friendly systems because a tuning factor is introduced. Especially because our experiments indicated that performance is not overly sensitive to the actual time-out value that is used, we expect that it can be tuned automatically, based on heuristics.

For communicating sub-problems, our parallel solver uses the OpenSolver configuration language. We expect this to have wider applicability than parallel search, and that it increases the value of OpenSolver as a software component. We already mentioned the potential for pre-search and in-search transformations of CSPs and solver configurations. In addition, as we discussed in Section 8.6,
it facilitates a checkpoint mechanism, where the search frontier is saved to disk so that a lengthy search process may survive a power failure or system crash. We use the textual format for steering the parallel search, where depending on the level annotation of a subproblem, a breadth-first or depth-first search is performed. We expect that in addition to this automated steering, there are also possibilities for interaction by users. For example, based on an inspection of the nodes, a user might decide to explore a particular subtree first, or to change the branching strategy for some areas of the search tree. The ability to publish the search frontier is essential for all these applications. The textual format of our configuration language increases the possibilities for external processing of the published nodes.

Returning to the subject of distributed constraint solving, OpenSolver is unsuitable for distributed search by asynchronous backtracking, or related algorithms proposed by Yokoo [Yok01]. These algorithms form an alternative approach, where the processes, or agents that are involved asynchronously propose value assignments to their peers with whom they share constraints. These algorithms rely less on termination detection than ours, and hence allow for greater autonomy of agents. It is unclear how the two approaches compare with respect to efficiency and flexibility.

10.2 Comparison with Other Systems

OpenSolver is an object-oriented framework, aiming at reuse of the basic solver design. Through its coordination layer mechanism it can be deployed as a software component in many environments. In Section 3.4.2 we already compared OpenSolver with other component-based toolkits and frameworks, namely ILOG Solver, Koalog Constraint Solver, Elisa, Disolver, Figaro, EasyLocal++, and Localizer++.

In this section we compare it further with two other types of constraint solving environments: logic programming systems, and modeling languages. Both kinds of systems provide declarative languages, that shield the user from implementation details of the computational model that they are based on. In the category of logic programming systems we consider the ECL\PS [WNS97] system. In the category of modeling languages we consider OPL [VH99] and COMET [MVH02a].

ECL\PS is a logic programming system with extensive libraries for constraint solving on various domain types using many different techniques. It supports tree search and local search, and interfaces with CPLEX, an industrial solver for linear and mixed-integer programs. Furthermore it has interfaces for calling routines written in other programming languages, and can also be embedded in programs written in those languages.

OPL is a language for modeling combinatorial problems, and for specifying
search strategies. The modeling facilities include high-level abstractions such as arrays, allowing for very compact and readable specifications that are close to the combinatorial problems that are being modeled. Compared to their implementation in a main-stream programming language the search abstractions are also at a very high level. OPL has dedicated facilities for specifying scheduling and resource allocation problems.

**Comet** is an object-oriented language for constraint-based local search. The modeling facilities are comparable to those of OPL. The facilities for specifying local search are comparable to those that are made available in a C++ environment through Localizer++ [MVH01].

From a certain perspective, the designs of ECL'PS® and OpenSolver are opposites. Both systems mean to provide a platform for composing constraint solvers by arbitrarily combining different solving techniques, but while OpenSolver is part of a toolbox of loosely connected building blocks, ECL'PS® provides an all-encompassing development environment. OpenSolver plug-ins are written in C++, and solver composition is a matter of black-box composition through a configuration language. A full constraint solver would then complete the OpenSolver core with an external user interface that offers modeling facilities. Conversely, in ECL'PS®, everything is done in the same language, which has a Prolog-like syntax. The basic solving algorithms such as branching strategies, the composite solver, and the problem specification are all coded in this language. A library mechanism is available to hide implementation details from users of facilities coded in ECL'PS®.

In our opinion, ECL'PS® and OpenSolver are both good approaches to composing constraint solvers. For users who are not acquainted with logic programming, the ECL'PS® user interface may have a steep learning curve. OpenSolver, on the other hand, is not intended as a full development environment, and can be coupled with a graphical "plug-and-play" user interface, or a modeling language, depending on the intended use. Being a configurable search engine rather than a development environment, OpenSolver allows for composition at a lower algorithmic level. For example, an algorithm like HC4 would be hard-wired in ECL'PS®. We could also introduce a dedicated OpenSolver plug-in for it, but a comparably efficient way to compute hull consistency can be composed from the facilities for decomposition-based hull consistency and a schedule for the operator-based scheduler. ECL'PS® does support parallel search, but being a closed system, we would not have been able to perform our experiments with the time-out mechanism with it.

As a dialect of Prolog, ECL'PS® is also a full programming language. In contrast, languages like OPL and **Comet** are suitable only for specifying combinatorial problems and solving strategies, and provide high-level abstractions for doing so. What is achieved in EasyLocal++ by providing C++ subclasses, can be accomplished directly by programming in **Comet**. OPL, which is targeted
at tree search instead of local search, is at a comparably high level of abstraction. As we discussed in Section 6.4, at this level of abstraction the OpenSolver configuration language should be seen as an assembly language. The rich facilities for specifying search in modeling languages can be implemented only if several assumptions about the variable domain types are made. For this reason, such systems depend on a fixed set of built-in data types. In OpenSolver, almost nothing is fixed. For a coherent set of plug-ins, though, the same facilities could probably be implemented as a compilation step, translating a higher level language to configuration specifications for OpenSolver, which is then used as a constraint solving engine.

10.3 Perspectives

In this section we identify some areas for further research and development related to OpenSolver, and composing constraint solvers in general.

In the first place, only a basic set of constraints has been implemented for OpenSolver. For example, global constraints such as the all_differnt constraint are missing, and mathematical modeling is currently limited to arithmetic constraints. Therefore the range of problems that can currently be solved is limited, especially compared to the commercial systems. This is not a fundamental limitation, though.

Also a modeling environment with a proper user interface is currently missing. For many experiments we used some peripheral programs to generate the configuration file for a problem instance of given dimensions. For other types of problems we wrote converters from standard file formats such as those used in the DIMACS test sets. Instead of these ad-hoc solutions, we would like to have a modeling layer that supports more abstract problem specifications than the OpenSolver configuration language. Our current program for rewriting arithmetic constraints could be used as a basis for this layer. It could well be coupled with a graphical user interface that offers menus from which the user can configure the solver for a given problem.

In Section 4.4 we concluded that the copying state restoration policy of OpenSolver impedes realizing efficient SAT solving schemes. The reason is that such schemes rely on non-chronological backtracking, which in turn is naturally combined with a trailing state restoration policy. It would be interesting to investigate to what extent this policy can be made configurable, like in Figaro [HMN99]. We expect that because this policy is closely related to the implementation of the variable domains, which is not fixed in OpenSolver, this is not straightforward. Another avenue would be to investigate the implementation of non-chronological backtracking on the basis of hierarchical information stored in node annotations.

From an implementation point of view, the following areas need attention:

- A problem with the current annotation mechanism is that two techniques
may have conflicting uses for the annotation field. For example, best-first search, as discussed in Section 4.3 is now incompatible with the round-robin variable selection strategy, discussed in Section 4.1.2. This can be solved using the adapter mechanism, where a special annotation plug-in implements an array of annotations, and other adapters modify the plug-ins that access the annotations to use a specific element of this array. It would be better not to restrict the annotation mechanism to a single slot in the first place. Probably name/value pairs would be a good solution.

- The facility to break out of constraint propagation, and resume the fixed point computation has not been implemented in the operator-based scheduler. We have not experimented with multiple exploration “probes” as suggested at the end of Section 3.2.3.

Finally, we are convinced that we have not exploited the full potential of the combined features of being able to publish the search frontier, and using a textual format for the generated subproblems. As we already indicated in Section 10.1 this has wider applicability than facilitating parallel search, and should allow for forms of solver cooperation that would otherwise require that one solver is embedded in another.

10.4 Summary

Composing constraint solvers based on tree search and constraint propagation through generic iteration leads to efficient and flexible constraint solvers. This was demonstrated using OpenSolver, an abstract branch-and-propagate tree search engine that supports a wide range of relevant solver configurations. We gave an account of the design and implementation, and of many experiments that were performed to evaluate the approach.

The efficiency of OpenSolver-based constraint solvers compares well to that of existing solvers, some of which are successful commercial products. Yet, the following combination of features gives OpenSolver some unique advantages over each of the other systems that we considered.

- OpenSolver is a branch-and-propagate constraint solving engine, and makes configurable those aspects that are hardwired in many other systems, such as the constraint propagation algorithm and implementation of the data types. Modeling languages can be implemented on top of it, by means of a compilation step.

- OpenSolver promotes the composition of complex strategies from atomic solver “building blocks,” implemented as plug-ins. This prevents duplicate solver code, and allows that techniques carry over to other data types and application domains.
• While it is a white-box framework from the perspective of writing new plug-ins for it, OpenSolver aims at black-box composition of solvers. This led to an inherently linguistic approach where solvers are composed through scripts in a simple configuration language.

• OpenSolver is designed as a stand-alone application. Compared to libraries for constraint solving, this makes it independent of a particular programming language. Composing constraint solvers around OpenSolver is primarily a matter of exogenous coordination and component-based software engineering. This is opposite to the approach taken by logic programming systems, which provide a full development environment.

• Because of the coordination layer mechanism, OpenSolver can be adapted to many different environments, and the solving process can be controlled externally. In particular, it is suited for distributed constraint solving.

• In combination with the configuration language, the coordination layer mechanism opens up new possibilities for implementing parallel search, in-search transformation of CSPs and solver configurations, and it allows for checkpoint mechanisms.

Thanks to the flexibility of the system, we could further achieve the following results related to the specific research questions addressed in some of the individual chapters.

• We demonstrated how a number of techniques that are normally hard-wired in solvers can be realized through composition.

• We performed a comparative study of several approaches to implementing arithmetic constraints on variables with integer interval domains. The best performance was observed for decomposition and hierarchical scheduling of the reduction operators. For this approach we characterized in part the effect of constraint propagation.

• We demonstrated the technique of constraining special purpose domain types.

• We demonstrated that several pruning techniques from various application domains can be expressed and implemented as applications of a generic operator for nested search.

• We evaluated a novel time-out mechanism for load balancing in parallel search, and demonstrated that it can lead to efficient and scalable parallel constraint solvers.