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.
The subject of this thesis can be categorized as the implementation of constraint solvers. In this chapter we put constraint solving, and the programming style that it enables into context. We motivate our work, give an outline of the thesis, and summarize the scientific contributions.

1.1 Context

*Constraint programming* falls in the category of *declarative* programming styles, where essentially a program describes *what* must be computed, as opposed to *imperative* programming, where a program describes *how* the output must be computed. A constraint “program” is called a *constraint satisfaction problem* (CSP), and consists of the following elements.

- a set of variables, each with a set of allowed values, and
- a set of constraints, where every constraint applies to a subset of the variables, and restricts somehow the values that these variables may assume.

Writing a CSP to represent a combinatorial problem that occurs in practice is called *modeling*. The purpose of *constraint solving* is to generate *solutions* to CSPs. These are assignments of values to variables that satisfy all constraints. We will also consider a modification of constraint satisfaction problems where the goal is to generate an optimal solution, according to some objective function. The combination of a CSP and an objective function is called a *constrained optimization problem* (COP). Programs, procedures, and algorithms for constraint solving are called *constraint solvers*. For a tutorial, and textbooks on constraint programming, the reader is referred to [Smi95, Apt03, Dec03]. The subject of this thesis is the implementation of constraint solvers.

Because of the very general nature of a CSP, constraint programming subsumes several other forms of modeling, such as linear programming. For some
specific forms of CSPs and COPs there exist very efficient solving methods, such as the simplex algorithm for linear programs. In this thesis we deal with general methods for constraint solving that are applied only when no efficient, problem-specific methods are available.

In particular, we deal with methods that are based on an exhaustive exploration of all possible assignments of values to variables. This excludes so-called local search methods, which start from an initial assignment, and try to improve it according to some notion of quality by iteratively making small (local) modifications. Local search methods naturally apply to COPs, but they can also be applied to CSPs by minimizing the number of violated constraints. Local search methods typically find good solutions quickly, but are not guaranteed to find the optimum, and therefore they cannot determine whether a solution to a CSP exists or not.

In contrast, the exhaustive methods that we consider in this thesis are guaranteed to find a solution if it exists, but in general they have a time complexity that is exponential in the size of the problem. Although computations with an exponential time complexity are considered to be intractable, this characterizes only the worst-case behavior, and despite their potentially intractable nature, such methods are successfully applied in practice. Examples of such applications are the generation of test patterns for digital circuits (see, e.g., [VHSD92]), the analysis of nonlinear functions [HMD97], and scheduling problems [BLPN01], such as sports tournament scheduling [Hen01].

### 1.2 Motivation

Constraint solving is based on a collection of largely independent techniques, that fall into two categories: search methods, and techniques for reducing the search space. In the context of constraint solving, techniques in the latter category are usually called constraint propagation techniques, and the approach to constraint solving that is considered in this thesis is known as branch-and-propagate search. To a large extent, constraint programming consists of determining the combination of techniques that make a given CSP solvable. Many of these techniques have successfully been used in specific domains to build practical constraint programming tools, but in general it is not possible to combine them without reprogramming or re-engineering the tools, and a major challenge in this field is how to achieve a combination of various existing methods and techniques within a single framework. We will refer to realizing such combinations as solver composition.

Recently, it was demonstrated that many constraint propagation algorithms proposed in the literature are actually instances of a generic iteration algorithm [Apt99, Gen02]. In this scheme, a constraint solver consists of a scheduler combined with the functions that are to be scheduled. If certain conditions are met,
different schedulers will lead to the same result, and the schedulers can be configured to exploit several properties of the functions. Moreover, distributed versions of the generic iteration algorithm exist [MR99, Mon00a], and distributed constraint propagation can be realized simply by substituting a sequential scheduler by a distributed scheduler.

Our goal is to exploit the conceptual simplicity of this scheme, and to complement it with facilities for search to form an integrated framework for constraint solving. The central theme of this thesis is an exploration of the possibilities and limitations of such a framework. We have taken a practical approach, which has led to the development of OpenSolver, a highly configurable constraint solving engine that supports a wide range of relevant solver configurations. A major design goal was that OpenSolver can be used as a software component, to form the core of a solver, and to participate in several solver cooperation schemes. As a result, composing constraint solvers around OpenSolver involves various methods of software composition. We give an account of the design and implementation of OpenSolver, and of the experiments that were performed to verify the efficiency of the resulting constraint solvers.

1.3 Outline of the Thesis

The thesis can be read as a description of the OpenSolver software, plus a series of case studies that demonstrate how it can be configured for various application domains and methods of constraint solving. In addition to exploring the possibilities and limitations of our approach, some of these case studies address more specific research questions. The following chapters fall naturally in three parts. The first part contains the preliminary material:

Chapter 2. Here we introduce constraint solving, and branch-and-propagate search. To motivate our work further, we review several forms of solver composition that are found in the literature.

Chapter 3. A description of the OpenSolver software, which is used in the subsequent chapters as a platform for experimenting with solver composition. Part of the material presented here was submitted to the CP 2003 doctoral program, and an abstract appeared in the conference proceedings [Zoe03c].

The second part consists of three chapters that describe applications of a single OpenSolver instance:

Chapter 4. Here we demonstrate how OpenSolver can be configured as a solver for constraints on finite domains, real, and Boolean variables. We also describe a number of general-purpose facilities that are independent of the application domain, and we investigate how a number of existing techniques that are normally hard-wired in solvers can be realized through composition.
Chapter 5. In this chapter we demonstrate how OpenSolver can be configured for solving arithmetic constraints on integer variables. Two specific questions are addressed in this chapter: first, there are a number of natural approaches to implement these constraints, and we investigate which one of them can be expected to give the best performance. Second, we try to characterize the effect of constraint propagation for these constraints. This chapter is based on joint work with Krzysztof Apt, the preliminary results of which appear in [AZ04].

Chapter 6. This chapter is a case study describing a solver for the job-shop scheduling problem, based on OpenSolver. Like Chapter 4, this involves existing solving techniques, but this material serves two other purposes. First, job-shop scheduling is considered to be a non-trivial problem whose complexity is representative of many scheduling problems that occur in practice. As such, it shows that our approach leads to realistic solver implementations. Second, it demonstrates a technique that we refer to as constraining special-purpose data types. In an open-ended solver, this technique can be applied when the existing facilities do not support an efficient implementation of a solver for a given problem. Part of the material in this chapter was submitted to the CP 2004 doctoral program, and an abstract appeared in the conference proceedings [Zoe04b].

In computer science, coordination refers to the orchestration of the interaction among the various components of a software system [Arb98]. While all meaningful computation involves coordination, it is a particularly relevant aspect of concurrent systems, where several computations overlap in time. The design of OpenSolver allows that in addition to its use as a stand-alone constraint solver, it can be coordinated from the outside in many different ways, to support solver composition at a higher level. In the last part of the thesis we look at several ways in which OpenSolver instances can cooperate to solve a single constraint satisfaction problem. The techniques described here are orthogonal to those of the three preceding chapters, and can be used in combination with them. Two of these techniques involve concurrency, and the emphasis of their presentation is on the coordination aspects.

Chapter 7. There exist constraint propagation techniques that internally involve search. This internal search process occurs in the context of another, encompassing branch-and-propagate search, and is therefore called nested search. In this chapter we propose a generic reduction operator for nested search, and investigate the extent to which three existing techniques from different application domains can be expressed as applications of this generic operator. We also describe an implementation of the operator based on an almost autonomous OpenSolver instance, and we evaluate its performance on some benchmark problems. This chapter is based on a paper [Zoe04a]
that was presented at the 2004 ERCIM/CologNet workshop on constraint solving and constraint logic programming.

Chapter 8. Parallel processing is used to reduce the turn-around time by distributing the workload among a number of threads or processes running on different processors or computers. In this chapter we describe a parallel constraint solver that uses a time-out mechanism for load balancing. To our knowledge, this is a novel approach to parallel search that supports the composition of a parallel solver from autonomous component solvers. The research question addressed in this chapter is whether this approach leads to efficient and scalable parallel solvers. This chapter is based on a paper with Farhad Arbab [ZA04].

Chapter 9. Here we discuss the use of OpenSolver as a software component for implementing a solver based on distributed constraint propagation. Several researchers have recognized the need for such a solver. A possible motivation is that in some cases, the CSP that we are trying to solve is distributed, while it is impossible, or undesirable to gather all constraints in a single solver. The chapter is based on two publications [Zoe03b, Zoe03a], that continue the research of Eric Monfroy and Farhad Arbab in the area of coordination-based constraint solving [Mon00a, AM00].

In Chapter 10 we review the material in the preceding chapters, and suggest directions for future work.

1.4 Contributions

This thesis demonstrates that solver composition can lead to efficient branch-and-propagate constraint solvers. Specific contributions are the following.

- An account of the design and implementation of a general purpose constraint solving engine, with a flexible architecture that supports a wide variety of relevant solver configurations. In particular
  - It is configurable with respect to low-level aspects such as the scheduling of the functions that implement constraint propagation. This allows the composition of techniques that are normally hard-wired in constraint programming tools. Solver composition in turn leads to reuse of code, and allows that solving techniques carry over to other data types and application domains.
  - It is designed as an autonomous application that communicates with its environment through a programmable interface and a solver configuration language. This design facilitates the component-based construction of constraint solvers around the solving engine, independent
Chapter 1. Introduction

of a particular computing environment or programming language. The configuration language gives unique possibilities for external manipulation of CSPs and solver configurations, which allow that special-purpose functionality can be implemented outside the solving engine.

- A demonstration and discussion of the technique of constraining special purpose data structures as a tool to implement solvers for problems that do not have a straightforward CSP formulation.

- A systematic study of several approaches to implementing arithmetic constraints on integers, using an interval representation for the variable domains, and integer interval arithmetic to describe and implement constraint propagation. For the most promising approach, we provide results that characterize the effect of constraint propagation.

- A demonstration that several operators for enforcing so-called stronger forms of consistency, which improve the efficiency of constraint solving in specific application domains, are actually instances of the same technique: nested search. We demonstrate that using a generic reduction operator for nested search, these operators can be \textit{composed} from the operators that enforce weaker forms of consistency. Experiments show that this compositional approach leads to a viable implementation of the techniques that we are interested in.

- A study of a time-out mechanism for implementing parallel search. We demonstrate that by equipping constraint solvers with a time-out mechanism, these solvers can then be used as software components for building a parallel constraint solver, resulting in a very simple implementation that performs well on shared memory and distributed memory architectures, and gives a good load-balance in practice.

In addition, we believe that the OpenSolver software itself is potentially of interest to a wider audience. However, it was developed primarily for the experiments reported in this thesis, and has not been used for other purposes. Consequently, ease of use has not been a priority, and no user's manual or programmer's manual exists. The latter would be essential for exploiting the open-ended nature of the system. If time permits, we hope to be able to continue the development of OpenSolver, and to make it available as open source software.