Foreword

Constraint Programming is an approach for modeling and solving combinatorial problems that has proven successful in many applications. It builds on techniques developed in Artificial Intelligence, Logic Programming, and Operations Research. Key techniques are constraint propagation and heuristic search.

Constraint Programming is based on an abstraction that decomposes a problem solver into a reusable constraint engine and a declarative program modeling the problem. The constraint engine implements the required propagation and search algorithms. It can be realized as a library for a general purpose programming language (e.g. C++), as an extension of an existing language (e.g. Prolog), or as a system with its own dedicated language.

The present book is concerned with the architecture and implementation of constraint engines. It presents a new, concurrent architecture that is far superior to the sequential architecture underlying Prolog. The new architecture is based on concurrent search with copying and recomputation rather than sequential search with trailing and backtracking. One advantage of the concurrent approach is that it accommodates any search strategy. Furthermore, it considerably simplifies the implementation of constraint propagation algorithms since it eliminates the need to account for trailing and backtracking.

The book investigates an expressive generalization of the concurrent architecture that accommodates propagation-preserving combinators (known as deep guard combinators) for negation, disjunction, implication, and reification of constraint propagators. Such combinators are beyond the scope of Prolog's technology. In the concurrent approach they can be obtained with a reflective encapsulation primitive.

The concurrent constraint architecture presented in this book has been designed for and realized with the Mozart programming system, where it serves as the basis for new applications and tools. One example presented in this book is the well-known Oz Explorer, a visual and interactive constraint programming tool.

The author of this book, Christian Schulte, is one of the leading experts in constraint technology. He also is one the creators of the Mozart programming

system. His book is a must read for everyone seriously interested in constraint technology.

December 2001 Gert Smolka

Preface

Constraint programming has become the method of choice for modeling and solving many types of problems in a wide range of areas: artificial intelligence, databases, combinatorial optimization, and user interfaces, just to name a few. In particular in the area of combinatorial optimization, constraint programming has been applied successfully to planning, resource allocation, scheduling, timetabling, and configuration.

Central to the success of constraint programming has been the emphasis on programming. Programming makes constraint-based modeling expressive as it allows sophisticated control over generation and combination of constraints. Programming makes an essential contribution to the constraint solving abilities as it allows for sophisticated search heuristics.

Todays constraint programming systems support programming for modeling and heuristics. However, they fall short for programming search strategies and constraint combinators. They typically offer a fixed and small set of search strategies. Search cannot be programmed, which prevents users from constructing new search strategies. Search hard-wires depth-first exploration, which prevents even system developers from constructing new search strategies. Combination is exclusively based on reification which itself is incompatible with abstractions obtained by programming and often disables constraint solving when used for combination.

The main contribution of this book is easy to explain: constraint services such as search and combinators are made programmable. This is achieved by devising *computation spaces* as simple abstractions for programming constraint services at a high level. Spaces are seamlessly integrated into a concurrent programming language and make constraint-based computations compatible with concurrency through encapsulation.

State-of-the-art and new search strategies such as visual interactive search and parallel search are covered. Search is rendered expressive and concurrency-compatible by using copying rather than trailing. Search is rendered space and time efficient by using recomputation. Composable combinators, also known as deep-guard combinators, stress the control facilities and concurrency integration of spaces. Composable combinators are applicable to arbitrary abstractions without compromising constraint solving.

The implementation of spaces is presented as an orthogonal extension to the implementation of the underlying programming language. The resulting implementation is shown to be competitive with existing constraint programming systems.

Acknowledgments

First and foremost, I would like to thank Gert Smolka, for his unfailing support, expert advice, and incomparable enthusiasm during my doctoral research. I am extraordinarily grateful to Seif Haridi, who amongst other things accepted to give his expert opinion on the thesis on which this book is based: I do consider Seif as my second thesis adviser. Gert, in particular, introduced me to constraint programming and research in general, and taught me that striving for simplicity is essential and fun. I am still grateful to Sverker Janson for convincing me that the solve combinator is a suitable starting point for a thesis.

I am lucky to have had the great advantage of having been surrounded by knowledgeable and helpful co-workers: Martin Henz, Tobias Müller, Peter Van Roy, Joachim P. Walser, and Jörg Würtz have been inspiring partners in many fruitful discussions on constraint programming; Michael Mehl, Tobias Müller, Konstantin Popov, and Ralf Scheidhauer shared their expertise in implementation of programming languages and constraints with me; Denys Duchier, Seif Haridi, Martin Henz, Michael Mehl, and Gert Smolka have been helpful in many discussions on spaces. I have benefitted very much from the knowledge – and the eagerness to share their knowledge – of Per Brand, Seif Haridi, Sverker Janson, and Johan Montelius on AKL. Konstantin Popov has been invaluable for this work by implementing the first version of the solve combinator.

Thorsten Brunklaus, Raphaël Collet, Martin Henz, Tobias Müller, Gert Smolka, and Peter Van Roy provided helpful and constructive comments on drafts of this work. Their comments have led to fundamental improvements.

April 2001 Christian Schulte