An Automated Approach to Generating Efficient Constraint Solvers

Dharini Balasubramaniam, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Peter Nightingale
School of Computer Science,
University of St Andrews,
St Andrews, UK
{*dharini,caj21,lk223,ijm,pwn1*}*@st-andrews.ac.uk*

***Abstract*—Combinatorial problems appear in numerous settings, from timetabling to industrial design. Constraint solving aims to find solutions to such problems efficiently and automatically. Current constraint solvers are monolithic in design, accepting a broad range of problems. The cost of this convenience is a complex architecture, inhibiting efficiency, extensibility and scalability. Solver components are also tightly coupled with complex restrictions on their configuration, making automated generation of solvers difficult.**

We describe a novel, automated, model-driven approach to generating efficient solvers tailored to individual problems and present some results from applying the approach. The main contribution of this work is a solver generation framework called Dominion, which analyses a problem and, based on its characteristics, generates a solver using components chosen from a library. The key benefit of this approach is the ability to solve larger and more difficult problems as a result of applying finer-grained optimisations and using specialised techniques as required.

***Keywords*-Generative programming; constraint solvers; software architecture; model-driven development**

I. Introduction

Combinatorial problems appear in a wide variety of settings that impact all of our lives – from institution timetabling and factory scheduling to industrial and experimental design, configuration and combinatorial mathematics. Constraint solving offers a means by which solutions to such problems can be found efficiently and automatically.

There are two phases to solving combinatorial problems using constraint technology. In the first phase, 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. In the case of timetabling, one might have two decision variables per lecture, one representing its time and the other its venue. The second phase uses a constraint solver to find solutions to the model: assignments of values to decision variables satisfying all constraints (e.g. a valid timetable). Constraint solvers typically employ a systematic backtracking search through the space of partial assignments in order to find solutions.

A major challenge facing constraints research is to deliver constraint solving that scales easily to problems of practical size. Current constraint solvers, such as Choco [1], Eclipse [2], Gecode [3], Ilog Solver [4], or Minion [5] are monolithic in design, accepting a broad range of models. This convenience comes at the price of a necessarily complex internal architecture, resulting in significant overheads and inhibiting efficiency and scalability. Each solver may thus incorporate a large number of features, many of which will not be required for most constraint problems. The complexity of current solvers also means that it is often prohibitively difficult to incorporate new techniques as they appear in the literature. A further drawback is that current solvers perform little or no analysis of an input model and 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 constraints as a technique for solving the most challenging combinatorial problems. The components of a constraint solver are also usually tightly coupled, with complex restrictions on how they may be linked together, making automated generation of different solvers difficult.

In this paper, we describe the development and evaluation of a novel and automated approach to improving the scalability of constraint technology, while simultaneously removing its reliance on manual tuning by an expert: a constraint solver generator framework called *Dominion* that, for a given problem, produces a solver tailored to that problem. There are two key benefits in generating a solver from scratch:

1) it will enable fine-grained optimisations that are not possible for a general solver, allowing the solution of much larger, more difficult problems, and
2) it will enable 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.

These benefits lead to faster and more scalable solvers. In addition, the automated approach simplifies the task of

modelling constraint problems by removing the need to manually optimise specifications.

A few current solvers, such as the Minion solver mentioned earlier, allow some specialisation. There are 7 variable types in Minion and each constraint is compiled for two parameters, each of which can be a different variable type. Each constraint is therefore compiled 49 times. Adding one extra option to variables (doubling the number of variable types) increases compilation time fourfold. In contrast, Dominion compiles only those variables and constraints that are required for each problem.

In order to manage the complex dependencies on the compatibility and configuration of components that constitute a solver, we adopt a model-driven approach. The generation process in Dominion is driven by the software architecture of the target solver. Software architecture provides a high-level model of a system in terms of its constituent elements and their interactions as well as properties that have to hold among the elements [6], [7]. The specified properties can be checked for consistency at the architectural level as well as against an implementation. Thus software architecture forms a useful basis for the design, implementation and evolution of systems. A brief outline of the implications of the Dominion approach, in particular the complex dependencies of solver components, on the specification of software architecture of the solver is provided in [8].

As well as architecture-driven development, we utilise concepts from generative programming, AI, domain-specific software engineering and product-lines in the Dominion approach. To achieve the goal of generating an efficient constraint solver for a given problem, we begin by specifying an architectural element (or component) corresponding to that problem. An architecture of the solver is constructed from this seed as a directed acyclic graph using pre-defined components from a library that satisfy the properties required by existing nodes in the graph. Thus, each new node represents a potential variation point in the architecture. Each DAG thus produced will model a valid solver but not necessarily an efficient one. Established AI approaches are used to determine the best option for each variation point. The chosen instantiation forms the software architecture of the desired solver.

The paper is structured as follows. We use the N-Queens problem, specified in Section II, throughout the paper to illustrate the Dominion approach. The challenges of automatically generating constraint solvers are discussed in Section III. Section IV provides a brief overview of the Grasp ADL used to represent the solver architecture while Section V outlines the generative process. The unique features of Dominion, which would be difficult to implement in monolithic solvers, are discussed in Section VI. The current status of the framework and results from evaluating the performance of solvers generated by Dominion against an existing solver are given in Section VII. Section VIII describes other attempts to fully or partially generate problem-specific solvers and differentiates them from our work. We conclude the paper with some thoughts on further work in Section IX.

II. An Example – N-Queens

N-Queens is the problem of placing n queens on an $n \times n$ chessboard, such that no two queens attack each other. It will be used in the remainder of the paper to illustrate the Dominion approach to automating the generation of specialised constraint solvers. It is also used, along with other problems, to evaluate the performance of generated solvers against existing work in Section VII.

Listing 1 shows the N-Queens problem expressed in the Dominion Input Language (DIL) [9], which is a modelling language that can express any constraint problem. In this example, we observe that every row of the chessboard has exactly one queen on it, and we represent the position of that queen using a decision variable with domain $\{1 \dots n\}$. Constraints are then added to rule out any pair of queens attacking each other.

A constraint problem is expressed as a list of parameters (`given`), decision variables (`find`), their domains and finally a list of constraints and constraint comprehensions that any solution must satisfy (following `such that`). A constraint comprehension is a compact way of specifying a set of parameterised constraints for a range of parameter values.

A problem class (such as N-Queens) has a set of parameters, specifying dimensions or other aspects of the problem. In this case there is a single parameter n specifying the dimensions of the chessboard ($n \times n$) and the number of queens to be placed on it. Before solving, all parameters are substituted in to create a *problem instance*.

N-Queens has a one-dimensional matrix of decision variables named `queens`. The matrix is first dimensioned (with the `dim` keyword). The explicit dimensioning is necessary to allow for cases in which only some of the variables in the matrix are used. This process can be compared to lazily allocating memory in a programming language — if the memory is not used, it is not actually allocated. Then a `find` statement is used to insert decision variables into the matrix. In this case all positions in the matrix have a decision variable with domain $1..n$.

Following the key words `such that`, each constraint or comprehension is given a unique name in the DIL (in this case, `alldifferent`, `diagonals1` and `diagonals2`). Constraints are specified as `con(arg1, arg2, ...)` where `con` is one of a fixed, but extensible set of constraint identifiers. Arguments may be matrices, single decision variables or constants. Finally, constraints (like many other structures in DIL) may be placed in a comprehension, as in `diagonals1` and `diagonals2`.

The first constraint in Listing 1 ensures that no two queens are in the same column. It uses the `alldiff` constraint,

```
language Dominion 0.1
given n: int {3..}
dim queens[n]: int
find queens[..]: int {1..n}

such that

alldifferent alldiff(queens[..])
[ diagonals1 sumneq([queens[j], j-i], queens[i]) | i in {0..n-2}, j in {i+1..n-1} ]
[ diagonals2 sumneq([queens[j], i-j], queens[i]) | i in {0..n-2}, j in {i+1..n-1} ]
```

Listing 1. The N-Queens constraint problem

which ensures all variables in an array take different values. The two constraint comprehensions ensure queens cannot attack each other along the diagonals, by ensuring the difference between the queens in columns i and j is neither $i-j$ or $j-i$. A DIL specification may contain references to a subset of the 28 constraints supported by the language.

III. Challenges for the Automatic Generation of Constraint Solvers

A simple constraint solver is not a fundamentally complex piece of software. The necessary components of a solver are: a representation of variables; a representation of constraints; a search engine that decides heuristically what decisions to make; a propagation engine that allows constraints to act on the consequences of those decisions; and a state maintenance facility that allows changes as a result of search and propagation, and reverses those changes on backtracking. Each of these components *can* be implemented simply: state maintenance, for example, can be as simple as copying all data structures before changes are made and then copying them back into place on backtracking. Certain propagation engines may be algorithmically complex to obtain optimal performance, but this need not be a problem for the construction of solvers from components. Given the apparent overall simplicity, we explain the major challenges in automatically constructing an efficient constraint solver in this section.

A vital input to the generation process is a specification of a solver in terms of its constituent elements including their structure, interactions and dependencies. This specification should provide adequate information for tools to make decisions on correctness, compatibility as well as efficiency. Software architectures are designed to capture this information at an abstract level and hence provide a reasonable basis for driving the process of solver generation.

However, a cluster of related issues contribute to the architectural complexity of a fast constraint solver, all driven by the need to optimise code for speed. First, there is tight linkage between the implementation method for each component and the services it then requires from other components. Second, a constraint problem can require thousands of variables and many constraints per variable. There may be different optimal choices for different variable or constraint components: thus we can either make a compromise choice leading to suboptimal performance, or allow for many different component types leading to greatly increased architectural complexity. Third, apart from the architectural issues in allowing multiple component types, the multiple choices tend to lead to "monolithic" solvers, in which a large number of choices are always available and crucially lead to inefficiencies. These inefficiencies can be as simple as much unnecessary code being in an executable, but more important ones arise where time and memory are used to maintain superfluous data structures which are needed only to support component choices not currently being used. The final issue is that the nature of constraint solving means that the difference between optimal and suboptimal performance can be critical. It is not unusual for the choice of search strategy to affect performance exponentially. Even in areas where no exponential speedup is available and an optimal algorithm is already being used, implementation efficiencies can make a difference in performance of thousands of times to do exactly the same work. Therefore we cannot assume that suboptimal performance is acceptable for the sake of architectural simplicity.

To illustrate the issues that arise, consider just one of the major components discussed earlier, the state maintenance facility. As mentioned, it can be implemented very simply, but this is unlikely to be optimal. In particular, naïvely copying blocks of memory does not scale well as problems become large. Therefore, it is natural to use 'trailing', where changes are pushed onto a stack, and their effects undone in reverse order when search backtracks. On the other hand, block-copying is very fast indeed on modern machines, so such a simple approach proved highly efficient when used on early versions of Minion [5]. Which of these simple choices is best varies from case to case. But there are other choices available. One is recomputation [10], in which states are only copied intermittently, with intermediate states recomputed each time they are revisited. Another family of techniques tries to avoid the need to restore state at all. A classic example would be the use of "watched literals", originating in satisfiability solving [11] and later for constraint solving [12]. With watched literals, while the data structure for a constraint changes during propagation, it is still valid for earlier states in the search tree. Therefore no state restoration

is necessary. Although it comes at the cost of losing a certain kind of optimal behaviour, the tradeoff is worthwhile in many cases.

The fact that there are many options for state maintenance is not the major issue. It is that, to achieve good performance, we might need to use a mixture of these techniques for different parts of the constraint solver. For example, certain kinds of constraint might work best with watched literals, while other constraints would work best with trailing. Constraint problems often have many different types of variable, constraint, and propagator, and so many different variants of memory management might be appropriate within a solver for one constraint problem. Therefore major complications arise because of the interactions between different kinds of state maintenance. For example, the speed advantage of watched literals is negated if all memory associated with the constraint is copied at each node. With memory management being so fundamental, it is also easy to introduce bugs through incorrect interactions between types of memory management.

In summary, we cannot produce constraint solvers with a single simple architecture and retain acceptable performance. The range of architectures we require for generating constraint solvers has to cater for a number of alternative implementations of the key components of a solver, for different alternatives to be used within a single solver, and for the complex interactions these choices allow to be managed to produce a correct and efficient constraint solver. 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.

IV. Representing Solver Architectures

The software architecture of the solver drives the generation process in Dominion and hence the architecture specification must be sufficiently expressive to deal with the complex requirements and dependencies described in the previous section. In addition to customary details of components and connections, the Dominion approach requires further support from the architecture representation in order to automate the process of producing an optimal architecture from the problem specification, and the solver code from the architecture.

A number of architecture description languages (ADLs) have been defined over the years [13]. We use a general purpose, textual ADL called Grasp [14] to represent solver architectures. Grasp has been designed to capture the structure, behaviour and rationale of systems at the architectural level. It supports architectural primitives such as layers, components, connectors, templates, interfaces, rationale, links, properties and check clauses. Components and connectors are typically the fundamental elements of a software architecture and represent units of functionality and interaction. Layers are logical structures used to promote modularity and flexibility by enforcing separation of concerns. Each layer may only communicate with the layers immediately above and below it. Rationale captures the reasoning behind architectural design decisions.

Templates are abstractions for architectural elements and can be used to create instances of components and connectors with shared behaviour. The required and provided interface names allow tools to check the compatibility of linked elements and also aid the generation of solver architectures from a problem component by matching required functionality.

Properties are characteristics (or restrictions of functionality) that are associated with architectural elements in the form of name-value pairs. Check clauses allow properties of parameters and linked elements to be checked for compatibility. In addition, Grasp provides a generic annotation mechanism to associate meta-data with architectural elements. This facility may be used for documentation purposes or to supply information to tools. Dominion uses this mechanism to specify the locations and file names of corresponding implementations for architectural elements to aid automatic generation. A number of standard functions are also supported by Grasp so that tools can query the state of the architecture, such as child elements of a composite component or support for a certain interface by a component.

Listing 2 shows a sample template specification in Grasp:

```
@Dominion(Classname = "BoolVarFact")
@Dominion(Filename = "boolvar.hpp")
template BoolVariableFactory() {
  provides IPropVariable;
  provides IremoveFromDomain;
  provides ItriggerOnDomain;
  requires ITriggerContainerFactory tcf;
  requires IMemoryManager mm;
  check mm.getProperties() subsetof
    [(MemoryChanges, 'Single')];
  property domainIs = "boolean";
}
```

Listing 2. A Template in Grasp

This template is for a factory that creates Boolean decision variables, which implement the interface `IPropVariable`. This template also implements the interfaces `IremoveFromDomain` and `ItriggerOnDomain`. These interfaces represent extensions to `IPropVariable`, providing extra functionality. The template requires to be connected to a component that implements the interface `IMemoryManager`. The `check` statement states that the memory manager is only allowed to have the property (MemoryChanges, 'Single'). The `domainIs` property ensures the domain is boolean, that is the domain $\{0, 1\}$. Other templates that implement `IPropVariable` impose other restrictions, such as that the domain must be an uninterrupted range, or must contain only 2 values. Being able to provide both extensions (with `provides`) and restrictions (with `property`) to a basic interface is vital to compactly representing the components of constraint solvers.

The genericity and expressive power of Grasp make it an ideal notation to capture the complexity of constraint solvers and to perform consistency checks before solvers are generated. The Grasp toolset currently consists of a compiler and a checker. Further tools for visualising, designing and validating architectures as well as performing traceability analysis are planned for the language.

V. The Automatic Generation of Customised Solvers

As described in Section III, producing an efficient and lean solver for a given constraint problem requires that only those components that are required to solve the problem are incorporated in the solver and that the most optimal combination of component implementations is used. The automated generation process is driven by software architecture of the target solver and contains the following steps as illustrated in Figure 1:

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

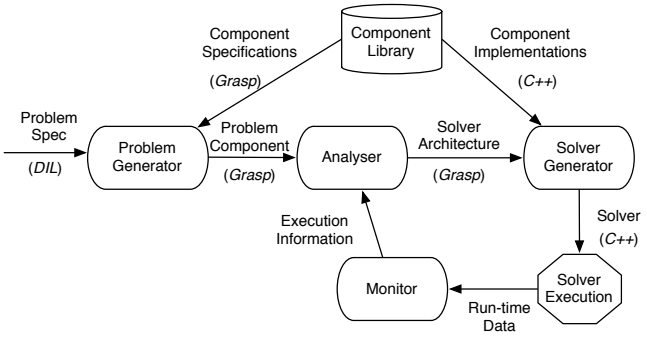

Figure 1. The Generative Process for Creating Solvers

The overall process may be considered as a control loop with the problem specification initiating feedforward control and data on the execution of the solver resulting in feedback control. C++ is used as the implementation language for performance, modularity and backwards compatibility reasons. The solver generation adopts component-based software engineering practices as well as an architecture-driven methodology. A component library is used throughout the solver generation process and is introduced first. Interesting aspects of the generation process are discussed in the subsequent subsections.

The current solver generator implements feedforward control, with a new problem initiating solver generation. Some feedback control using execution monitoring and analysis has been implemented as part of the tuning process in the analyser.

A. Component Library

Components form the building blocks of Dominion constraint solvers. A large number (currently 275) of reusable components are maintained in a library by the Dominion framework to be used to assemble the specialised solver. There are two aspects to the Dominion component library: the specification of components in Grasp 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 templated and parameterised to allow customisation as required.

A binding from Grasp to C++, which maps each Grasp template to a C++ class, is used for all components. For efficiency reasons, the mapping to C++ is performed statically with templates, with the type of each **requires** clause transformed into a template argument for the class, and then the object itself passed into the object's constructor. Listing 3 shows an example class which implements the Grasp template from Listing 2.

```
template<typename TrigConFactory,
         typename MemoryManager>
struct BoolVarFact
{
  BoolVarFact(TrigConFactory* tcf,
              MemoryManager* mm)
  { ... }
};
```

Listing 3. A C++ binding for the Grasp in Listing 2

The exact C++ methods that should be provided for classes providing a particular interface are described using C++ templates. Each class is stored in a separate file, promoting flexibility of use and modularity. This decision also enables the generation of the final solver to be automated in a straightforward manner, making use of the class name and file name annotations of architectural elements.

B. Problem Component Generation

The generative process is initiated by a constraint problem, represented in the Dominion Input Language, being supplied to the problem component generator tool. This tool parses the DIL specification and generates a new component, providing both C++ and Grasp versions, which captures the essence of the problem such as the interfaces that should be supported by variable and constraint components required to solve the problem, and any restrictions, such as domain compatibility, among them. Each problem component is unique to the constraint problem to be solved and thus such components are not stored in the component library for reuse. While this component is generated specially for every class of Dominion problems, once generated it is not treated differently from any other Grasp component.

For the N-Queens problem from Listing 1, the problem component generator will produce the Grasp specification shown in Listing 4.

This problem component is linked to appropriate variable and constraint components that support the specified interfaces and satisfy the given conditions. The check clauses of a component can be divided into two parts. The first set places restrictions on the variables based on the domains they will have. In this case $queens$ must have a bound domain. Restrictions can be specified on the type and size of required domain. Thus the choice of component for the variables is restricted 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 creates a copy of a component with added properties. In this case the `(DomainType, 'bound')` property is added. We know in this particular problem that the domain of each variable will be an unbroken range, which we denote with domain type `bound`.

Regardless of the chosen implementation for `queens`, which may or may not use this restriction, any variable produced by `queens` will have an unbroken range as its domain. The key point is that this allows implementations for the constraints that require the `bound` property even when the implementation for `queens` may not provide `bound` in general.

C. Determining Valid Solvers

The process of generating valid solvers is complex and difficult because of the restrictions on combining and connecting components. Composing the components to form a valid solver is a classic configuration problem. There are several ways of solving configuration problems, one of which is to encode it as a constraint satisfaction problem (see for example [15]).

This approach usually requires extensions of the standard constraint paradigm. We adopt a slightly different approach that requires no such extensions to use the Minion constraint solver unmodified to find a valid solver configuration. A detailed description of the process that translates a Grasp problem specification and components database into a constraint problem is beyond the scope of this paper; the interested reader is referred to [16].

The result of this step is a valid, but not necessarily good Dominion solver. One of the advantages of modelling the configuration problem as a constraint problem and solving it using Minion is that we have several means of guiding the configuration process towards a Dominion solver that we expect to perform well. Furthermore, we can generate several valid solvers instead of just one and compare them. Note that we are using Minion only as a tool here – we do not require constraint technology to find valid Dominion solvers, but we chose this way because it offers good performance and flexibility.

D. Analyser

The work on the analyser tool is ongoing. It generates a list of candidate solver architectures using the component library with the problem component as the seed and selects the best one using artificial intelligence techniques.

Each possible solver architecture produced by the analyser will include component instantiations and configurations (`link` statements). Each component instantiation makes use of a component template from the library, with a format similar to that shown in Listing 2.

The reader is referred to existing literature on algorithm selection [17], [18] and algorithm portfolios [19] for more details on possible techniques for use by the analyser.

In practice, the variable and constraint components required by the problem component themselves will require other interfaces, which leads to further components being instantiated. Interface names are sufficient to match components at the architectural level. Given either a partial or complete solver, we can execute the `check` statements for each component to ensure that dependencies hold and the solver is valid.

The architecture generated by the analyser is run through the Grasp checker tool, which looks for potential inconsistencies in the specification such as template and component redefinitions, attempts to instantiate components from undefined templates and incompatibilities in the source (provides) and target (required) interfaces in configurations. It also determines whether check clauses, such as properties expected of connected components, hold.

E. Solver Generation

Finally, the architecture chosen by the analyser as optimal for the input problem is passed to the solver generator to create the target solver. This tool traverses the architecture graph, using the location and file name information attached to each element to find the corresponding component implementations in the component library. The main tasks of the solver generator are to:

- include the component files required by the chosen architecture
- instantiate the included components and parameters as appropriate, and
- generate code to read run-time parameters, set-up and begin the execution of the solver, which is denoted by a component called `main`.

The translation from the Grasp architecture to a C++ implementation is straight-forward, given the component-based design of the system, and the decisions made and information recorded by other tools earlier in the process.

VI. Dominion Features

In this section we overview the current features of Dominion which would be difficult, or impossible, to implement using a monolithic constraint solver architecture.

```
@Dominion(Filename="../../models/queens.dominion.hpp")
@Dominion(Classname = "DominionProblemClassFactory")
template DominionProblem() {
  provides IProblemClassFactory;
  requires IConstraintStoreFactory csf;
  requires IPropagatorFactory_alldiff alldifferent;
  requires IPropagatorFactory_sumneq diagonals1;
  requires IPropagatorFactory_sumneq diagonals2;
  requires IDiscreteVarFactory queens;
  check queens.getProperties() subsetof [(DomainType, 'bound')];
  check alldifferent.param(1) accepts ( queens + [(DomainType, 'bound')]);
  check diagonals1.param(1) accepts ( queens + [(DomainType, 'bound')]);
  check diagonals1.param(2) accepts ( queens + [(DomainType, 'bound')]);
  check diagonals2.param(1) accepts ( queens + [(DomainType, 'bound')]);
  check diagonals2.param(2) accepts ( queens + [(DomainType, 'bound')]);
}
```

Listing 4. Specification of the Problem Component

A. Mappers

Mappers [20] are a method of performing simple transformations on variables, such as negation or addition of a constant, for very low cost. These are used in Minion and Gecode internally, for example to implement the Max constraint by taking the Min constraint and negating all the variables. Schulte et al. [20] show this is as efficient as implementing a Max constraint directly. While mappers are a very useful feature, they are not traditionally provided to users because they cause an explosion in the number of types of variables, and implementing them with a generic interface is inefficient. As Dominion compiles each solver individually, we do not have to compile all possible combinations of mapper and variable types, but only those which are used in a particular problem class. Therefore we provide mappers in the Dominion Input Language.

B. Variable Implementations

Monolithic constraint solvers typically provide a very limited set of variable types, or just one variable type. To support many types, one must either sacrifice efficiency by accessing variables through an interface with virtual functions, or compile constraint propagators many times (for each combination of types for its arguments). Minion uses the second approach, but the number of variable types is limited to 7, and constraints with more than two arguments are not compiled for all possible combinations of types. Most constraints are compiled 49 times (7 types for two arguments). Gecode provides only two variable types. Dominion removes the limitations on the number and diversity of variable types by compiling propagators as needed.

For example, Minion provides a variable type named `bound` that stores only the upper and lower bounds of the domain. `bound` variables do not allow triggers on individual domain elements (interface `ItriggerOnDomain`). This is again an efficiency issue: without `ItriggerOnDomain` updating a bound is an $O(1)$ operation. Minion also provides a variable type (`discrete`) that stores the whole domain explicitly in $O(d)$ space, and supports `ItriggerOnDomain`. In some cases we would want some features of `bound` and others of `discrete` but we cannot have every possible variable type.

In particular, in Minion all variable implementations which support `ItriggerOnDomain` also take $O(d)$ space to store a domain. The current best propagator for disjunctions of constraints (by Jefferson et al [21]) requires `ItriggerOnDomain`. We cannot use this in Minion on variables with very large domains. In Dominion we can construct the variable implementation with exactly the set of features we require.

C. Heuristics

Heuristics for variable and value ordering can be used to improve the efficiency of constraint solvers. For example, the weighted degree variable ordering (WDEG) [22] has been shown to be a good heuristic for a wide range of problems. However not all heuristics are suitable for all problems and there may be costs associated with using heuristics. WDEG requires an extra data structure in every variable and constraint. WDEG is not present in Minion because it slows down the solver significantly even when it is not in use. However in Dominion it is possible to add these data structures only when required.

VII. Experiments

In this section, we compare the performance of Minion with solvers generated by the Dominion framework. The solvers are compared by program size and execution time. Experiments were run on an 8-core Intel Xeon E5430 server with a clock speed of 2.66GHz.

A. Local Search Analyser

Section V-D described the role of the analyser. In this section we describe an analyser based on local search, that is used for the experiments presented below.

The approach taken is hill climbing in the space of candidate solver specifications. Given a current state A, we search the neighbourhood of A in a random order. As soon as a state B where B is better than A is found, the hill climber moves to state B immediately.

Section V-C describes the process of creating a valid solver specification by solving a configuration problem using the Minion solver. The hill climbing algorithm sits above this, and adjusts the variable and value ordering used by Minion to obtain different valid solver specifications. By repeated changes to the variable and value ordering it is possible to reach every valid solver specification. Some changes to variable or value ordering will not affect the solver specification found by Minion.

For each candidate specification, a solver is generated, compiled and executed on a set of instances of the problem class and its run time is measured for each instance (with a time limit of 10 seconds). Solver A is considered better than solver B iff A solved more instances, or A and B solved the same number of instances and A used less time in total.

The hill climber begins at a random candidate specification and terminates when it is no longer possible to improve the specification. For efficiency a cache mapping solver specifications to run times was implemented to avoid generating and running the same solver twice. We executed the hill climber 10 times and took the best solver overall to compare to Minion.

B. Modelling Problems in Dominion and Minion

The feature sets of Dominion and Minion do not exactly match. To make the comparison we modelled each of the problem classes for both solvers as closely as possible, while still making best use of the constraints available in each solver. In two cases it is not possible for the models to exactly match. The `sum` constraint present in Dominion is translated to `sumleq` and `sumgeq` in Minion. The `sumneq(X,y)` constraint in Dominion is translated to one new decision variable `auxxy`, and the constraints `sumleq(X,auxxy)`, `sumgeq(X,auxxy)` and `diseq(auxxy, y)`. `sum` arises in every model except N-Queens. `sumneq` arises in N-Queens only.

C. Experimental Results

We used six problem classes, as follows.

- **N-Queens** The N-Queens problem, as described in Section II.
- **BIBD** The Balanced Incomplete Block Design problem (CSPLib problem 028 [23]).
- **Golomb** The problem of proving optimality of known optimal Golomb Rulers (i.e. the solver searches for any ruler that is shorter than the known one). Golomb Ruler is described on CSPLib, problem 006 [23].
- **Graceful** The problem of finding graceful labellings of graphs [24].

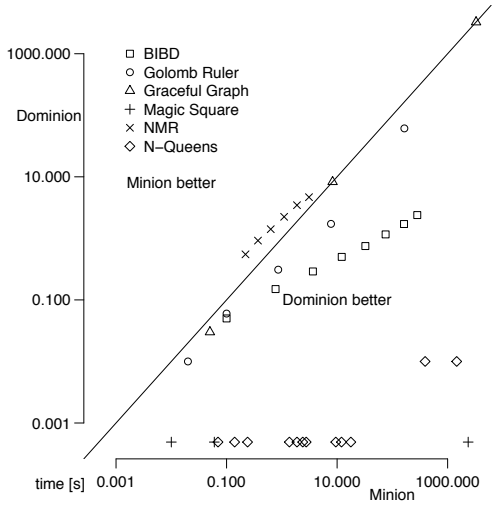

Figure 2. Experimental results comparing Dominion to Minion. The reported times are the median of three runs.

- **NMR** The problem of finding non-monochromatic rectangles [25].
- **MSquare** The problem of finding magic squares, CSPLib problem 019 [23].

The model and parameter files we used for these problems and the corresponding solver architectures are available on the Web.[1] With the exception of Graceful, we search for only one solution. For Graceful, we searched for all solutions.

For each problem class, we took a selection of parameter settings and generated a Dominion solver using the analyser described above. We then benchmarked the Dominion solver against Minion. We used only instances where both solvers could be run to completion within 1 hour.

Figure 2 plots the time taken (in seconds) by Dominion against Minion on a log scale for both axes. Overall, Dominion shows promising speed improvements. We make comments about particular problem classes below.

Figure 3 plots memory use (in MiB) of Dominion against Minion, again using log scales. We measured the peak resident set size, which is the amount of physical memory actually used by the program, including both data and the program binary. Minion never used less than 35 MiB, and on all problem instances Dominion used less memory than Minion.

For each problem class, we ran the hill climbing analyser 10 times. For N-Queens, NMR and Graceful, each run produced a solver that can solve at least one instance. For Golomb and MSquare, 9 of the 10 runs produced a solver that can solve at least one instance. However for BIBD only 4 of the 10 runs produced a solver that can solve at least one instance.

[1] http://www.cs.st-andrews.ac.uk/~pn/dominion-icse-problems.tgz

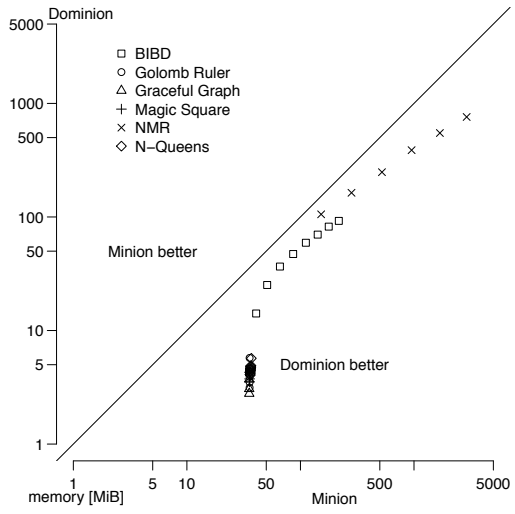

Figure 3. Experimental results comparing Dominion to Minion. The reported memory numbers are the median of three runs.

1) N-Queens and MSquare: For both these problems, except for the two largest instances of N-Queens, the times measured for the Dominion solver were zero. To plot these on the log-scale graph, we used 4.9 ms (the timer resolution is 10 ms therefore 4.9 ms would round down to 0). The largest instance of N-Queens was solved by Dominion in 0.01 s and by Minion in 1459 s.

For both problem classes, Minion used between 35.6 MiB and 37.0 MiB memory. Dominion however used between 3.5 MiB and 5.7 MiB memory.

2) BIBD: Dominion is consistently faster than Minion on BIBDs, with greater performance gains as problem size is increased. Dominion is over 100 times faster than Minion on largest BIBD (7-3-80). Dominion also improves upon Minion in memory use.

3) Golomb: For the largest instance of Golomb, Dominion was slightly more than 2.5 times faster than Minion, and Minion used over 6 times the memory. Other instances are similar.

4) Graceful: The Graceful instances showed almost identical time performance, however Minion used more than 9 times the memory that Dominion used.

5) NMR: NMR is the only class where the Dominion solver is slower than Minion. On the largest instance, Minion is 1.51 times faster. Dominion is substantially more efficient in terms of memory however, taking 760 MiB whereas Minion takes 2909 MiB. The Dominion solver consistently uses less memory on instances of NMR.

VIII. Related Work

One of the earliest examples of a system that attempts to generate constraint solvers tailored to a specific problem is MULTI-TAC [26], 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 [27] 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 general and, crucially, relies on almost no background knowledge. Westfold and Smith [28] are closer to our approach and use KIDS to synthesise efficient constraint solvers. They rely on reformulation and specialisation of the constraints however and do not consider the other components of a solver. Srivastava and Kambhampati [29] use KIDS to synthesise planners, but rely on explicit domain knowledge to do so.

The EasySyn++ system [30] automatically generates stochastic local search algorithms from a number of templated components. Again the synthesis is limited to a number of key components and does not encompass all aspects of the solver. Aeon [31] is a similar system for the automated generation of scheduling algorithms.

RT-Syn [32] uses simulated annealing to select the best ones from a set of abstract data structures and algorithms and synthesises a programme from the selected abstract descriptions. First, all algorithms and data structures that meet the requirements specified by the problem to solve are chosen. Then RT-Syn analyses all candidates and greedily selects the best one. The analysis is purely based on the abstract representation. Our approach relies on empirical evidence instead of belief about the actual performance and takes a more systematic approach to finding the best implementation.

Cahill [33] builds a knowledge base to aid with the construction of numerical algorithms from subcomponents. He models dependencies between components, but relies (at least partially) on knowledge input manually by human experts and does not report any results demonstrating the effectiveness of the system.

Brewer [34] builds statistical models to select data layout and sorting algorithm for iterative partial differential equation solvers. He also automatically tunes the parameters of the selected algorithms. Our approach follows the same general idea, but is not restricted to a small number of decisions and takes dependencies of components into account. We select implementations for *every* component that we have a choice for and model the ramifications of that choice on the rest of the software.

More recently, research has focused on tuning the parameters of existing solvers or selecting from among different solvers in algorithm portfolios. An algorithm portfolio contains algorithms that complement each other and the task is to select the most suitable one for solving the problem at hand [19]. One of the most prominent approaches is the SATzilla system [35]. SATenstein [36] is a successful system

that automatically tunes parameters and generates solvers based on the results of this tuning process. A system that combines both approaches and generates a portfolio of tuned algorithms is Hydra [37].

Such systems shift the focus from engineering efficient solvers to tuning and selecting from among existing ones. Software engineering techniques thus become less important than machine learning techniques. Our approach is more general. We retain an element of machine learning by selecting the most efficient solver architecture, but in the context of the automated generation of the solver from scratch.

There are also systems that make use of components to build efficient solvers, but do not accomplish this in an automated fashion. Van Hentenryck and Michel [38] describe how to generate efficient implementations from high-level descriptions of local search procedures. They focus on the high-level implementation choices and abstract from low-level details. Schulte and Tack [39] describe how to generate automatically variations of specific solver components.

To the best of our knowledge no previous approach to formally specifying the architecture of specialised constraint solvers exists.

IX. Conclusions and Future Work

We have provided an outline of a novel, automated, architecture-driven approach to generating constraint solvers that are optimised for a given problem. In addition to constraint programming, techniques from a variety of disciplines such as software architecture, learning, component-based software engineering and domain-specific development are combined in the Dominion framework to produce optimal solvers. The generative approach is supported by Grasp, an expressive ADL able to capture different types of compatibility requirements among the components of a solver, analysis techniques for choosing optimal components and a modular implementation of these components.

In addition to efficiency, the Dominion framework also improves usability compared to monolithic solvers. A DIL programmer will require the same knowledge of constraints needed to program in any other constraint modelling language but the task is simpler in Dominion since the programmer does not need to manually optimise the model.

Initial results from comparing solvers generated by Dominion with an existing solver are positive and indicate this approach is promising. Dominion is in fact expected to make bigger gains in the cases where there are many interdependent decisions to be made from a large number of components, where traditional solvers are limited by having to cater for the generic problem.

The Dominion approach improves performance and scalability of solving constraint problems as a result of:

- tuning the solver to characteristics of the problem
- making more informed choices by analysing the input model
- specialising the solver by only incorporating required components, and
- providing extra functionality that can be added easily and used when required.

A number of avenues are open for further work. An incremental approach will be adopted for refining the steps of the generation process, particularly the analyser tool. Further evaluation of the framework using constraint problems of different types and sizes continues. The evaluation activity will also continue to populate the component library.

The current Grasp checker performs some basic static analysis on architecture specifications for compatibility among linked elements. Work is ongoing on developing a semantic model that will allow more sophisticated and dynamic checks to be carried out. This work forms part of a larger research agenda for controlling architecture erosion by maintaining consistency and correctness between architecture and implementation in the face of system evolution.

As shown in Fig 1, a feedback loop to tune solvers based on data from their execution is to be incorporated into the generation process. A number of design decisions need to be made on capturing and representing adaptation policies and thresholds for acceptable behaviour for the solver, which may be unique to each problem. Given the architecture-driven nature of the generative process, we plan to explore the possibility of using Grasp to model this information.

Acknowledgment

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

References

[1] Choco. [Online]. Available: http://choco.emn.fr/

[2] Eclipse. [Online]. Available: http://www.eclipse-clp.org/

[3] Gecode. [Online]. Available: http://www.gecode.org/

[4] Ilog solver. [Online]. Available: http://www.ilog.com/products/cp/

[5] I. P. Gent, C. A. Jefferson, and I. Miguel, “MINION: A fast scalable constraint solver,” in *Proceedings of the Seventeenth European Conference on Artificial Intelligence*, 2006, pp. 98–102.

[6] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[7] M. Shaw and D. Garlan, *Software Architecture: Perspective of an Emerging Discipline*. Prentice Hall, 1996.

[8] D. Balasubramaniam, L. de Silva, C. Jefferson, L. Kotthoff, I. Miguel, and P. Nightingale, “Dominion: An architecture-driven approach to generating efficient constraint solvers,” in *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2011, (To appear).

[9] I. P. Gent, C. Jefferson, L. Kotthoff, I. Miguel, and P. Nightingale, "Specification of the dominion input language version 0.1," University of St Andrews, Tech. Rep., 2009. [Online]. Available: http://www-circa.mcs.st-and.ac.uk/Preprints/InLangSpec.pdf

[10] R. M. Reischuk, C. Schulte, P. J. Stuckey, and G. Tack, "Maintaining state in propagation solvers," in *CP*, 2009, pp. 692–706.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. 39th Design Automation Conference (DAC 2001)*, 2001.

[12] I. Gent, C. Jefferson, and I. Miguel, "Watched literals for constraint propagation in minion," in *Proc. CP 2006*, 2006.

[13] N. Medvidovic and R. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[14] D. Balasubramaniam and L. de Silva, "Grasp language reference manual version 1.0," University of St Andrews, Tech. Rep., 2011. [Online]. Available: http://www.cs.st-andrews.ac.uk/˜dharini/reports/GraspManual.pdf

[15] S. Mittal and B. Falkenhainer, "Dynamic constraint satisfaction problems," in *AAAI*, 1990, pp. 25–32.

[16] I. P. Gent, C. Jefferson, L. Kotthoff, and I. Miguel, "Modelling constraint solver architecture design as a constraint problem," in *Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, 2011.

[17] J. R. Rice, "The algorithm selection problem," *Advances in Computers*, vol. 15, pp. 65–118, 1976.

[18] E. Fink, "How to solve it automatically: Selection among Problem-Solving methods," in *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*. AAAI Press, 1998, pp. 128–136.

[19] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artif. Intell.*, vol. 126, no. 1-2, pp. 43–62, 2001.

[20] C. Schulte and G. Tack, "Perfect derived propagators," in *Fourteenth International Conference on Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, P. J. Stuckey, Ed., vol. 5202. Sydney, Australia: Springer-Verlag, Sep. 2008, pp. 571–575. [Online]. Available: http://www.ict.kth.se/ cschulte/-paper.php?id=SchulteTack:CP:2008

[21] C. Jefferson, N. Moore, P. Nightingale, and K. E. Petrie, "Implementing logical connectives in constraint programming," *Artificial Intelligence*, vol. 174, pp. 1407–1429, 2010.

[22] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*, 2004.

[23] B. Hnich, I. Miguel, I. P. Gent, and T. Walsh, "CSPLib: a problem library for constraints," http://csplib.org/.

[24] K. E. Petrie and B. M. Smith, "Symmetry breaking in graceful graphs," in *Proceedings of 9th International Conference on Principles and Practice of Constraint Programming (CP03)*, 2003, pp. 930–934.

[25] S. Fenner, W. Gasarch, C. Glover, and S. Purewal, "Rectangle free coloring of grids," arXiv, Tech. Rep. 1005.3750, 2010. [Online]. Available: http://arxiv.org/abs/1005.3750

[26] S. Minton, "Automatically configuring constraint satisfaction programs: A case study," *Constraints*, vol. 1, pp. 7–43, 1996.

[27] D. R. Smith, "KIDS - a Knowledge-Based software development system," in *Automating Software Design*. MIT Press, 1990, pp. 483–514.

[28] S. J. Westfold and D. R. Smith, "Synthesis of efficient constraint-satisfaction programs," *Knowl. Eng. Rev.*, vol. 16, no. 1, pp. 69–84, 2001.

[29] B. Srivastava and S. Kambhampati, "Synthesizing customized planners from specifications," *J. Artif. Int. Res.*, vol. 8, no. 1, pp. 93–128, Mar. 1998.

[30] L. D. Gaspero and A. Schaerf, "EasySyn++: a tool for automatic synthesis of stochastic local search algorithms," in *Proceedings of the 2007 international conference on Engineering stochastic local search algorithms: designing, implementing and analyzing effective heuristics*, 2007, pp. 177–181.

[31] J. N. Monette, Y. Deville, and P. van Hentenryck, "Aeon: Synthesizing scheduling algorithms from High-Level models," in *Operations Research and Cyber-Infrastructure*, J. W. Chinneck, Ed., 2009, pp. 43–+.

[32] T. E. Smith and D. E. Setliff, "Knowledge-based constraint-driven software synthesis," in *Knowledge-Based Software Engineering Conference, 1992., Proceedings of the Seventh*, Sep. 1992, pp. 18–27.

[33] E. Cahill, "Knowledge-based algorithm construction for real-world engineering PDEs," *Mathematics and Computers in Simulation*, vol. 36, no. 4-6, pp. 389–400, 1994.

[34] E. A. Brewer, "High-Level optimization via automated statistical modeling," in *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95, 1995, pp. 80–91.

[35] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: portfolio-based algorithm selection for SAT," *J. Artif. Intell. Res. (JAIR)*, vol. 32, pp. 565–606, 2008.

[36] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "SATenstein: automatically building local search SAT solvers from components," in *IJCAI'09*, 2009, pp. 517–524.

[37] L. Xu, H. H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for Portfolio-Based selection," in *AAAI-10*, 2010, pp. 210–216.

[38] P. V. Hentenryck and L. Michel, "Synthesis of constraint-based local search algorithms from high-level models," in *AAAI-07*. AAAI Press, 2007, pp. 273–278.

[39] C. Schulte and G. Tack, "Perfect derived propagators," in *CP'08*, 2008, pp. 571–575.