Refining Portfolios of Constraint Models with Conjure

Ozgur Akgun, Ian Miguel, and Chris Jefferson
{ozgur,ianm,caj}@cs.st-andrews.ac.uk

School of Computer Science, University of St Andrews, St Andrews, Scotland, UK.

Abstract. Modelling is one of the key challenges in Constraint Programming (CP). There are many ways in which to model a given problem. The model chosen has a substantial effect on the solving efficiency. It is difficult to know what the best model is. To overcome this problem we take a portfolio approach: Given a high level specification of a combinatorial problem, we employ non-deterministic rewrite techniques to obtain a portfolio of constraint models. The specification language (Essence) does not require humans to make modelling decisions; therefore it helps us remove the modelling bottleneck.

1 Introduction

Many interesting real life problems can be formalised as constraint satisfaction problems (CSPs). A CSP consists of decision variables with associated domains, constraints on the assignments of values to a subset of decision variables and optionally an objective function. Solving a CSP is a well studied practice. There are many existing solvers, which employ advanced algorithms to reason about given constraints and run efficient search algorithms.

In order to solve a problem using a CSP solver, one needs to *model* the problem at hand. CSP solvers have different input languages - a common input language provides boolean and integer variables, arithmetic, logical and global constraints on these variables. CSP solvers provide a relatively high level of abstraction and expressivity when modelling a problem compared to MIP and SAT, and they still provide fast and scalable black-box solvers.

Most real world problems contain complex combinatorial structures such as sets, multi-sets, functions, relations, tuples, etc. Modelling a problem that can naturally be specified using these high level constructs is not a straightforward task - there are many ways to model a certain combinatorial object and a relation between a number of combinatorial objects. Moreover these alternative ways of modelling the same abstract expression do not dominate each other in terms of efficiency. Thus, given an abstract problem specification, building an efficient CSP model requires a great deal of expertise in CSP technologies and many experiments.

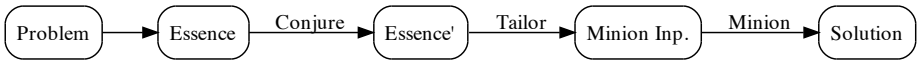
This work is an ongoing attempt to automate the CSP modelling process. In order to do so, we first design and implement a system to generate a selection

of valid CSP models given an abstract problem specification. The natural next step is to study relative strengths of generated models and design a system to select a good (if not the best) model. We will also explore the selection of a *portfolio* of models (as opposed to just selecting one) and running multi-model search techniques on this portfolio.

2 Tool chain

Our automated tool chain takes the approach of specifying the problem in an abstract constraint specification language, then compiling it to a low level model which current constraint solvers will accept. There are many decisions to be made at every step. These decisions have crucial impact on the actual time we spend on solving a given problem.

We employ different tools at different levels to best handle these tasks. Conjure takes Essence[1] specifications and generates a portfolio of Essence′ models. Tailor[2] takes Essence′ models as input and generates efficient Minion[3] input files. Conjure and Tailor have the capability to work at the problem class level, whereas Minion, the actual solver, works at the instance level.



3 Current status

The implementation of a working version of the refinement system Conjure is mostly complete. It is implemented as a non-deterministic term rewriting system. The current design and implementation of Conjure uses Haskell, provides an embedded implementation (EDSL) of the Essence language to be used by the rule authors, and an actual implementation of the language to be used by the problem owners. It successfully decouples the task of rewrite rule authoring from the implementation of the language and the actual process of applying the rewrite rules. The current rewrite rules database is a proof of concept demonstrating the fact that we can handle almost all of the language structures.

There exists a prototype implementation of Conjure, presented in [4], which refines a fragment of Essence limited to nested set-based decision variables into models in the Essence′ solver independent modelling language[1]

Current implementation successfully supports most of the Essence types, with minor limitations. For instance we currently do not handle function variables which map items from a nested combinatorial type to any other type. Function variables mapping integers to any type are fully supported though.

[1] Essence′ is, in turn, the input for the Tailor system [2], which transforms Essence′ models into input suitable for a particular constraint solver.

3.1 The Architecture of Conjure

This section gives an overview of the architecture of Conjure, which is a compiler-like system. Like most of the compilers, it has a pipeline which starts with parsing, validating the input, and type-checking. After these foundation phases, it has several phases for preparing the input specification for the rewriting phase, the actual rewriting phase, and some housekeeping phases. The pipeline is summarised below:

1. Parsing
2. Validating the input
3. Type checking the input
4. Representations phase
5. Auto-Channelling phase
6. Adding structural constraints
7. Expression rewriting
8. Fixing auxiliary and quantified variable names

Phases 1–3 are the foundation phases. The representations, auto-channelling, and adding structural constraint phases (4–6) prepare the input specification for the actual task of rewriting (Phase 7). Phase 8 can be viewed as housekeeping, it makes the output models easier to read and understand. Phase 7 (expression rewriting) is described in detail in the following sections. We will now give brief descriptions for the three preparatory phases preceding it.

Representation phase There are typically many ways to represent a combinatorial object. In this phase we make the representation decisions on the input specification in every possible way, and create multiple copies of it.

Auto-Channelling phase If we choose more than one way of representing a combinatorial object within a specification, we automatically add channelling constraints between different representations of the same variable at this phase. This way we link the different representations and make sure they represent the same combinatorial object.

Adding structural constraints At this phase we add all necessary structural constraints on every decision variable in the specification. The structural constraint for a representation of a decision variable makes sure the selected representation actually represents a valid combinatorial object with the intended properties. We add these constraints before rewriting take place, because they will be added regardless of the rest of the specification and they only depend on the representation of a combinatorial object.

3.2 Non-deterministic Rewriting

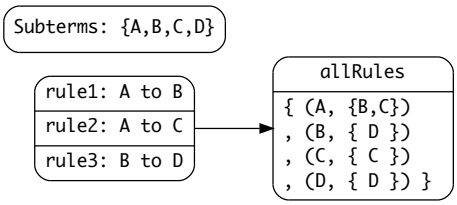
Our automated modelling system employs a term rewriting system to refine Essence specifications into the target language Essence′. Generally, rewrite rules can be thought of as partial functions, which map from a subterm to an *equivalent* subterm [5]. Given a set of rewrite rules and a term, a rewrite system

repeatedly applies the rules until no further rules can be applied. The term is then said to be in **normal form**.

As noted earlier, in order to produce alternative models we wish to generate not a single normal-form term, but all the normal-form terms that are attainable by applying the given rules to the input term. For this purpose we slightly adjust the definition of a rewrite rule: instead of a function that maps from a subterm to an equivalent subterm, we define a rewrite rule to be a function that maps from a subterm to a *set* of subterms.

Hence, a single rule in this definition is sufficient to represent the whole rule database. This representation is natural while applying the rules, but it is not a natural way to write them. It is, however, trivial to automate the combination of a set of partial functions into the single function used by the implementation.

For example we can combine `rule1`, `rule2` and `rule3` in `allRules` as follows:



Here `rule1`, `rule2` and `rule3` are partial functions. However the combined `allRules` is a total function, which maps from a subterm to a set of equivalent subterms.

`rule1` rewrites `A` into `B`, `rule2` rewrites `A` into `C` and `rule3` rewrites `B` into `D`. Since there is no rule matching `C` or `D` they are mapped to a singleton set of themselves.

In what follows, we will present our rules as partial mappings from single subterms to single subterms.

Figure 1 presents the elements of a rule: the mapping denoted by the $\rightsquigarrow$ operator; the guards that the left hand side of the mapping must satisfy; and the declarations to be used while constructing the right hand side of the mapping. Any expression that matches with the left hand side of the $\rightsquigarrow$ symbol is replaced by the right hand side, if all guards are satisfied.

Figure 2 shows an example rule that matches with a`subseteq` constraint between two sets of same types. It rewrites the constraint into a universal quantification over the first set. Can be read as *every element in set a, must also be an element of set b*. Notice also that it creates a quantified variable of type τ, which is the type of the elements of the two sets a and b. The actual name of the quantified variable is to be decided by the system.

```
essence_expression ⇝ equivalent_expression
    guards: properties that essence_expression must satisfy
    declarations: newly created variables and local aliases for expressions
```

Fig. 1. Anatomy of a refinement rule

```
a subseteq b ⇝ forall i : a . i elem b
    guards: a ∼ set of τ
            b ∼ set of τ
    declarations: i = quantifiedVar(τ)
```

Fig. 2. An example rewrite rule, `ruleSetSubsetEq`

It is useful to view our rules as operating upon an *Abstract Syntax Tree* (AST) representation of an ESSENCE specification. In the AST, every node represents a term in the specification and is also labelled with that term's type. To illustrate, Figure 3 presents a simple specification and its associated AST.

The root of the AST is the outer equality constraint, its immediate children are the decision variable x and the intersection operator, and so on. An identifier node, such as that associated with x, serves as a reference to the declaration of that identifier in the specification (the `find` statement in the case of x).

The rewriting system works by traversing the AST and attempting to apply the rules in the database at every node. A rule is allowed to modify the subtree rooted at the current node, and, for contextual information, is allowed to access (but not to modify) the remainder of the AST via the parent of the current node. If a rule matches the current node, the whole subtree is replaced with the equivalent subtree the rule suggests.

4 What's next?

Having a robust implementation of the automated refinement system, we are now one step closer to our ultimate goal, exploiting the opportunity of having multiple equivalent models for a given problem and eventually removing the *modelling bottleneck* from CSP to make it more accessible to wider audiences.

There is still a great necessity of improvements on the rules database. The quality and quantity of generated models directly depend on the quality and diversity of rules at hand.

```
given a,b : set of τ
find x    : set of τ

such that
    x = a intersect (b union c)
```

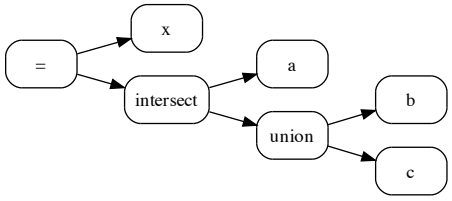


Fig. 3. A simple Essence specification and its AST view. Note that τ represents any concrete type.

Once we are confident about the models we generate, we will start studying effective model selection techniques. There are two basic stages where we can benefit from having multiple models, as described below.

Static exploitation of multiple models. Most constraint models describe a parameterised problem "class" (e.g. the class of sudoku puzzles). For input to a constraint solver, an instance of the class is obtained by giving values for the parameters (e.g. the pre-filled cells on the sudoku grid). We can exploit multiple models of a problem class by using small sized training instances to find the best-performing model, then we can use the best-performing model to solve other instances of the problem class. Although the search through the model space is initially uninformed, the system will learn which components of models tend to lead to better models and use this information to inform future model selection decisions.

Dynamic exploitation of multiple models. Constraint solvers typically employ a backtracking-style search combined with inference at each search node (constraint propagation) in order to find solutions. Since each search decision results in a new sub-problem that differs slightly from that associated with its parent node, our initial model selection might in fact be sub-optimal after a few decisions have been made. Hence, we can exploit multiple models dynamically by switching model mid-search. In order to do so, we must be confident that the new model will perform better than our current selection. The changing structure of the problem resulting from the decisions made by the constraint solver will provide the basis for this model selection, again employing a machine-learning methodology.

The result of these two approaches will be significantly enhanced performance of constraint solving, which will benefit a wide variety of industrial and academic users with combinatorial problems to solve. It will also remove the modelling bottleneck, in that it will no longer be necessary to have the expertise to select the "best" model.

References

1. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In Veloso, M.M., ed.: IJCAI. (2007) 80–87
2. Rendl, A.: Thesis: Effective compilation of constraint models. (2010)
3. Gent, I.P., Jefferson, C., Miguel, I. (In: ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings)
4. Frisch, A.M., Jefferson, C., Hernández, B.M., Miguel, I.: The rules of constraint modelling. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI, Professional Book Center (2005) 109–116
5. N. Dershowitz, J.-P. Jouannaud: Rewrite Systems. In: Handbook of Theoretical Computer Science. North-Holland (1990)