Dominion: An Architecture-driven Approach to Generating Efficient Constraint Solvers

Dharini Balasubramaniam, Lakshitha de Silva, Chris Jefferson, Lars Kotthoff, Ian Miguel and Peter Nightingale
School of Computer Science, University of St Andrews,
North Haugh, St Andrews, Fife KY16 9SX, UK
Email: {dharini, ldrs, caj, larsko, ianm, pn}@cs.st-andrews.ac.uk

Abstract—Constraints are used to solve combinatorial problems in a variety of industrial and academic disciplines. However most constraint solvers are designed to be general and monolithic, leading to problems with efficiency, scalability and extensibility. We propose a novel, architecture-driven constraint solver generation framework called Dominion to tackle these issues. For any given problem, Dominion generates a lean and efficient solver tailored to that problem. In this paper, we outline the Dominion approach and its implications for software architecture specification of the solver.

I. INTRODUCTION

Constraints are a natural and powerful means of representing and reasoning about combinatorial problems. Constraint solving provides a means of finding solutions to such problems automatically. Its simplicity and generality are fundamental to its successful application in a wide variety of disciplines such as scheduling, industrial design, aviation, banking, combinatorial mathematics and the petrochemical and steel industries [1].

Constraint solving of a combinatorial problem proceeds in two phases. First, the problem is modelled as a set of decision variables and a set of constraints on those variables that a solution must satisfy. A decision variable represents a choice that must be made in order to solve the problem. The domain of potential values associated with each decision variable corresponds to the options for that choice. The second phase consists of using a constraint solver to find solutions to the model: assignments of values to decision variables satisfying all constraints. Constraint solvers typically employ a systematic backtracking search through the space of partial assignments in order to find solutions.

Most current constraint solvers, such as Minion [2], are constructed to be as general as possible. They are monolithic in design, accepting a broad range of models. While this generality is convenient, it leads to a complex internal architecture, resulting in significant overheads and inhibiting efficiency, scalability and extensibility. Another drawback is that current solvers perform little analysis of an input model, so the features of an individual model cannot be exploited to produce a more efficient solving process. To mitigate these drawbacks, constraint solvers often allow manual tuning of the solving process. However, this requires considerable expertise, preventing the widespread adoption of constraint solving.

The main aim of the work introduced in this paper is to improve the scalability of constraint technology, while simultaneously removing its reliance on manual tuning by an expert. We propose a novel, elegant means to achieve this aim: a constraint solver generator framework called Dominion which, for a given problem, produces a solver tailored to that problem. There are two key benefits to this approach:

1) it enables fine-grained optimisations not possible for a general solver, allowing the solution of much larger, more difficult problems, and
2) it enables the utilisation of many techniques in the literature that, although effective in a limited number of cases, are not suitable for general use, leading to more powerful solvers.

The Minion solver allows some specialisation of components. There are 7 variable types, and each constraint is compiled for two inputs which can each be different variable types, therefore currently each constraint is compiled 49 times. Adding one extra option to variables (doubling the number of variable types) increases compilation time fourfold. In contrast, Dominion compiles exactly the required variables and constraints for each problem.

The generation process in Dominion is driven by the software architecture of the target solver. Software architecture provides a high-level model of the structure and behaviour of a system in terms of its constituent elements and their interactions as well as conditions that have to hold among the elements [3], [4]. Desired properties of a system can be checked at the architectural level as well as against an implementation. Thus the architecture forms a useful basis and guide for the implementation and evolution of systems.

A number of Architecture Description Languages (ADLs) have been developed to capture these features [5]. We use a powerful, general-purpose ADL called Grasp [6] in this work. Grasp provides support for commonly used generic architectural primitives such as layers, components, connectors, interfaces, properties and wiring as well as the ability to define user-defined constructs for particular domains. It allows the specification of templates which can be instantiated to produce the required architectural elements. Most importantly, unlike many existing ADLs, Grasp has been designed to capture sophisticated dependencies among architectural elements. These features make Grasp ideally suited to represent the architectures of customised constraint solvers in Dominion.

In this paper, we present an outline of our approach and focus on its implications for software architecture specification.
II. RELATED WORK

One of the earliest examples of systems that attempt to
generate constraint solvers tailored to a specific problem is
the MULTI-TAC system [7], which configures and compiles a
constraint solver for a specific set of problems. It is written in
LISP and performs ad-hoc customisation of a base constraint
solver limited to a few characteristics.

KIDS [8] is a more general system that also uses LISP
to synthesise efficient algorithms from an initial specification.
The approach is knowledge-based, i.e. the user supplies the
knowledge required to generate an efficient algorithm for the
specific problem. Refinements are limited to a number of
generic transformation operations. Our approach is more gen-
eral and, crucially, relies on almost no background knowledge.

There are other constraint solver systems that perform code
generation or modification, but to the best of our knowledge
no previous approach to formally specifying the architecture
of specialised constraint solvers exists.

III. AN EXAMPLE

The following listing shows a very short example of a
constraint problem in the Dominion Input Language (DIL) [9]
and will be used in the remainder of the paper to illustrate architectural support for the Dominion approach.

```
language Dominion 0.1
given b : int {1..}
given s : int {1..}
find w : int {2}
find x : int {0..1}
find y : int {0..b}
dim z[a] : int
find z[..] : int {a, 0..1}
such that
conA sum([x,y], w)
conB sum([z[..], w])
```

Listing 1: A Constraint Problem

The basic format of a problem is to give a list of parameters
given), decision variables (find), their domains and finally
a list of constraints the variables must satisfy. z is a one
dimensional matrix of size s. Given the values \( b = 2 \) and
\( s = 2 \) for the parameters, one solution to this problem is
\( w = 2, x = 1, y = 1, z[0] = 1, z[1] = 1 \). Different stages of the
Dominion solver generator process will be illustrated using this example, beginning with a description of issues that arise
in modelling constraint solvers.

IV. ISSUES IN MODELLING CONSTRAINT SOLVERS

A constraint solver is typically built from a number of
factories, which create the various parts of the solver and the
connections between them.

Two of the most important parts of a constraint solver are
the components which represent the (decision) variables and
constraints. Variable components maintain a set of values the
variable might take in a solution (the domain). The constraints
query variable domains, and remove values that can take part
in no solution.

To illustrate some of the key problems in architectural
modelling of constraint solvers, we turn to the Minion solver,
introduced earlier. In Minion there are several variable factories,
producing variables with different features and performance
characteristics. For example, there is a factory for variables with
initial domain \( \{0,1\} \). Some other variable factories require the
initial domain to be an unbroken range. Some implementations
permit values to be removed from the middle of the domain,
while others only allow changes at the upper and lower bounds.
Constraints place callbacks on variables to indicate when they
should be informed of changes. Some constraints change the
callbacks during search (dynamic callbacks) while others do not. It would be desirable to specialise variables on whether
they allow dynamic callbacks. Minion does not do so because
each specialisation adds to its compile time.

Minion may query variable domains hundreds of millions
times per second, therefore tiny changes to variable imple-
mentations can lead to large changes in solver performance.

Constraints also have restrictions upon them. As an example,
consider the constraint \( \sum X = y \), where \( X \) is a vector of
variables. The components below, among others, have all been
implemented in Minion.

1) \( \text{GACSum} \): An exponential time algorithm which reduces
domains as much as possible. Requires all variables to
allow arbitrary domain removals.
2) \( \text{BoundSum} \): A generic polynomial time algorithm.
3) \( \text{BoolSum} \): Requires all variables in \( X \) to have domain
\( \{0,1\} \).
4) \( \text{BoolSumConst} \): Requires all \( X \) to have domain \( \{0,1\} \),
and support dynamic callbacks. Also the domain of \( y \)
must be a single value.

Some requirements, such as dynamic callbacks or arbitrary
domain removals, are restrictions on the type of variable used.
Other requirements, for example in \( \text{BoolSum} \) that variables
have domain \( \{0,1\} \), are not restrictions on the type, but on the
variable. While there are variable factories which only produce
variables with domain \( \{0,1\} \), any variable with domain \( \{0,1\} \)
will be accepted by \( \text{BoolSum} \).

These options are wired into Minion in different ways.
\( \text{GACSum} \) can be explicitly chosen by users; others are chosen
using simple heuristics, known to be incorrect in some cases.
As these heuristics may pick an implementation which requires
dynamic callbacks, all variables are required to support them.

Thus, an important requirement of an architecture description
of solvers is the ability to associate properties with components
and specify and check dependencies among them.

V. ARCHITECTURE SUPPORT FOR DOMINION IN GRASP

Software architecture is the main driver of solver generation
in Dominion and contains much of the vital information
required for the process. This has implications for the expressive
power of the chosen ADL. In addition to customary details of
components and connections, the Dominion approach requires
further support from the ADL in order to automate the
process of generating an optimal architecture from the problem
component, and the solver code from the architecture.
Grasp is a general purpose textual ADL designed to capture architecture rationale as well as structural and behavioural aspects. Supported architectural primitives include layers, components, connectors, templates, interfaces, rationale, wiring, properties and check clauses. Templates are abstractions for architectural elements and can be used to create instances with common behaviour. The required and provided interfaces allow tools to match the functionality of linked components. Properties are characteristics (or restrictions of functionality) associated with architectural elements in the form of name-value pairs, while check clauses allow properties of parameters and linked elements to be checked for compatibility. Grasp also provides a generic annotation mechanism to associate metadata with architectural elements. In Dominion, this mechanism is used to specify file names and locations of corresponding implementations for components. In Dominion, this mechanism is used to specify file names and locations of corresponding implementations for components.

Listing 2 shows a template definition in Grasp for a factory that creates boolean decision variables, which implement the interfaces IPropVariable and IRemoveFromDomain, and require to be connected to a component that implements IMemoryManager. The check statement means that the memory manager can only have the property (MemoryChanges, ‘Single’). The domain size of boolean variables is 2.

The Grasp toolset currently consists of a parser and a checker. Further tools for visualising and validating architectures as well as performing traceability analysis are planned for the language.

VI. THE GENERATIVE PROCESS

The process of generating an efficient and lean solver for a given problem is driven by its software architecture and contains the following steps as illustrated in Figure 1:

- Problem component generation,
- Architecture generation and analysis,
- Solver generation, and
- Execution monitoring

The overall process can be considered as a control loop with the problem specification initiating feedback control and data on the execution of the solver leading to feedback control. C++ is used as the implementation language for performance, modularity and backwards compatibility reasons. The solver is developed using component-based software engineering practices. The component library, which is used throughout the solver generation process, is introduced first. The first 3 steps shown in Figure 1 is discussed in the later subsections. Execution monitoring will be added as part of future work.

A. Component Library

Components form the building blocks of Dominion constraint solvers. A large number of reusable components are kept in a library to aid the different stages of the generation process. The Dominion component library consists of two parts: the specification of components as elements at the architectural level and the corresponding implementation of these components at the code level. Listing 2 is an example of the architecture level specification of a component template. Each component is implemented as a C++ class, appropriately parameterised to allow customisation as required. Each class is stored in a separate file, increasing flexibility of use and modularity. This practice also allows the generation of the final solver to be automated in a relatively straight-forward manner.

B. Problem Component Generation

The generative process is initiated by a constraint problem being supplied to the problem component generator tool. This tool parses the DIL specification and defines a new component in Grasp capturing the essence of the problem such as interfaces to be supported by variable and constraint components and any restrictions, such as domain compatibility, among them. Each problem component is unique to the problem being solved and thus such components are not stored in the component library.

For the example from Listing 1, the problem component generator will produce the specification shown in Listing 3.

The problem component will require appropriate variable and constraint components. The check clauses of this component can be divided into two parts. The first places restrictions on the variables based on the domains they will have. For some variables, such as $w$, the exact domain and size of domain is known. For others such as $y$, the type of domain is known but not its size. Thus the choice of component for the variables is constrained to those which implement the required domain.

The second part restricts the choice of implementation for the constraints by checking each value we will give for each parameter. The $+$ operator is used to attach properties to a component. We know in this particular problem that the domain of $x$ will be $\{0, 1\}$. Regardless of the chosen implementation $\text{pvx}$ for $x$, which may or may not make use of this restriction, any variable produced by $\text{pvx}$ will have the domain $\{0, 1\}$. 
The relevant template specifications are not shown due to lack of space.

The work on the analyser tool is ongoing. In essence, this tool will generate a list of candidate solver architectures and select the best one using artificial intelligence techniques.

Listing 4 shows a possible solver architecture produced by the analyser for the problem from Listing 1, including component instantiations and configurations using `link` statements. The relevant template specifications are not shown due to lack of space.

```
@Dominion(Location = "/lib/src/*", FileName = "problem1.hpp")
template ThisProblem()
{
    requires IPropVariable pvw, pvx, pvy, pvz;
    requires ISumCon scA, scB;

    check pvw->properties() subsetof
        {{(domainType, 'range'), (domainSize, 1)});
    check pvx->properties() subsetof
        {{(domainType, 'range'), (domainSize, 2)});
    check pvy->properties() subsetof
        {{(domainType, 'range')}};

    check scA->param(1) accepts (pvx +=
        {{(domainType, 'range'), (domainSize, 2), (length, 2)}});
    check scA->param(2) accepts (pvy +=
        {{(domainType, 'range'), (domainSize, 1)}});
    check scB->param(1) accepts (pvz +=
        {{(domainType, 'range'), (domainSize, 1)}});
    check scB->param(2) accepts (pvw +=
        {{(domainType, 'range'), (domainSize, 1)}});
}
```

Listing 4: A Possible Solution Architecture

C. Analysers

The work on the analyser tool is ongoing. In essence, this tool will generate a list of candidate solver architectures and select the best one using artificial intelligence techniques.

Listing 4 shows a possible solver architecture produced by the analyser for the problem from Listing 1, including component instantiations and configurations using `link` statements. The relevant template specifications are not shown due to lack of space.

```
Listing 3: Specification of the Problem Component
```

D. Solver Generation

The architecture chosen by the analyser is given to the solver generator to create the target solver. This tool uses the location and file name information attached to each element in the architecture graph to find the corresponding implementations in the C++ component library. The main tasks of the solver generator are to:

- create a file for the main solver program,
- include the component files required by the chosen architecture,
- instantiate the included components and parameterise them as appropriate, and
- generate code to set-up and begin execution of the solver.

The solver generator performs a fairly straightforward translation based on the decisions made earlier in the process.

VII. CONCLUSIONS AND FUTURE WORK

We have provided an outline of a novel, architecture-driven approach to generating constraint solvers that are optimised for a given problem. The generative approach demands an expressive ADL, such as Grasp, able to capture different types of compatibility requirements among architectural elements.

In addition to completing work on the remaining tools and evaluating the generative framework against existing solvers, numerous interesting avenues remain for further work. These include extending the role of software architectures in assisting the tasks of analysis and execution monitoring. In particular, we aim to explore the use of variation points in the architecture.

ACKNOWLEDGMENTS

This work is supported by the EPSRC grant “A Constraint Solver Synthesiser” (EP/H004092/1) and SICSA studentships.

REFERENCES