Natural Optimisation Modelling for Software Developers

Kathryn Glenn Francis
orcid.org/0000-0003-4217-6672

A thesis presented for the degree of

Doctor of Philosophy

December 2016
Department of Computing and Information Systems
The University of Melbourne

This thesis is submitted in total fulfilment of the degree.
Abstract

Constraint solving technology has been successfully applied to large industrial combinatorial optimisation problems with high impact, but is not widely applied to smaller problems. Unfortunately, including even basic optimisation or constraint satisfaction functionality in an application currently requires the use of an entirely separate paradigm with which most software developers are not familiar. The aim of this thesis is to demonstrate the potential for an interface to a constraint solver which is much more approachable for general software developers, in order to promote wider use of the technology.

Instead of defining a conventional constraint model directly, programmers use native code (in the application programming language) to define how to combine individual decisions to construct a candidate solution, and how to evaluate the result. Constraint satisfaction or optimisation is then seamlessly integrated into the wider application through automatic conversion between this definition and a conventional model solved by an external solver. We call this a native programming language interface.

This thesis presents a prototype implementation of a native Java interface to a finite domain constraint solver, before exploring several ideas for improving the automated conversion from procedural code into an equivalent constraint model. This conversion process has already been studied in the context of software engineering applications, and the improvements discussed here should be transferable back to that domain.

The following new techniques are presented.

- A novel query-based approach to handling destructive assignments, designed to improve the translation particularly when aliasing between variables is allowed.

- An alternative technique (loop untangling) for creating copies of loop bodies in such a way that the uncertainty within the loop body is reduced, at the expense of a greater number of copies and an unknown execution order.

- A new global constraint (reaching definitions) generalising the query-based translation technique to eliminate the assumption of a known order of
execution for assignments (for use with loop untangling), and to achieve stronger deduction in some cases.

To support these new techniques, two further contributions are included in the thesis.

- A study into the circuit constraint, exploring how this constraint can be extended for the subcircuit/subpath case as required to constrain the execution path when using loop untangling, and how both versions can be implemented in a lazy clause generation constraint solver.

- As part of the implementation of the reaching definitions constraint, a discussion of how optional variables (solver variables which are allowed to take no value) can be implemented in a lazy clause generation solver.
Declaration

I hereby declare the following.

1. This thesis comprises only my own original work towards the degree of Doctor of Philosophy.

2. Due acknowledgment has been made in the text of all other material used.

3. The thesis is fewer than 100,000 words, exclusive of tables, maps, bibliographies and appendices.

Kathryn Glenn Francis, December 2016
Preface

The research presented in this thesis was conducted in the Computing and Information Systems department at the University of Melbourne, in collaboration with my primary supervisor Peter Stuckey and a sequence of secondary supervisors Sebastian Brand, Jorge Navas, and Harald Søndergaard. The thesis includes material published in the following conference and journal papers.


I have been the grateful recipient of an Australian Postgraduate Award, and a NICTA (National Information and Communications Technology Australia) top-up scholarship.
Acknowledgments

Firstly I would like to thank my primary supervisor Peter Stuckey. I consider myself very lucky to have had a supervisor who is well known and respected in his field and yet still approachable and consistently available to his students. I very much appreciate Peter’s dedication to regular meetings, during which I have enjoyed many interesting discussions, and his willingness and ability to adjust his supervisory style for each student.

I would also like to acknowledge the contribution of my sequence of second supervisors Sebastian Brand, Jorge Navas, and Harald Søndergaard, and to thank all of the other people in the CIS department who have made my time at the university pleasant and enjoyable. I have enjoyed participating in various social events including pizza talks and board game lunches, and simply walking down the hall exchanging greetings and smiles. So thank you to everyone especially my fellow occupants of level 6.

I feel very grateful to have been the recipient of a NICTA top-up scholarship, primarily not for the stipend, but rather because it has given me the opportunity to be involved in such a vibrant, productive intellectual community. As a NICTA student I have been able to attend several conferences, summer schools, and NICTA retreats, all of which I have thoroughly enjoyed as well as benefiting academically.

I have also greatly enjoyed participating in the Constraint Programming community, and would like to thank senior members of that community, especially Christian Schulte and Pierre Flener, for making me feel welcome and valued from my very first encounter with them at the 2011 CP summer school in Turkey.

On a more personal note, I am extremely grateful for the support of my husband Richard, in his various roles including sounding-board, proof-reader, poster-formatter, and stress-reducer. Thank you also to the rest of my family, for moral support and extensive babysitting.

Finally, thank you to my daughter Sophia for giving me perspective and a well timed break, and my son Aurelius for prompting me to finally just finish.
To Sophia and Aurelius.
May you find your passion and follow it.
## Contents

1 Introduction and Preliminaries .............................................. 1
  1.1 Introduction ............................................................... 1
  1.2 Combinatorial Optimisation ......................................... 2
    1.2.1 Constraint Programming (CP) ................................. 3
    1.2.2 Boolean satisfiability (SAT) solvers ....................... 4
    1.2.3 Lazy Clause Generation (LCG) ............................... 4
    1.2.4 Satisfiability Modulo Theories (SMT) ...................... 6
  1.3 Java ............................................................................. 7
    1.3.1 Classes and interfaces .......................................... 7
    1.3.2 Variables .................................................................. 8
    1.3.3 Generics .................................................................. 9

2 Native Programming Language Access to CP ................................ 10
  2.1 Introduction ................................................................. 10
    2.1.1 Motivation .............................................................. 10
    2.1.2 Definition of a native interface ............................... 11
  2.2 Examples ....................................................................... 13
    2.2.1 Haskell ................................................................. 13
    2.2.2 Java ....................................................................... 14
    2.2.3 M-Queens .............................................................. 15
  2.3 Comparison with a Conventional CP Library ....................... 19
    2.3.1 Decisions ............................................................... 19
    2.3.2 Constraints ............................................................ 21
    2.3.3 Integration ............................................................. 25
    2.3.4 Summary of advantages ......................................... 26
  2.4 Related Work - Inspiration ............................................ 28
    2.4.1 Comparison with our proposal ................................. 29
    2.4.2 Extensions to CoJava ............................................. 30
  2.5 Other Related Work ....................................................... 32
    2.5.1 Special purpose constraint modelling languages ........ 32
    2.5.2 Re-purposing existing tools ................................... 34
    2.5.3 Learning ............................................................... 35
    2.5.4 Simulation-based optimisation ................................ 37
    2.5.5 Combining programming paradigms ......................... 38
7.6.4 Basic change propagation ................................. 157
7.6.5 The change and query propagators ..................... 159
7.6.6 The reachOrNotMatch propagator ..................... 164
7.6.7 Evaluation ............................................ 166

7.7 Explanations ............................................. 169
7.7.1 Explaining query propagation ......................... 169
7.7.2 Explaining reachOrNotMatch propagation .......... 170
7.7.3 Evaluation of explanations ......................... 173

7.8 Extension: Filtering Change Values ..................... 175
7.8.1 Using single queries ................................... 175
7.8.2 Using sets of queries ................................. 175
7.8.3 Experimental evaluation .............................. 178

7.9 Extension: Eliminating Edges ............................ 180
7.9.1 Explanation ............................................ 180
7.9.2 Evaluation ............................................ 181

7.10 Revisiting Loop Untangling ............................ 182
7.10.1 Graph compression .................................... 182
7.10.2 Experimental results ................................. 184

7.11 Conclusion ............................................. 185
7.11.1 Query-query reasoning ............................... 186
7.11.2 Equality reasoning .................................... 186
7.11.3 Dominance reasoning ................................. 186

8 Conclusion ................................................. 187
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Native Haskell interface to CP</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Native Java interface to CP</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Haskell program solving the M-Queens problem</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Java program solving the M-Queens problem</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Print method extracted from MQueens class in Figure 2.4</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>An example CoJava program</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>The compile time operation</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>Type conversion</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>The optimisation process, triggered on demand during execution</td>
<td>55</td>
</tr>
<tr>
<td>3.4</td>
<td>ProjectPlan getFinishDay method, called from within evaluate</td>
<td>56</td>
</tr>
<tr>
<td>3.5</td>
<td>MiniZinc model for project planning example</td>
<td>57</td>
</tr>
<tr>
<td>3.6</td>
<td>A simplified Day class</td>
<td>61</td>
</tr>
<tr>
<td>4.1</td>
<td>A Java simulation of a pizza ordering optimisation problem</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Flattened version of buildOrder method</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>Example: translation using SMT and obvious CP approach</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Java code defining a routing problem</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>Constraints generated using unwinding versus untangling</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>Java simulation of pizza ordering optimisation problem</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>Flattened loop</td>
<td>93</td>
</tr>
<tr>
<td>5.5</td>
<td>Example path control graphs</td>
<td>96</td>
</tr>
<tr>
<td>6.1</td>
<td>MiniZinc model for tour design problem</td>
<td>107</td>
</tr>
<tr>
<td>6.2</td>
<td>Solutions to variations of the circuit constraint</td>
<td>108</td>
</tr>
<tr>
<td>6.3</td>
<td>MiniZinc model for tour design problem using subcircuit</td>
<td>110</td>
</tr>
<tr>
<td>6.4</td>
<td>Example: depth first search exploring disjoint subtrees</td>
<td>118</td>
</tr>
<tr>
<td>6.5</td>
<td>Illustration of scc propagation rules</td>
<td>119</td>
</tr>
<tr>
<td>6.6</td>
<td>Pseudo-code for the scc propagation algorithm</td>
<td>120</td>
</tr>
<tr>
<td>7.1</td>
<td>Example: decomposition versus global translation</td>
<td>138</td>
</tr>
<tr>
<td>7.2</td>
<td>Example: ‘compressed’ graph with more edges than original</td>
<td>141</td>
</tr>
<tr>
<td>7.3</td>
<td>Algorithm: selecting retained nodes and representatives</td>
<td>143</td>
</tr>
<tr>
<td>7.4</td>
<td>Channeling constraints enforced by optional integer variables</td>
<td>149</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>7.5</td>
<td>Propagation function for the query propagator</td>
<td>161</td>
</tr>
<tr>
<td>7.6</td>
<td>Propagation function for the change propagator</td>
<td>163</td>
</tr>
<tr>
<td>7.7</td>
<td>Propagation function for the reachOrNotMatch propagator</td>
<td>166</td>
</tr>
<tr>
<td>7.8</td>
<td>Example: overlapping query sets for filtering change value</td>
<td>176</td>
</tr>
<tr>
<td>7.9</td>
<td>Example: compressing untangled graphs</td>
<td>182</td>
</tr>
<tr>
<td>7.10</td>
<td>Example: simplifying compressed untangled graphs</td>
<td>183</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Performance compared with hand-written models</td>
<td>67</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparing approaches to modelling destructive assignments</td>
<td>82</td>
</tr>
<tr>
<td>4.2</td>
<td>Further comparison for benchmarks with variable collections</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparative performance of unwinding and untangling</td>
<td>103</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparing explanation clauses for <code>check</code></td>
<td>112</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparing heuristics for failure explanation for <code>check</code></td>
<td>114</td>
</tr>
<tr>
<td>6.3</td>
<td>Comparing explanation clauses for <code>prevent</code></td>
<td>116</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparing evidence literal selection heuristics</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparing root selection strategies for <code>scc</code> algorithm</td>
<td>126</td>
</tr>
<tr>
<td>6.6</td>
<td>Evaluating additional propagation rules for <code>scc</code> algorithm</td>
<td>127</td>
</tr>
<tr>
<td>6.7</td>
<td>Comparing evidence literal selection heuristics for <code>scc</code> algorithm</td>
<td>128</td>
</tr>
<tr>
<td>6.8</td>
<td>Comparing propagation algorithms without explanation</td>
<td>129</td>
</tr>
<tr>
<td>6.9</td>
<td>Comparing propagation algorithms with explanation</td>
<td>130</td>
</tr>
<tr>
<td>6.10</td>
<td>Evaluating the effect of explanation</td>
<td>131</td>
</tr>
<tr>
<td>7.1</td>
<td>Evaluating initial version of global without explanation</td>
<td>168</td>
</tr>
<tr>
<td>7.2</td>
<td>Evaluating initial version of global with explanation</td>
<td>174</td>
</tr>
<tr>
<td>7.3</td>
<td>Evaluating change value filtering</td>
<td>179</td>
</tr>
<tr>
<td>7.4</td>
<td>Evaluating loop untangling using global constraint</td>
<td>184</td>
</tr>
<tr>
<td>7.5</td>
<td>Evaluating loop untangling using global constraint, with special cases enabled</td>
<td>185</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Preliminaries

1.1 Introduction

In the field of combinatorial optimisation, much effort is spent on pushing the bounds of possibility. This thesis focuses instead on a perhaps less glamorous, but equally important form of progress. That is, making problems which our solvers can now solve with ease, easy to solve in practice.

In the next chapter, I argue that to achieve widespread use we need to make our technology accessible for software developers, and further, that the ideal way to do this is to provide an interface to optimisation solvers using general purpose programming languages in a natural way. Typical optimisation libraries demand that programmers explicitly build a data structure representing the problem domain and the constraints. However, in most cases a representation of the problem domain will already exist in the form of types or classes, and the constraints are more naturally represented using a function. Chapter 2 explains how this observation leads to a more natural optimisation interface, which we call a native language interface. The idea is further illustrated with examples in Java and Haskell, and then a prototype implementation for Java is presented in Chapter 3.\(^1\)

In order to implement a native language interface to an optimisation solver, the native problem definition should be automatically translated into a standard constraint model acceptable to the solver. In turn, the solver output must be transformed back into the expected form. The automatic translation from code to constraints has already been studied for applications such as test case generation, and model checking. Although it is possible to implement a native language optimisation library using these existing techniques, the models

\(^1\)Chapter 2 includes material published in the paper ‘Rethinking the quest for declarativity’, and Chapter 3 is based on the paper ‘Optimisation modelling for software developers’. Details for these papers are listed in the preface.
produced in this fashion are excessively large, with weak inference. The bulk of this thesis is dedicated to exploring better translation techniques and specialised constraints to provide stronger inference. The progress made should also be transferable to the more traditional applications of symbolic execution mentioned above.

Chapter 4 introduces a new translation technique which directly models the relationships between assignments and variable references, rather than modelling the state at each execution step.² Experimental results show that this technique is much more effective for our problems than the standard approaches used in the fields of constraint programming and satisfiability modulo theories. In some cases the generated models can be solved almost as quickly as a hand written model for the original problem.

Chapter 5 suggests an alternative way of expanding while and for loops during the pre-processing phase of the translation, demonstrating the potential to greatly simplify models in some cases.³ This new technique is called loop untangling, and it works by trading reduced complexity within each copy of the loop body for a greater number of copies and uncertainty about their order of execution. The application of this technique requires two new global constraints - subpath and reaching definitions, to efficiently manage the uncertainty we introduce to the execution path.

The subpath constraint is required to ensure that the chosen execution path has no cycles. In Chapter 6 we investigate the (existing) circuit constraint, exploring how this constraint can be extended to support our required subpath constraint, and how both versions of the constraint can be implemented in a lazy clause generation solver.⁴

Chapter 7 introduces the reaching definitions constraint, which provides a new way of linking each variable reference to its reaching assignment. This new constraint is capable of stronger reasoning than the previously used decomposition (from Chapter 4), and unlike this decomposition does not assume a known ordering for assignments, thereby facilitating the use of loop untangling. Experimental results in this chapter confirm that loop untangling can greatly improve the models produced from problem definitions making use of complex loops.

Chapter 8 concludes with a summary of the presented contributions. The remainder of this chapter covers various background information assumed in the rest of the thesis.

### 1.2 Combinatorial Optimisation

Combinatorial optimisation involves finding the best of a finite (but usually large) set of possible solutions to a problem. The fields of Constraint Programming (CP) and Mathematical Programming (MP) both investigate techniques

---

²Chapter 4 is based on the paper ‘Modelling destructive assignments’ (details in preface).
³Chapter 5 is based on the paper ‘Loop untangling’.
⁴Chapter 6 is based on the paper ‘Explaining circuit propagation’.
for combinatorial optimisation, and both have enjoyed significant success solving large scale industrial problems. The focus of this thesis is providing software developers with better access to CP technology. However, the same basic technique could be applied using MP technology or any other constraint-based solving approach.

1.2.1 Constraint Programming (CP)

The field of Constraint Programming (CP) is concerned with solving constraint satisfaction and optimization problems. To use CP solution techniques for a given problem, it must first be modelled by a set of decision variables, and a set of constraints over those variables. Each decision variable has a domain of possible values, and the constraints specify (dis)allowed combinations of values for the variables. Optionally, an expression containing one or more decision variables is specified as the optimisation objective.

Given such a model, a constraint solving engine uses sophisticated algorithms to search for a feasible or optimal solution. A feasible solution is a set of values for the decision variables chosen from their respective domains such that all constraints are satisfied. An optimal solution is a feasible solution which maximizes (or minimizes) the value of the expression given as the optimisation objective.

More formal definitions introducing the notation to be used throughout this thesis are given below.

Variables and domains

We consider a set of variables $\mathcal{V}$, each $v \in \mathcal{V}$ having a finite set of possible values $D(v)$. A domain $D$ is a complete mapping $v \mapsto D(v)$ for $v \in \mathcal{V}$.

Let $D_1$ and $D_2$ be domains for the same set of variables $\mathcal{V}$. We say that $D_1$ is stronger than $D_2$, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$.

We assume an initial domain $D_{init}$ such that all domains $D$ that occur during the solving process will be stronger i.e. $D \sqsubseteq D_{init}$.

Valuations, constraints, and solutions

A valuation $\theta$ is a mapping of variables to values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation $\theta$ to map expressions or constraints involving the variables in the natural way.

Let $\text{vars}$ be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we define a valuation $\theta$ to be an element of a domain $D$, written $\theta \in D$, if $\theta(v) \in D(v)$ for all $v \in \text{vars}(\theta)$.

A constraint $c$ is a set of valuations over $\text{vars}(c)$ giving the allowable combinations of values for that set of variables. A domain $D$ is consistent with constraint $c$ if for at least one valuation $\theta$ in $c$, $\theta \in D$. We say a variable $v$ is
fixed by domain $D$ if $|D(v)| = 1$, since it implies that $v$ takes the same value for any valuation $\theta \in D$.

A domain represents a solution if it fixes all variables, and is consistent with all constraints.

**Propagation-based constraint solving**

In finite domain propagation solvers, constraints are implemented by propagators. A propagator $f$ for $c$ is a contracting and weakly monotonic [99] function over domains such that for all domains $D \subseteq D_{\text{init}}$: $f(D) \subseteq D$ and no solutions are lost, i.e. $\{\theta \in D | \theta \in c\} = \{\theta \in f(D) | \theta \in c\}$. The power of propagation-based constraint solving arises from the fact that a propagator for a particular constraint is completely independent of other propagators.

The basic solving algorithm involves interleaving the application of propagators with search. Each search step heuristically strengthens the current domain $D$ to $D'$. If subsequent propagation (possibly combined with further search) reveals that there are no solutions $S \subseteq D'$, then the solver backtracks to $D$ and attempts an alternative search step.

For satisfaction problems, if the current domain represents a solution, the solver may either terminate the search or continue searching for alternative solutions. For optimisation problems, upon reaching a domain $D_s$ which represents a solution, a new constraint $c_{\text{obj}}$ is added excluding valuations giving an equal or less desirable objective value. This triggers backtracking and a continuation of the search, as $D_s$ is not consistent with $c_{\text{obj}}$.

For more information on propagation-based constraint solvers, see e.g. [98].

**1.2.2 Boolean satisfiability (SAT) solvers**

The strength of CP is the ability to apply specialised propagation algorithms for specific high-level constraints. An alternative is to convert all constraints and variables to a basic form, and then develop highly efficient algorithms for that form. This is the approach taken for Boolean satisfiability (SAT) solvers, where all variables are Booleans and all constraints are clauses. Modern SAT solvers (e.g. [85]) are capable of extremely efficient search and propagation, with highly effective general purpose algorithms and heuristics. In particular, search heuristics based on the activity level of variables (how frequently they have been involved in conflicts), and techniques for nogood learning (preventing futile exploration using new clauses summarising reasons for failure) have been very successful.

**1.2.3 Lazy Clause Generation (LCG)**

Lazy clause generation (LCG) [88] is a hybrid constraint solving technique combining the benefits of finite domain propagation with Boolean satisfiability techniques. Propagators in an LCG solver are essentially clause generators for an
underlying SAT solver. Instead of applying propagator $f$ to domain $D$ to obtain $f(D)$, whenever $f(D) \neq D$ the propagator builds a clause that encodes the change in domains. The underlying SAT solver can then use these clauses to generate nogoods and guide search.

Representing integer domains

In order to explain propagations over integer domains, these must be linked to a Boolean representation to be used in the SAT solver. An integer variable $x$ with domain $D_{init}(x) = [l..u]$ is represented using the Boolean variables $J_x = l, \ldots, J_x = u$ and $J_x \leq d, \ldots, J_x \leq u - 1$. The variable $J_x = d$ is true if $x$ takes the value $d$, and false for a value different from $d$. Similarly the variable $J_x \leq d$ is true if $x$ takes a value less than or equal to $d$ and false for a value greater than $d$. The inclusion of these bounds literals is the main feature distinguishing lazy clause generation from other attempts to incorporate nogood learning in an FD solver, such as [68].

Not every assignment of Boolean variables is consistent with the integer variable $x$, for example \{ $J_x = 3, J_x \leq 2$ \} (i.e. both Boolean variables are true) requires that $x$ is both 3 and $\leq 2$. In order to ensure that assignments represent a consistent set of possible values for the integer variable $x$, a set of clauses $DOM(x)$ are added to the SAT solver. We let $DOM = \cup \{ DOM(v) \mid v \in V \}$.

Assuming an initial domain $D_{init}(x) = [l..u]$, the clauses $DOM(x)$ encode the following constraints.

\[
\begin{align*}
[x \leq d] & \rightarrow [x \leq d + 1], & l \leq d < u \\
[x = l] & \leftrightarrow [x \leq l] \\
[x = d] & \leftrightarrow ([x \leq d] \land \neg[x \leq d - 1]), & l < d < u \\
[x = u] & \leftrightarrow \neg[x \leq u - 1]
\end{align*}
\]

Any assignment $A$ on these Boolean variables can be converted to a domain: $domain(A)(x) = \{ d \in D_{init}(x) \mid \forall [c] \in A, vars([c]) = \{ x : x = d \models c \} \}$. That is, the domain includes all values for $x$ that are consistent with all of the Boolean variables related to $x$. It should be noted that the domain may assign no values to some variable.

Example Assume $D_{init}(x_1) = D_{init}(x_2) = [0..10]$. The assignment $A = \{ [x_1 \leq 5], [x_1 \leq 1], [x_1 = 4], [x_2 \leq 7], [x_2 \leq 3] \}$ is consistent with $x_1 = 2, x_1 = 3$ and $x_1 = 5$. Hence, $domain(A)(x_1) = \{ 2, 3, 5 \}$. Similarly $domain(A)(x_2) = [4..7]$.

Propagators

As stated previously, in lazy clause generation a propagator is extended to not only map from domains to domains but also to generate clauses describing the reasons for each propagation. When $f(D) \neq D$ the propagator $f$ must be able to produce a set of clauses $C$ which explain the domain changes.
Example Asssuming a current domain $D$ with $D(x_1) = \{2, 3, 5\}$ and $D(x_2) = \{4..7\}$, consider the propagator $f$ for $x_1 \leq x_2 - 3$. When applied to domain $D$ it obtains $f(D)(x_1) = \{2, 3\}$, $f(D)(x_2) = \{5..7\}$. The clausal explanation of the change in domain of $x_1$ is $\lbrack x_2 \leq 7 \rbrack \rightarrow \lbrack x_1 \leq 4 \rbrack$. This becomes the clause $\lbrack x_2 \leq 7 \rbrack \lor \lbrack x_1 \leq 4 \rbrack$. 

In practice explanations can be produced lazily when it is discovered that the corresponding propagation contributed to failure. The lazy clause generation solver keeps track of the domains $D$ of variables $\mathcal{V}$ and the equivalent state $A$ of the Booleans in $DOM (D = domain(A))$. When a propagator detects an inconsistency and triggers failure, it provides an explanation $c \rightarrow false$ where $c$ is a conjunction of true literals. This initial nogood $c$ is transformed into an equivalent nogood containing at most one literal which became true at the current decision level. This is achieved by repeatedly selecting from $c$ a literal $l$ which was set true by propagator $p$, asking $p$ to provide an explanation in the form of a conjunction of literals $l_1 \land \cdots \land l_n$ which imply $l$, and then replacing $l$ in $c$ with the conjunction $l_1 \land \cdots \land l_n$. Once this learning process has derived a final nogood, the solver adds this nogood to the SAT constraint store and backtracks or backjumps.

Example Consider a problem with the following constraints and initial domain:

\[
\begin{align*}
  &x_1 \leq x_2 - 3 \\
  &x_1 \leq x_4 \\
  &x_1 + x_2 + x_3 \geq 14
\end{align*}
\]

\[
\begin{align*}
  &D_{init}(x_1) = \{0..7\} \\
  &D_{init}(x_2) = \{0..10\} \\
  &D_{init}(x_3) = \{0..10\} \\
  &D_{init}(x_4) = \{0..10\}
\end{align*}
\]

Suppose search chooses to set $\lbrack x_3 \leq 2 \rbrack$, there is no propagation. Then if search sets $\lbrack x_4 \leq 6 \rbrack$, the propagator for $x_1 \leq x_4$ sets $\lbrack x_1 \leq 6 \rbrack$, after which the propagator for $x_1 + x_2 + x_3 \geq 14$ sets $\lbrack x_2 \geq 6 \rbrack$. Now assume search sets $\lbrack x_2 \leq 7 \rbrack$. The propagator for $x_1 \leq x_2 - 3$ sets $\lbrack x_1 \leq 4 \rbrack$. Then the propagator for $x_1 + x_2 + x_3 \geq 14$ triggers failure, giving the initial nogood $\lbrack x_3 \leq 2 \rbrack \land \lbrack x_2 \leq 7 \rbrack \land \lbrack x_1 \leq 4 \rbrack \rightarrow false$. To remove $\lbrack x_1 \leq 4 \rbrack$ we ask the propagator that set it ($x_1 \leq x_2 - 3$) for an explanation $\lbrack x_2 \leq 7 \rbrack \rightarrow \lbrack x_1 \leq 4 \rbrack$ and replace obtaining $\lbrack x_3 \leq 2 \rbrack \land \lbrack x_2 \leq 7 \rbrack \rightarrow false$. Since this only contains one literal true at the last level we add it to the constraint store and backtrack.

The advantages of lazy clause generation over a standard FD solver (e.g. [98]) are that we automatically have the nogood recording and backjumping ability of the SAT solver applied to our FD problem. We can also use activity counts from the SAT solver to direct the FD search.

1.2.4 Satisfiability Modulo Theories (SMT)

The technology most frequently used for program symbolic reasoning (e.g. for program verification or automated testing) is Satisfiability Modulo Theories (SMT) [19]. SMT is an extension of SAT where special purpose theory solvers are used to check satisfiability of parts of a formula, with a standard SAT engine.
handling the Boolean part of the problem and acting as an interface between
different theories. This allows higher level modelling of problems as well as
special purpose reasoning. For example, the theory of arithmetic can reason
directly about conjunctions of arithmetic equations, simultaneously eliminating
the need for a user to convert these equations into equivalent Boolean clauses,
and supporting more effective deduction.

There is a strong similarity between SMT and LCG, in that both technologies
combine specialised reasoning algorithms with a SAT solver. The main differ-
ence is the method of communication between propagators / theory solvers. In
an LCG solver, all propagators share the same representation of the current
domain. This limits the freedom of representation for different constraints, but
means information (in the form of reduced domains) can be passed directly dur-
ing propagation. In a typical SMT solver, each literal in the SAT solver belongs
to a particular theory and can have any interpretation appropriate for that the-
ory. Communication between theories only occurs through SAT propagation,
using clauses with literals belonging to different theories.

1.3 Java

The host programming language we use for our prototype is Java. For clarity
of the examples and discussion in later chapters, we explain some relevant Java
syntax and terminology here.

1.3.1 Classes and interfaces

Java is an object-oriented language, which means both data and operations are
tied to objects. Each object has a class, which defines the fields (data) and
methods (operations) associated with that object. Both fields and methods are
accessed via the dot operator. A new object is created using the new keyword
and a constructor method which must have the same name as the class. As an
example, the class declaration below says that an object of type Cupboard has
an associated Boolean value doorOpen, a method to open the door, a method to
close the door, and a constructor which sets the initial state of the door.

```java
class Cupboard {
    boolean doorOpen; // field
    Cupboard(boolean initialDoorState) { // constructor
        doorOpen = initialDoorState;
    }
    void openDoor() { doorOpen = true; } // method
    void closeDoor() { doorOpen = false; } // method
}
```

It is also possible to define an interface, which is essentially a list of required
methods. Classes implementing an interface must define the specified methods.
It is not possible to instantiate an interface directly. Instead one must construct
an object of a concrete class implementing that interface. In the below example both House and Cupboard implement the interface ThingWithDoor. Therefore a House or Cupboard object can be used anywhere a ThingWithDoor is expected.

```java
interface ThingWithDoor {
    void openDoor();
    void closeDoor();
}

class Cupboard implements ThingWithDoor {
    boolean doorOpen;
    Cupboard(boolean initialDoorState) {
        doorOpen = initialDoorState;
    }
    void openDoor() { doorOpen = true; }
    void closeDoor() { doorOpen = false; }
}

class House implements ThingWithDoor {
    int numDoors;
    int doorsOpen;
    House(int doors) {
        numDoors = doors;
        doorsOpen = 0;
    }
    void openDoor() { if(doorsOpen < numDoors) doorsOpen++; }
    void closeDoor() { if(doorsOpen > 0) doorsOpen--; }
}

1.3.2 Variables

Java is strongly typed, with every variable taking either a primitive type (e.g. boolean, int), or a class or interface as its type. A variable using an interface as its type can hold an object of any class implementing that interface (e.g. the thing variable in the example below can hold a House or a Cupboard). The methods included in the interface are guaranteed to be supported by the object, and no other methods may be called using this variable.

A field exists for the lifetime of its owning object. Other variables have a scope defined by the code block in which they are declared. These are called local variables. Method parameters are also local variables, which are only defined within the method body, meaning assignments to method parameters do not affect the caller. That is, parameters are passed by value. However, any changes to the state of objects passed to a method will be observable afterwards by the caller. In the example below both thing and n are local variables (thing is defined only within the method, and n is defined only within the for statement).
Also note the use of the dot operator: `thing.openDoor()` is a method invocation, while `c.doorOpen` is a field reference.

```java
boolean playWithDoors(ThingWithDoor thing) {
    for (int n = 0; n < 5; n++) {
        thing.openDoor(); // change the state of the caller's object
    }
    Cupboard c = new Cupboard(true); // create a new Cupboard object
    thing = c; // this does not change the caller's object
    thing.closeDoor(); // change the newly constructed Cupboard
    return c.doorOpen; // return the doorOpen value of the Cupboard
}
```

### 1.3.3 Generics

A more recent addition to Java is generics, which enables classes, interfaces and methods to be parameterised with a type (a class or interface). Only a basic understanding of generics in Java is necessary to follow the examples used in this thesis. The main thing to note is that type parameters are surrounded with angled brackets.

For example, the `List` class can be parameterised with the type of elements. A `List<House>` holds `House` objects, while a `List<ThingWithDoor>` can hold any object whose class implements `ThingWithDoor`. Methods can be parameterised independently of the owning class, in which case the type parameter is declared at the start of the method declaration (before the return type). For example, the `getLast` method below can work with any type of `List`, but the compiler knows the return type will be the same as the the item type for the input list.

```java
<T> T getLast(List<T> list) {
    int s = list.size();
    return list.get(s-1);
}
```
Chapter 2

Native Programming Language Access to CP

2.1 Introduction

This chapter introduces the notion of a native interface to CP within a general purpose programming language. The first section explains the motivation behind such a thing, before describing exactly what is meant by this term. The next section illustrates the concept using examples from two different programming languages. Finally we cover related work, including a detailed comparison of a native interface to a conventional CP library.

2.1.1 Motivation

Despite significant success solving high impact combinatorial optimisation problems, Constraint Programming (CP) is not widely used for smaller scale applications. The available tools require a high level of specialised expertise, which makes their use prohibitively expensive for lower-valued problems.

Before attempting to address this usability problem, we must consider our intended use case. Who should be using the tools we provide, what skills will those people possess and what will their objective be? In considering these questions, I claim that widespread use of CP can only be achieved by allowing constraint solving to be incorporated into application-specific software. We cannot expect end users to undergo significant constraint-specific training, or to spend excessive time defining each individual problem to be solved. Instead they should be able to specify problem parameters and interpret solutions via a user-friendly, application specific interface.

It is also not realistic to expect constraint problems to be defined and solved in isolation. Rather, in many cases constraint solving functionality will form only a small part of a wider software system whose main purpose is something else, such as record keeping. It is therefore important not only that problems
can be defined easily, but also that the solving of the problem can be easily incorporated into a wider program. If an optimisation tool is otherwise easy to use but it is awkward, time consuming and error-prone to achieve the required integration with a wider system, this will greatly discourage potential users.

This has important implications for our choice of target users for our tools. It means both that usability improvements targeting mathematicians or CP experts are not especially useful for our purpose, and that aiming for a problem definition interface simple enough to be used by anyone (e.g. a natural language interface) is unnecessarily ambitious. A more practical choice is to aim for an interface which is intuitive and convenient for application programmers. This person should possess relevant skills, and will be involved in the integration effort already.

The creation of CP tools which are easier for software developers to use has great potential to increase the profile and impact of research in our field. Currently, application of the technology is limited mainly to the hobbies and research related projects of optimisation experts/enthusiasts, and large-scale high-valued industrial applications (where the expense of hiring an expert is justified). If general software developers could easily access the technology to solve simple problems, many more applications aimed at individuals and small businesses would become viable. The problems solved by these applications would not be particularly interesting or ground-breaking, but capturing this market would greatly increase both the volume of end users benefiting from the technology, and the interest in and data available for further research.

2.1.2 Definition of a native interface

Having decided that application programmers are our target users, we need to provide an interface to CP solvers that utilises the skills they have and does not assume knowledge they do not have. It should also simplify the process of embedding CP technology into a wider application.

Currently the main limiting factor for usability is the requirement for the programmer to define the constraint problem using an unfamiliar paradigm which is incompatible with the rest of the software application. When attempting to embed CP solving into a wider program, some representation of the problem domain almost certainly already exists. It is likely that at least some of the constraints representing rules or business logic are also present. However, in order to access CP technology, we insist that the programmer create a new model of the problem using foreign modelling concepts with which they have no experience. The decisions to be made must be declared as (usually primitive-typed) variables with specified domains, while the relations which must hold between the chosen options, and the objective function, are defined using declarative constraints. The programmer must then write inelegant and error-prone code to translate between this new representation and the existing one. While the CP paradigm provides great benefits to experts tackling difficult problems, for a software developer trying to include simple optimisation capability in a wider application it is confusing and highly inconvenient.
Fortunately, it is not actually necessary to require the programmer to build a conventional constraint model directly. The core of an optimisation problem definition is a function defining the relationship between the decisions and the outcome (the outcome is either that the solution is invalid, or that it has a certain objective value). Standard programming languages already have the capacity to define functions, and programmers should be adept at this. If we can allow programmers to use this capacity to define the optimisation problem, and then automatically convert that definition into a form suitable for a solver, then we can greatly improve usability.

The key barrier to this is the treatment of decisions. As soon as we introduce decision variables we necessitate a separate representation of the problem based on these. I propose that instead decisions should be understood as nondeterministic procedures. Imagine if the programmer could use some sort of magic oracle to make perfect individual decisions, and then use standard code to combine these individual answers into a solution. Clearly this would be very easy; unfortunately it is also impossible. However, if decisions are represented as nondeterministic procedures, then it is possible to create an interface which appears to the programmer to be very close to this.

Code which combines individual decisions made by an oracle (or the results of several nondeterministic procedures) into a solution defines a pool of candidate solutions. If we also obtain code to evaluate a candidate solution (which should also be straightforward to write), this is enough to define the constraint problem. A native interface to CP allows the programmer to define problems in this way and receive solutions exactly as if their own code had been given a perfect oracle.

Essentially the programmer says:

1. This code builds a candidate solution from individual decisions.
2. This code checks if a candidate solution is feasible.
3. This code calculates the score for a solution. (optional)
4. Build the (best) solution.

Note that the actual computation happens in the last step, and the code does not specify how that step should be achieved. In this sense the interface is declarative even though it may use procedural code.

In theory the final step could be achieved by repeatedly executing the candidate building code with a randomised oracle, each time testing the result for feasibility and score, until one is satisfied that no better solution is likely. This possible implementation allows the programmer to understand what the final step means, but obviously it should not be used in practice. Instead we treat the provided code (1-3) not as code to be executed, but as a declarative specification of the problem to be solved. The library automatically converts that specification into a standard constraint model, solves it, and then converts the result back into the required form.
The next section uses examples in two different programming languages to illustrate more clearly what such an interface would look like from the programmer’s perspective. A proof-of-concept implementation for Java is discussed in Chapter 3.

2.2 Examples

In this section, example interfaces for two very different programming languages (Haskell and Java) are used to explain in more detail what a native interface to CP might look like and how it would be used. First the interface for each language is defined, and then we examine complete programs solving an example problem taken from the First International Lightning Model and Solve competition, held at CP 2013 [104].

2.2.1 Haskell

A Haskell version of a native interface to CP is shown in Figure 2.1. In Haskell we have the advantage that functions are first class. So the three pieces of code provided by the programmer (candidate-building, checking and scoring) can simply be functions which will be passed to the ‘find the best solution’ function (the interface provides three of these: `satisfy`, `minimise` and `maximise`).

The candidate-building function returns type `DeciderState s`, where `s` is the type of a candidate solution. It should make one or more calls to the provided decision functions (`chooseBool`, `chooseInt` and `chooseOne`), combining the results to build a candidate. Since `DeciderState` is a `State` this can be done using familiar `do` notation (as will be shown in the example). We are using the `State` monad (rather than having the function simply return type `s`) to account for the fact that a value of type `s` will be just one from the pool of candidate solutions. The state within `DeciderState` is understood to determine which values are returned by `chooseBool`, `chooseInt` and `chooseOne`, and therefore which candidate is produced.

The checking function takes the candidate type `s` and returns a `Bool` which is true if the candidate is acceptable, while the scoring function takes an `s` and returns a value of some orderable type `o`.

This interface is very simple and easy to understand. There is only one type and very few functions to become familiar with, all of which have a very clear meaning. The type signatures of the main functions provide obvious cues for the programmer about the code they need to write, and that code does not need to use any different syntax or special types (`DeciderState` is a provided type, but it is really just a `State`).
module CP where
import Control.Monad.State

type DeciderState = State Int

satisfy :: DeciderState a -> (s -> bool) -> s
minimise :: Ord o => DeciderState s -> (s -> bool) -> (s -> o) -> s
maximise :: Ord o => DeciderState s -> (s -> bool) -> (s -> o) -> s

-- make a yes/no decision
chooseBool :: DeciderState Bool
-- choose a number between the first arg and the second
chooseInt :: Int -> Int -> DeciderState Int
-- choose one item from the given list
chooseOne :: [a] -> DeciderState a

Figure 2.1: Native Haskell interface to CP

2.2.2 Java

We now consider how the same thing might be achieved in Java. Unfortunately in Java we cannot pass functions as arguments,\(^1\) so instead we define an interface for the programmer to implement.

A class implementing CSP must contain a candidate-building method build, and a checking method check. The build method is passed a Decider object, which provides the same decision functions as in the Haskell version (chooseBool, chooseInt and chooseOne). Instead of returning a candidate the build method should change the state of the CSP object itself to represent the candidate. The check method will then use this state to determine if the candidate is acceptable. For optimisation problems the programmer must implement the CSPopt interface which extends CSP with another method to calculate a score (Comparable is a standard Java interface for classes having a natural ordering).

This time the ‘find the best solution’ methods belong to a library class called Solver. The buildMinimal method (for example) can be understood to call the build method of the provided CSP object using a Decider which makes optimal decisions (those producing a state for which calling check will return true and calling score will return the smallest possible value).

Once again there are very few library classes and methods to become familiar with, and the CSP/CSPopt interface prompts the programmer to write the required code. The decision making code can be more straightforward than the Haskell version because Java has no qualms about methods returning different values each time they are called. However the inability to directly pass functions as arguments makes the semantics of buildSatisfactory / buildMinimal / buildMaximal less clear than that of their Haskell counterparts.

---

\(^1\)Recent versions of Java include the facility to pass code as an argument through the use of lambda expressions. Making use of this facility one could define an interface for Java similar to the one proposed for Haskell.
interface CSP {
    void build(Decider d);
    boolean check();
}  

class Decider {
    boolean chooseBool();
    int chooseInt(int min, int max);
    T chooseOne(Set<T> options);
}  

interface CSPopt extends CSP {
    Comparable score();
}  

class Solver {
    void buildSatisfactory(CSP problem);
    void buildMinimal(CSPopt problem);
    void buildMaximal(CSPopt problem);
}  

Figure 2.2: Native Java interface to CP

2.2.3 M-Queens

To illustrate the use of these native interfaces, we will use an example problem M-Queens, which is an optimisation extension of N-Queens. As with N-Queens, the queens placed on the board must not be able to attack each other. That is, no row, column or diagonal can contain more than one queen. The difference is that rather than aiming to place \( n \) queens on an \( n \)-by-\( n \) board, the goal is to place as few queens as possible while still covering every square on the board (making it impossible to add another queen).

Haskell implementation

Figure 2.3 shows a complete Haskell program solving M-Queens using the native interface to CP introduced above. The main function reads the parameter \( n \) from standard input, calls mqueens to find an optimal placement of queens for an \( n \)-by-\( n \) board, and then prints the result to standard output. In the competition from which this example was taken [104], the required output format for solutions was a list (using the same syntax as Haskell lists) giving the column for the queen in each row, or 0 if no queen is placed in that row. Our program uses this same list representation for solutions to make producing the output convenient (it is sufficient to call the built-in function print as shown).

The \texttt{mqueens} function simply calls \texttt{minimise} passing the three required functions as arguments.

- The candidate-building function \texttt{choosequeens} takes the parameter \( n \) and uses \texttt{replicateM} (from \texttt{State}) and \texttt{chooseInt} (from \texttt{CP}) to choose \( n \) numbers between 0 and \( n \), returning them in a list. The number at index \( i \) represents the column for the queen in row \( i \) (0 means no queen in row \( i \)).
- The checking function \texttt{validqueens} takes the parameter \( n \) and a list representing a candidate solution and checks that all board positions are covered by the chosen coordinates (using \texttt{all coveredby}), and no pair of chosen coordinates clash (using \texttt{noclash}). The \texttt{covers} function defines what it means for one coordinate to cover (or clash with) another: the two coordinates must have the same row or column, or the same sum or difference between row and column (as then they are on the same diagonal).
• The scoring function `numqueens` counts the number of positive numbers in its input list. This corresponds to the number of queens placed on the board.

Note that almost all of this code uses standard types and functions and can be understood independently from the CP module. In fact the CP module is only used on three lines (highlighted with arrows): there is one use of `minimize`, one use of `chooseInt`, and the `choosequeens` function returns a `DeciderState`. Minimal use of CP-specific code makes this program easy to understand (and to write) for someone familiar with Haskell. Furthermore, the checking functions `validqueens`, `covered`, `noclash` and `covers` would likely be needed for testing pur-

```haskell
main = do
    n <- readLn
    let chosenCols = mqueens n
    print chosenCols

mqueens :: Int -> [Int]
mqueens n = minimise (choosequeens n) (validqueens n) numqueens ←

choosequeens :: Int -> DeciderState [Int] ←
choosequeens n = replicateM n (chooseInt 0 n) ←

validqueens :: Int -> [Int] -> Bool
validqueens n colchoices =
    let allcoords = [(i,j) | i <- [1..n], j <- [1..n]]
        pairs = zip [1..n] colchoices
        chosencoords = filter (\(r,c) -> c>0) pairs
        in all (coveredby chosencoords) allcoords &&
            noclash chosencoords

coveredby :: [(Int,Int)] -> (Int,Int) -> Bool
coveredby chosenpositions pos =
    any (covers pos) chosenpositions

noclash :: [(Int,Int)] -> Bool
noclash coords =
    let pairs = [(c1,c2) | c1 <- coords, c2 <- coords, c1/=c2]
        covers’ (a,b) = covers a b
        in not (any covers’ pairs)

covers :: (Int,Int) -> (Int,Int) -> Bool
covers (x,y) (i,j) =
    (x == i) || (y == j) || ((x-y) == (i-j)) || ((x+y) == (i+j))

numqueens :: [Int] -> Int
numqueens = length . filter (>0)
```

Figure 2.3: Haskell program solving the M-Queens problem.
poses regardless of the solving method.

**Java implementation**

We now consider a Java version of the same program.

To demonstrate that we do not have to build the solution in a simple format such as a list of integers, this version instead uses a Map from row number to column number. Only rows in which a queen is placed should be included as a key in the map. Map is a standard Java library class, but it would also be possible (in both the Java and Haskell interface) to build a candidate solution using user-defined classes. For example, another valid representation for the selected queen positions would be a Set of Coordinate objects, each Coordinate having a row and a column.

The majority of the code is shown in Figure 2.4 (for space reasons print is shown separately in Figure 2.5). The main method retrieves the parameter \( n \), constructs an MQueens object for this size board, then calls buildMinimal to set up a minimal placement before printing the result using the print method of MQueens.

To be accepted as an argument to buildMinimal, the MQueens class needs to implement the CSPopt interface by defining the methods build, check and score.

- The build method iterates through the rows deciding first whether or not a queen should be placed in this row, and then if so which column should be used. The chosen column for each row is recorded in colForRow.
- The check method checks that every board coordinate is covered by one of the chosen queen positions (stored in colForRow), and that for every pair of (different) positions in this map, the queens cannot attack each other.
- The score method simply returns the size of the map (the number of keys), which gives the number of queens placed on the board.

The print method of MQueens (Figure 2.5) prints the solution in the required format. Recall that buildMinimal can be understood to call build with an optimal Decider. So when the print method is called (in main immediately after the call to buildMinimal) the colForRow map will contain optimal coordinates.

Once again almost all of the code is independent of the CP library (arrows indicate lines using library methods or classes). Comparing the Haskell and Java solutions, the Java code is unsurprisingly longer (Java is famously verbose), and the Haskell version is more reminiscent of a model written in a dedicated constraint language (with its use of not, any and all), but they are actually very similar. Which version is easier to understand would almost certainly depend on the familiarity of the reader with the two different languages. For this reason it would be beneficial to create native interfaces for a variety of languages rather than attempting to choose a single language which is most suited to the task.
```java
class QueensMain {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        MQueens mq = new MQueens(n);
        Solver.buildMinimal(mq);
        mq.print(System.out);
    }
}

class MQueens implements CSPopt {
    int n;
    Map<Integer,Integer> colForRow;
    MQueens(int n) {
        this.n = n;
        colForRow = new HashMap<Integer,Integer>();
    }
    void build(Decider d) {
        for(int row = 1; row <= n; row++)
            if(d.chooseBool()) // put a queen in this row?
                colForRow.put(row, d.chooseInt(1, n));
    }
    boolean check() {
        for(int row = 1; row <= n; row++)
            for(int col = 1; col <= n; col++)
                if(!coveredByChosen(row,col)) // this position is not covered
                    return false;
        for(int r1 : colForRow.keySet())
            for(int r2 : colForRow.keySet())
                if(r1 != r2 && covers(r1, colForRow.get(r1), r2, colForRow.get(r2)))
                    return false; // this pair of queens can attack each other
        return true;
    }
    boolean coveredByChosen(int row, int col) {
        for(int qrow : colForRow.keySet())
            if(covers(qrow, colForRow.get(qrow), row, col))
                return true;
        return false;
    }
    boolean covers(int r1, int c1, int r2, int c2) {
        return (r1==r2 || c1==c2 || (r1-c1 == r2-c2) || (r1+c1==r2+c2));
    }
    Integer score() { return colForRow.size(); }
}
```

Figure 2.4: Java program solving the M-Queens problem (excluding print method).
void print(PrintStream ps) {
    ps.print("[");
    for (int row = 1; row <= n; row++) {
        if (row > 1)
            ps.print(", ");
        if (colForRow.containsKey(row))
            ps.print(colForRow.get(row));
        else
            ps.print(0);
    }
    ps.println("]");
}

Figure 2.5: Print method extracted from MQueens class in Figure 2.4.

2.3 Comparison with a Conventional CP Library

If our target users are programmers then there are significant advantages to allowing problems to be defined using existing general purpose programming languages. Programmers are already familiar with and practiced at using these languages, mature editors/IDEs exist, and integration with a wider program (which is essential for widespread use of CP) is much easier.

For many programming languages there already exist one or more CP libraries designed to facilitate the integration of constraint solving into a wider program. Here we give an extended example comparing the native CP interface for Java proposed above with a conventional CP library for Java: Choco [67].

The example we will use is an application to generate crossword puzzles. The user provides a list of words and the maximum height and width of the puzzle, and the program must create a layout including as many intersections between words as possible, and then display this to the user graphically. Note that not all words need be used.

Obviously any words which intersect must agree on the character in the intersecting cell. For a well formed puzzle we might also wish to disallow overlapping between words in the same direction, and to require that any vertically adjacent cells both containing letters must be part of a vertical word (and similarly for horizontally adjacent cells). However, in the interest of keeping the example simple we will leave out these constraints.

2.3.1 Decisions

To define an optimisation problem it is necessary to specify the decisions to be made and the effect of each choice on the outcome. For this particular example, the decisions to be made are which words to include, and where to place those words (which cell should the word start in and should it be down or across).

A build method which makes requests for these decisions to be made and then records the result is shown below. We iterate through the potential words, asking the Decider to choose whether or not to include each one. If the word is to
be included, the Decider is asked to decide whether to place the word vertically or horizontally, and then which row and column the word should start in. We choose the direction first, so that we can compute which starting positions will not run the word off the edge of the grid. The chosen location is recorded in a new Word object which is added to our list of included words.

```java
void build(Decider decider) {
    for (String wordString : words) {
        if (decider.chooseBool()) // decide whether to include the word
        {
            Word word = new Word(wordString);
            word.direction = decider.chooseFrom(Direction.values());
            int maxRow = numRows - 1;
            int maxCol = numCols - 1;
            if (word.direction == Direction.Horizontal)
                maxCol -= wordString.length();
            else
                maxRow -= wordString.length();
            word.startCol = Decider.chooseInt(0, maxCol);
            word.startRow = Decider.chooseInt(0, maxRow);
            includedWords.add(word);
        }
    }
}
```

Let us contrast this with a traditional model constructed in Choco. We will need to define a variable for each word to represent whether or not it should be included, plus another binary variable to represent the direction and two integer variables for the starting row and starting column.

```java
CPModel m = new CPModel();
int numWords = words.size();

// 1 is true, 0 is false
IntegerVariable[] includeVars =
    Choco.makeBooleanVarArray("include", numWords);
    m.addVariables(includeVars);

// 1 is horizontal, 0 is vertical
IntegerVariable[] directionVars =
    Choco.makeBooleanVarArray("direction", numWords);
    m.addVariables(directionVars);

IntegerVariable[] startRows =
    Choco.makeIntVarArray("row", numWords, 0, numRows-1);
    m.addVariables(startRows);

IntegerVariable[] startCols =
    Choco.makeIntVarArray("col", numWords, 0, numCols-1);
    m.addVariables(startCols);
```
Note that we have to define these variables for every word even though not all of them will be included in the puzzle. In the previous version, although these variables are created behind the scenes, the programmer never has to see them. The code makes it very clear that the decision about where to place a word is only relevant if the word is to be included. This makes it easier to avoid accidentally referring to the wrong variables, and makes the model easier to understand.

Another difference is that in Choco we have to represent all of our decisions as integers. This means we will need to remember that 0 is false and 1 is true, and that we have assigned 1 to mean horizontal while 0 means vertical. We could of course define int constants for this purpose, but it is still somewhat error prone.

Finally, in the Choco version, in order to make sure the words do not run off the edge of the grid, we need to add explicit constraints, as shown below. The clumsy expression building code is required because Java does not allow operator overloading. In the procedural model these constraints were incorporated into the choice of row and column by computing the maximum starting positions after choosing the direction.

```java
for(int i = 0; i < numWords; i++)
{
    int wordLen = words.get(i).length();
    // if the word is vertical, row <= numRows - (wordLen + 1)
    m.addConstraint(
        Choco.implies(Choco.eq(0, directionVars[i]),
                     Choco.leq(startRows[i], numRows - (wordLen+1))));
    // if the word is horizontal, col <= numCols - (wordLen + 1)
    m.addConstraint(
        Choco.implies(Choco.eq(1, directionVars[i]),
                     Choco.leq(startCols[i], numCols - (wordLen+1))));
}
```

### 2.3.2 Constraints

The next requirement of a problem definition is a specification of how the decisions affect the outcome. In this case we need to know how the choice of which words to include and their positioning affects the character assigned to each cell and the number of word intersections. We consider the native interface first.

Having decided which words to include and where to put them, we can straightforwardly calculate the character assigned to each cell of the grid. We use a list of lists of Cell objects, each of which records the character assigned to this position. This is null initially and will remain so if no words intersect this cell. As we go, if we find ourselves attempting to assign a character to a cell which already contains a different character, we return false to indicate that the Decider has made a bad decision (two intersecting words do not agree on the character at the intersection). If we assign a character to a cell which already
has the same character, then we have found a valid intersection, so we add one to our count. The score method simply returns this count of word intersections, as this is the measure we are using for the quality of the puzzle.

```java
boolean check() {
    numIntersections = 0;
    for (Word w : includedWords) {
        int row = w.startRow;
        int col = w.startCol;
        for (Character c : w.wordString().toCharArray()) {
            Cell cell = cells.get(row).get(col);
            if (cell.assignedChar() != null) {
                if (cell.assignedChar() == c) {
                numIntersections++;
                } else {
                    return false; // character clash
                }
            }
            cell.setAssignedChar(c);
            if (w.direction == Direction.Horizontal) col++;
            else row++;
        }
    }
    return true;
}
Comparable score() {
    return numIntersections;
}
```

Moving back to the traditional model, in order to encode these constraints and the objective function we need to define some more decision variables to represent the index of the cell in which each character of each word falls (if that word is included). Then for every pair of characters in different words, we can build an expression for whether or not they form an intersection. We need to disallow intersections for characters which are different, and count the number of intersections we have for characters which are the same. The code is shown below.

```java
// For each character of each word, create a variable representing the index // of this character in the grid. Store these in a list of lists.
List<List<IntegerVariable>> indices = new ArrayList<List<IntegerVariable>>();
for (int i = 0; i < numWords; i++) {
    List<IntegerVariable> indicesThisWord = new ArrayList<IntegerVariable>();
    indices.add(indicesThisWord);
    int wordLen = words.get(i).length();
```
for(int offset = 0; offset < wordLen; offset++)
{
    // The row offset is 0 if the word is horizontal
    IntegerVariable rowOffset = Choco.makeIntVar("", 0, wordLen);
    m.addVariable(rowOffset);
    int[] rowOffsetOptions = new int[] {offset, 0};
    m.addConstraint(Choco.nth(directionVars[i], rowOffsetOptions, rowOffset));
    IntegerVariable row = Choco.makeIntVar("", 0, numRows);
    m.addVariable(row);
    m.addConstraint(Choco.eq(row, Choco.plus(startRows[i], rowOffset)));

    // The column offset is 0 if the word is vertical
    IntegerVariable colOffset = Choco.makeIntVar("", 0, wordLen);
    m.addVariable(colOffset);
    int[] colOffsetOptions = new int[] {0, offset};
    m.addConstraint(Choco.nth(directionVars[i], colOffsetOptions, colOffset));
    IntegerVariable col = Choco.makeIntVar("", 0, numCols);
    m.addVariable(col);
    m.addConstraint(Choco.eq(col, Choco.plus(startCols[i], colOffset)));

    // the index is width*row + 1*col
    IntegerVariable index = Choco.makeIntVar("", 0, numCols*numRows);
    m.addVariable(index);
    int[] coeffs = new int[] {numCols, 1};
    IntegerVariable[] vars = new IntegerVariable[] {row, col};
    m.addConstraint(Choco.equation(index, vars, coeffs));
    indicesThisWord.add(index);
}

// Now for every pair of characters in a word, create an expression for
// when both words are included and the indices match.
// Either post the negation of this as a constraint or store it as an integer,
// depending on whether the characters are the same or different.
List<IntegerVariable> possibleIntersections = new ArrayList<IntegerVariable>();
for(int i = 0; i < numWords; i++)
{
    for(int offset1 = 0; offset1 < words.get(i).length(); offset1++)
    {
        IntegerVariable index1 = indices.get(i).get(offset1);
        for(int j = i+1; j < numWords; j++)
        {
            for(int offset2 = 0; offset2 < words.get(j).length(); offset2++)
            {
                IntegerVariable index2 = indices.get(j).get(offset2);
                Constraint charsIntersect = Choco.and(
                    Choco.eq(1,includeVars[i]),
                    Choco.eq(1,includeVars[j]),
                    Choco.eq(index1, index2));
                possibleIntersections.add(charsIntersect);
            }
        }
    }
}
if(words.get(i).charAt(offset1) == words.get(j).charAt(offset2)) {
    IntegerVariable binVar = Choco.makeBooleanVar(""");
    m.addVariable(binVar);
    m.addConstraint(Choco.reifiedConstraint(binVar, charsIntersect));
    possibleIntersections.add(binVar);
} else
    m.addConstraint(Choco.not(charsIntersect));
}
}

IntegerVariable[] intersectionsArray =
    (IntegerVariable[])possibleIntersections.toArray();
IntegerVariable numIntersections =
    Choco.makeIntVar("", 0, possibleIntersections.size());
    m.addVariable(numIntersections);
    m.addConstraint(Choco.eq(numIntersections, Choco.sum(intersectionsArray)));

There are a number of factors which make this code much more error prone and harder to write than the earlier version.\(^2\)

Firstly, there is no distinction in the code between the actual decisions and their consequences, which makes the model harder to understand and also makes the code more error prone. It should be the case that the number of intersections is completely determined by the choices made about whether to include words and where to put them. Unfortunately there is no simple way to verify that this is the case because the decisions, intermediate expressions, and final result are all represented by IntegerVariables, with constraints declared separately. In the procedural version it is straightforward to observe that the results are completely determined by the decisions and the original state of the problem.

Secondly, the overall approach used in the Choco version required significantly more thought, and the code is still much longer. An earlier attempt, which was more similar to the other version, failed miserably when it came to counting the number of intersections. While Java programmers are adept at writing code to calculate values based on other values, they typically have much less experience constructing declarative representations of relationships between variables.

Finally, the Choco library includes several classes with a multitude of methods, with a manual 214 pages long. Because the problem is represented using

\(^2\)Note that there may well be a simpler way to construct this model, and there is almost certainly a different model which would be solved more efficiently. The presented code is the best attempt of an intermediate Java programmer with limited CP experience and no prior exposure to Choco (myself early in my candidature) to create a straightforward correct model.
types from this library it is necessary to constantly look up the documentation for the various methods used to construct different types of variables and constraints. In contrast, the native version so far makes a total of three calls to library methods, all from the Decider class, and the code in the check and score methods does not refer to the library at all.

2.3.3 Integration

The final aspect to consider is how easy it is to trigger optimisation and then actually use the solution.

The use of the Solver class is demonstrated below. The Crossword class is the class implementing CSPopt, which contains the build, check and score methods defined previously. A Crossword is initialised with a list of potential words and limits on the height and width of the puzzle. Optimisation occurs within the buildMaximal method. After this call it is as though the build method defined above has been called with the best possible decisions made about which words to include and where to put them. The result is a list of Words and a list of list of Cells stored inside the Crossword object, representing a crossword with the maximum possible number of word intersections.

```java
Crossword crossword = new Crossword(wordStrings, width, height);
Optimiser opt = new Optimiser();
opt.buildMaximal(crossword);
crossword.display();
```

When displaying the crossword, we can use the Word and Cell classes directly. For example, we could iterate through the Word objects and assign clue numbers, recording these in the relevant Cell (the start location for the word) as well. We might choose to make the Cell class extend from the visual component JPanel, and then display the crossword as a grid of these Cell objects (with each Cell displaying itself using its assigned character and clue number). The Word objects could be displayable as well and used for the clue list.

In the Choco version, we would either use a similar arrangement for the GUI, in which case we would need to construct Cell and Word objects since we don’t already have them, or we might display the results directly from the Crossword class. In either case we need to explicitly retrieve the results from the solver. The code below uses the first approach, which is probably more robust and extensible (for example we may want to allow the user to make manual changes, and for that it would be useful to have a representation of each word and cell).

```java
// Obtain a solution
CPsolver s = new CPsolver();
s.read(m);
s.maximize(s.getVar(numIntersections), false);
```
for(int i = 0; i < numWords; i++)
{
    if(s.getVar(includeVars[i]).getVal() == 1)
    {
        // The word will be included
        Word w = new Word(words.get(i));
        if(s.getVar(directionVars[i]).getVal() == 1)
            w.direction = Direction.Horizontal;
        else
            w.direction = Direction.Vertical;
        w.startCol = s.getVar(startCols[i]).getVal();
        w.startRow = s.getVar(startRows[i]).getVal();
        includedWords.add(w);
    }
}

Notice the similarity between this code and that of the build and check methods defined above. Because we have been forced to represent the problem in a form suitable for the solver rather than the application, we end up writing code to construct a more meaningful representation of the solution after the problem is solved. The beauty of the native interface is that code constructing a useful representation of the solution can actually define the problem, and then the solution is automatically available in the required form.

2.3.4 Summary of advantages

The above example has hopefully illustrated some of the advantages of a native CP interface over a conventional library. Here we give a summary of all such advantages.

No CP-specific modelling

While using a standard CP library is certainly more convenient than using a separate language, the programmer still has to learn CP-specific modelling skills. With a native interface that is not the case. The programmer does not have to learn how to write a CP model, coming to grips with decision variables and
constraints. Unlike using a standard library it is not necessary to understand anything about how constraint solvers work.

Obviously the programmer still needs to choose a representation for solutions. This is modelling, but since the choice is based on the requirements of the application rather than the requirements of the solver, it is no different from the modelling already required to write the rest of the application.

**Concise documentation**

CP libraries typically require writing code which makes heavy use of library types representing decision variables and constraints. This is inconvenient for programmers who have to look up how to use each library-provided type or function. Indeed, studies on API design have shown that the difficulty of discovering the relationships between library types and the correct method of constructing each one is a key factor in the usability of an API [40, 105]. Even if clear and thorough documentation is provided, it can be very daunting. The Gecode [97] tutorial is more than 500 pages long. Even Numberjack [61], which is aimed more at beginners, has 36 classes with almost 70 documented members. In contrast, a native interface introduces a very small number of types and methods and should require very little documentation. This means programmers can quickly read the entire specification.

**Readable code**

The requirement to use special types representing decision variables also often necessitates special syntax or awkward code which obfuscates the problem definition, especially when dealing with mathematical expressions. While this can sometimes be avoided using operator overloading, with a native interface there is no need to introduce special types in the first place, so the code is readable even in languages such as Java which do not allow operator overloading.

**Reusable code**

When using a native interface, code forming part of the problem definition can be re-used, for example to evaluate current practice or user-provided solutions. The data structures used to represent the input data and solution can also be re-used. For example, there is no need to invent an integer representation for cards if the rest of the card playing application uses a structured type. The same structured type can be used directly in the candidate-building, checking and scoring code. Any required conversion between solver and host language types is performed automatically.

**Easier debugging and testing**

Native interfaces permit natural, straightforward debugging. Incorrect solver results indicate a bug in the building or checking code. An invalid solution
gives a failing test case for the checking code (it returns true for the given candidate when it should return false), while a missing solution indicates either that the building code cannot build that solution, or that the checking code erroneously returns false. Once one of these problems has been discovered, it can be corrected with no further reference to a constraint solver, using standard debugging techniques and tools (e.g. interactive debuggers or logging). Narrowing down which constraint is erroneous in a standard constraint model is often much harder (in part because fewer tools are available).

Similarly, the code defining the constraint problem can be tested in the same manner as the rest of the program, without using a constraint solver. Individual methods/classes/functions used in the definition may have their own unit tests. Even the decision-making code can be tested independently, using a Decider/DeciderState implementation making random or predetermined decisions.

**IDEs and compile-time checking**

A native interface allows advanced features of IDEs such as text-prediction, automatic re-factor, and live compilation to be directly applicable to the code defining the model. Standard static checking performed by the compiler can help prevent errors in the problem definition, with no need for error-prone string-based references to variables.

### 2.4 Related Work - Inspiration

This work was inspired by the language CoJava [26, 27]. CoJava is an extension to Java intended to allow programmers to use optimisation technology to solve a problem modelled as a simulation.

The simulation is written in Java with calls to CoJava library functions at various points. These functions can be used to choose a random value within a given range, to assert a condition which must hold, and to nominate a program variable as the optimisation objective. The program can be compiled and run normally, using what the authors call the simulation semantics. In this case there are multiple possible outcomes for the simulation corresponding to different random choices. Alternatively, the program can be compiled using the CoJava compiler, which transforms the simulation code into a constraint generation program, then runs this program, sending the resulting model to an external solver to identify a set of choices which satisfy the assertions and produce the optimal objective value. Finally the original code is recompiled with all random choices replaced by assignments to the values found by the solver. The result is what the authors call the optimisation semantics of the program. Compiling the code in this manner creates a simulation which appears to coincidentally make the best possible random choices.

An example CoJava program taken from [27] is shown in Figure 2.6. This is a basic optimisation problem where prices affect demand and then the combina-
tation of prices and demand determine profit. In the simulation semantics random prices are chosen and the profit is calculated based on these. In the optimisation semantics, the chosen prices will produce the greatest possible profit.

```java
public class ExampleSupplyChain {
    public static void main(String[] args) {
        double[] prices = new double[2];
        prices[0] = Nd.choice(100, 200);
        prices[1] = Nd.choice(105, 205);
        Demand demand = new Demand(prices);
        Manufacturer manufacturer = new Manufacturer(demand.quantities);
        Supplier supplier = new Supplier(manufacturer.materials);
        double revenue = 0;
        for (int i = 0; i < 2; i = i + 1) {
            revenue += demand.revenues[i];
        }
        double cost = manufacturer.cost + supplier.cost;
        double profit = revenue - cost;
        Nd.checkMaxObjective(profit);
        /* ... printing statements omitted here ... */
    }
}
```

Figure 2.6: An example CoJava program

### 2.4.1 Comparison with our proposal

The intention behind CoJava is essentially the same as ours; to improve access to optimisation technology by automatically converting simulation-style code into a conventional constraint model. However there are important differences in purpose, architecture and implementation, as discussed below.

**Interactivity**

A key difference between a native Java interface to CP and CoJava is that we aim to facilitate the embedding of optimisation functionality within interactive applications. The architecture of CoJava prohibits meaningful interaction with a user because the constraint solving happens at compile time. This means the programmer must be involved in solving every instance of a problem. In contrast, our intention is for the developer to produce interactive applications enabling non-technical end users to access optimisation technology independently.
Intuitiveness

We have also attempted to make the interface more intuitive and less confusing for programmers, in the following ways.

1. **Clearer semantics**  
   In our approach, the code written by the programmer is pure Java and does not take on any new semantics. The behaviour of decision methods is determined by the implementation of Decider, rather than the mode of compilation.

2. **Explicit optimisation**  
   In CoJava optimisation happens implicitly at compile time. We instead allow optimisation to be triggered explicitly as required during program execution, which is more consistent with a procedural mindset.

3. **Encapsulation**  
   Our architecture means that all code relating to a particular optimisation problem is encapsulated within a class implementing CSPopt. The use of an interface to define the requirements for an object to be optimisable fits well with the Java way of thinking, and this approach also makes including multiple separate optimisation problems in a single application straightforward.

**Supported code**

Finally, in the prototype described in the next chapter we dramatically expand on the type of code supported as part of the problem definition. In CoJava decisions were restricted to be made over real numbers only. This is unremarkable given their choice of mathematical programming solvers for the back end, and associated focus on continuous problems. We use a CP solver with a corresponding bias for discrete problems and therefore discrete choice methods. However, beyond simple integer decisions we also lift decisions to objects and even collections of objects. This means, for example, that you can make decisions about which object to choose from a number of options, or whether to include an object in a set, or in which order objects should appear in a list. All of these decisions can be made directly, with no need for the programmer to break them down into a numerical representation. It is also possible to iterate over collections which have been modified based on the outcome of some decision and therefore have unknown (but bounded) size. These additions present significant technical challenges and greatly improve usability.

**2.4.2 Extensions to CoJava**

CoJava was later extended but also narrowed in focus towards supply chain applications. Service Compositional CoJava (SC-CoJava) [22], uses the same semantics but includes a framework facilitating supply chain optimisation. Subsequently [4], specific library components relevant to strategic sourcing and
transportation were developed, along with a simplification to the SC-CoJava semantics.\(^3\)

The intention behind the provision of an extensive library of application-specific classes is to simplify the modelling of more complex problems. We propose an alternative approach, of introducing high level (but still generic) decisions. Consider the following supply chain example used to illustrate the use of SC-CoJava in [22].

**Problem** Deliver emergency packages (containing water, food, medicine, etc) to a set of emergency response locations. There are multiple possible suppliers for each individual product, several packaging locations, and a variety of different transportation services available to deliver packages to the emergency response locations. The objective may be to reduce the total cost or the time for delivery.

In this example, the SC-CoJava program used several library-provided classes to simulate the services, including two aggregate service classes `PackagingSerAgg` and `SupplierSerAgg`. These aggregate classes are used when demand can be met by a combination of services rather than selecting a single service. The `SupplierSerAgg` class, for example, chooses an order quantity for each supplier, and then sums the total amount ordered and constrains this to equal the amount ordered by the aggregate service.

Rather than introducing aggregate classes to combine simple decisions into more complex ones, our approach is to support the complex decision directly using a high level decision method. In this case, we would introduce a decision method which takes a total quantity and a set of options, and chooses how to distribute that quantity over the options.

```
Map<T, int> allocate(int totalquantity, Set<T> options);
```

This decision method could be used to decide how many packages should be constructed at each packaging location, and also how to spread the demand for a product across suppliers. Using this approach the user may have to write more code (e.g. to compute the total cost based on the cost of each service, which in SC-CoJava is also handled by the aggregators), but this is offset by the fact that they do not have to look up how to use a large number of provided classes. The main benefit is that the tool remains independent of the specific application. Complex decisions are discussed further in the next chapter.

**Related work and further extensions of CoJava**

The most recent work on CoJava incorporates some stochastic elements while remaining focused on supply chain optimisation [5]. We consider stochastic

\(^3\)The new semantics encapsulate the choices and assertions within the constructor of a class extending the abstract Service class. The constructor must also update a field holding the objective value, and a flag indicating whether this is to be minimised or maximised. When the optimised version of the simulation is run, the constructor of a Service class builds an optimal object of that class given the values of the constructor parameters.
problems to be beyond the scope of this thesis.

One piece of work related to CoJava is the tool URBiVA [77], which allows users to define constraint satisfaction problems using a C-like language. Any variable which is used before being defined is a decision variable, and constraints are declared using assert statements. The code is translated to bit-vector arithmetic and solved by an SMT solver. As with CoJava, the intention is to ease modelling. URBiVA is however much more restricted than CoJava in that only numeric and Boolean types are supported, and all branching conditions must be ground. That is, the decision variables cannot affect the execution path.

2.5 Other Related Work

There seems to be a long-standing consensus that ease of use, particularly with respect to modelling, is a challenge for CP [109, 48, 92, 49, 56]. Work addressing this challenge generally falls into one of the categories outlined in the survey paper [89]; model acquisition, model reformulation, synthesis of filtering algorithms, and automated solving. Our focus is on model acquisition, as well as the less frequently discussed consideration of integration with wider applications. In the sections below we summarise existing work related to these goals.

2.5.1 Special purpose constraint modelling languages

An obvious approach to facilitate model acquisition is to create a special purpose modelling language with constructs especially designed for the task of defining constraint problems. The main drawbacks of this approach are the potentially steep learning curve for users, and the difficulty of integrating optimisation into a wider program which must necessarily be developed using a separate language.

The AIMMS platform [20], which was originally designed for Mathematical Programming solvers but later extended to also support CP [111], addresses the integration problem by providing the entire toolchain. Within AIMMS, users can develop 'pages' to interact with the solver and display results graphically. The intention is that both rapid prototyping and the development of fully fledged industrial applications be achieved using AIMMS as the central tool. This does improve the integration problem, but seems inefficient in its overlap of functionality with conventional software development tools.

Control versus simplicity

Most CP modelling languages, such as OPL [108] and MiniZinc [86], are designed to address the requirement of advanced users to easily experiment with different models and solving strategies. This thesis is instead concerned with improving usability of CP for software developers who desire simplicity rather than control.

Several researchers have warned against attempting to 'package' CP too soon [112, 82]. The concern is that in our quest for simplicity we may box ourselves in and stifle further research. An often cited example (both as goal and as warning) is the field of Mathematical Programming, whose technology is considered
easier to use because of the relatively small number of modelling constructs and the focus on automated search. These advantages are also drawbacks, because information which could be useful to the solver is lost during the modelling process, and the user has limited control over the algorithms used to solve the problem. The flexibility and control afforded by constraint programming is one of its greatest advantages.

My view is that while we do not want to lose this advantage, there is room for another interface above the level of the one giving control. This non-expert interface should avoid dictating anything about the underlying solving process. The most obvious way to achieve this is to ensure that as far as possible its design is based on the requirements of the user rather than those of the solver (which may change over time), while attempting to retain rich information about the problem. Ideally, as new techniques become available in the underlying solver these should be automatically applicable to previously defined problems.

**Essence**

The discussion above resonates exactly with the philosophy of the problem specification language Essence [51]. The intention behind Essence is for problems to be defined at a very high level, before modelling decisions are made. The resulting specifications can be used to communicate between humans, or as the input for an automatic modelling tool. Essence aims to be accessible for discrete mathematicians, and as such uses notation from this field. The example below, taken from [51], is an Essence specification of the golomb ruler problem (problem 6 in CSPLib [58]).

```essence
language Essence 1.2.0
given n : int
where n ≥ 1
letting bound be 2**n
find Ticks : set {size n} of int {0..bound}
minimizing max(Ticks)
such that ∀pair1, pair2 : set {size 2} of int ⊆ Ticks . pair1 ≠ pair2 →
max(pair1) − min(pair1) ≠ max(pair2) − min(pair2)
```

The main components of the specification are statements declaring parameters and constraints on those (given, where), defining named expressions (letting), declaring decision variables (find), specifying the objective (minimising) and declaring constraints for decision variables (such that). The most notable feature of Essence is that decision variables can take nested combinatorial objects as their values, such as a set of sets of partitions, or a function from tuples to multisets. This means that often only a single decision variable is required, whose type is exactly that of the desired combinatorial object.

The high-level nature of Essence allows great flexibility in automating the transformation to a solver-level model. The tool Conjure [52, 3, 2] is able to automatically refine an Essence specification into many alternative concrete
constraint models. This leaves great scope for automatic selection of an effective model.

However, while the syntax and approach used in Essence is natural for mathematicians, it is not necessarily intuitive for software developers, and does not assist with the integration with a wider application. Native programming language interfaces do a much better job of this, and still satisfy the goal of being based on the user’s requirements rather than the solver, but they may fail to retain as much rich information about the problem. We can partially combat this problem through the use of high-level decision methods, and code-pattern detection, both of which are discussed in Chapter 3.

### 2.5.2 Re-purposing existing tools

Direct experience with constraint modelling is rare. However, many more people have experience using more generic modelling tools. By re-purposing these tools to support constraint/optimisation modelling, we can improve accessibility. Native programming language interfaces (and CoJava) attempt to re-purpose the modelling facilities provided by general purpose programming languages. Here we discuss several other modelling/querying tools which have been suggested as a basis for constraint modelling.

#### UML

Graphical modelling languages are a possibility which could appeal to software developers, assuming they use familiar notation. For example, constraint networks can be defined using an extension of the (graphical) Unified Modelling Language (UML) [114, 63]. The s-COMMA platform [31] also includes a graphical modelling tool which uses similar notation to UML. The advantage of a modelling tool based on UML is that UML already includes notation for basic constraints in the form of the OCL (Object Constraint Language) [113], and it is very well known amongst software developers (although more widely used in academia than in industry [90]). The drawback of a graphical modelling tool is that although diagrams may be used during software development, the conversion into an executable program is often manual. Therefore extra effort (either on the part of the programmer or the tool provider or both) will be required to support the use of graphically defined models within a wider program.

#### SQL

Other work has investigated the possibility of reusing relational database queries written in SQL (Structured Query Language) to express optimization problems [23]. Here optimization is described as an ‘inverse’ of reporting. While reporting evaluates the outcome of a set of decisions, optimization finds the decisions which would result in the best report. This is achieved through an extension to SQL called DGQL (Decision Guidance Query Language).
The additional features added to SQL by DGQL echo the features added to Java by CoJava. A DGQL program is a sequence of queries followed by an optimization statement specifying the objective to be minimized or maximized. Each query creates a new view which may be used in later queries. Some views are augmented with an extra column whose values are the equivalent of the random choices made in CoJava. It is also necessary to check conditions on views, in much the same way as conditions are asserted in CoJava.

When a DGQL script is run, the code is automatically translated into a model solvable by an external optimization engine. The results of the optimization are then inserted into the appropriate views as specified in the script.

Since constraints are declarative and relational in general, it is quite elegant to express them using a declarative relational database language. However, the lack of strong typing in SQL makes including complex business logic in database queries undesirable, as this code is relatively error prone and difficult to maintain.

**XQuery**

XQuery is a special purpose programming language designed by the World Wide Web Consortium (W3C). Its purpose is to facilitate the extraction and manipulation of data from XML (Extensible Markup Language) documents. Exactly the same approach taken to extend Java to CoJava, and SQL to DGQL, can by applied to extend XQuery to support optimisation [24]. The language DG-Query adds three simple annotations to XQuery. These are used to introduce nondeterministic attributes (whose values will be chosen by the solver), to declare XQuery variables as constraints (i.e. state that they must evaluate to ‘True’), and to nominate a particular XQuery variable as the objective. Standard XQuery defines the function between the nondeterministic attributes and the constraint and objective variables, and the desired output format. As with CoJava and DGQL, optimisation is performed via translation into a conventional constraint model which is sent to an external solver, after which the optimal attribute values are inserted before executing the XQuery program normally.

**2.5.3 Learning**

Rather than asking a user to define the problem directly, various techniques have been proposed to learn the model, as discussed below.

**Constraint acquisition**

The term *constraint acquisition* refers to learning a constraint network based on positive and negative examples (solutions and non-solutions). Usually the decision variables and their domains are defined upfront, and the task is to find a set of constraints on these variables which correctly classify the provided examples. Note that a given set of examples may be consistent with many different constraint networks. The usual goal is for sufficient examples to be
provided so that the learner converges on a single constraint network which is the only correct network in the target constraint language.

The simplest approach to constraint acquisition is passive learning, where it is assumed that a diverse training set is provided and the learner discovers a set of candidate constraint networks, reporting on whether or not convergence has been achieved. This is the approach used by Conacq [16]. The alternative approach is active learning, where the system asks targeted questions of the user. This adds a new requirement for the system to select appropriate questions in order to achieve convergence quickly. The most basic form of question is simply a request to classify a newly generated example as a solution or non-solution. This can greatly reduce the size of the training set required for convergence [18].

The basic assumption behind constraint acquisition is that it is easier for a user to distinguish between solutions and non-solutions than to provide a formal definition of the problem. This is similar to our assertion that it is easier for a programmer to write code to check a candidate solution than to construct a declarative model. Indeed, the ‘user’ answering membership queries may not necessarily be a person, but could just as well be a simulation (which is assumed to be easier to produce than the equivalent constraint model).

By asking other types of questions, it is possible to converge with fewer queries. For example, QuAcq [15] uses partial queries (asking the user to check if a given partial assignment of the variables is acceptable) to focus in on a single constraint explaining a given negative example, achieving convergence in a polynomial (but still potentially very large) number of queries. In a further improvement, g-QuAcq [14] adds generalisation queries, where the user is asked whether a given constraint can be generalised to other variables of the same type.

Actually, convergence is not absolutely necessary for a useful tool, and the extremely large number of examples or queries required to achieve this may be unwarranted in practice. For example, ModelSeeker [13] is able to provide the user with a ranked list of candidate partial models using only positive examples. For the highly structured problems considered, even a single positive example can be sufficient to provide useful suggestions for a novice modeller.

Interactive constraint solving

Closely related to active constraint acquisition is the idea of solving constraint problems interactively with a user who defines the constraints along the way. The MatchMaker system [50] is an early example of this, where the user supplies a constraint violated by each suggested solution. A more recent system Ask&Solve [17] allows problems to be solved without user-specified constraints. The user is repeatedly presented with a proposed solution to evaluate. For each rejected solution, the system uses the same technique as QuAcq to deduce a constraint based on further partial queries. As soon as a complete solution is accepted, the problem is solved. While constraint acquisition is usually limited to satisfaction problems, interactive constraint solving has also been applied to optimisation problems, for example in [57] where preferences were elicited.
The problem with this approach is that it requires extensive human involvement in the problem solving process for every instance. However, for problems which are inherently ill-defined, such as those involving human preferences, this may be unavoidable.

**Empirical model learning**

Even if the user is capable of constraint modelling, modelling certain problems can be extremely difficult. These problems typically involve systems whose behaviour is extremely complex, or for which an exact model is computationally intractable. In this case it is necessary to model an approximation of the true system behaviour.

Empirical model learning [75] is a framework for approaching these problems. The problem is modelled for the desired optimisation technique (e.g. CP, SMT, local search, or Mixed Integer Non-Linear Programming), leaving some part of the system as an undefined function. Then machine learning techniques are applied to learn an approximation of this function, producing a machine learning model (e.g. a neural network or a decision tree).

Finally the machine learning model is integrated into the constraint model in such a way that the solver can reason about its structure. For example, when embedding a neural network in a CP model (as in [9]) each neuron can be represented with a global propagator able to propagate bidirectionally. The integration of the machine learning model into the constraint model is a problem similar to our conversion from code into constraints.

The training data for learning may originate from experiments on (or past behaviour of) the real system, or be computed using a simulation. If learning from a simulation, then empirical model learning can be considered as a compromise between a CoJava style approach (which attempts to capture the simulation exactly) and simulation-based optimisation (which ignores the internal workings of the simulation completely).

**2.5.4 Simulation-based optimisation**

In previously discussed approaches, simulations were used as tools to assist in the development of a constraint model. In simulation-based optimisation [29, 53, 7], a simulation program is used as a substitute for a model. Heuristic techniques are used to search for input parameters resulting in a good outcome, treating the simulation as a black box. Typically, the simulation is stochastic in nature and represents a noisy estimate of the actual behaviour of the system, making it difficult to optimise analytically. Note that simulation-based optimisation is essentially interactive constraint solving (the simulation is the ‘user’), but with different techniques applied to the problem of selecting good queries.

An advantage of the simulation approach is that it is generally considered easier to create a simulation than to create an equivalent constraint model. Modern object-oriented languages are designed to allow a direct correspondence
between real world objects and program data. Sequences of events in the real
world are also easily reflected as sequences of program events. For these reasons,
most people find it much easier to write a simulation rather than a declarative
definition of the same problem. This advantage still applies for deterministic
problems, and simulation-based optimisation has been applied successfully to
these as well [73]. The intention behind CoJava is to retain this advantage while
harnessing the power and stronger guarantees of analytical optimisation.

Native interfaces also attempt to capture this benefit, but are not restricted
to stand-alone simulations. A stand-alone simulation evaluates a set of decisions
by mimicking some sort of process or behavioural interaction, outputting the
result. A native programming language interface to CP is equally applicable if
the simulation code is only part of a wider (possibly interactive) application, or
if the decisions are evaluated using a more direct method. For example, code
which evaluates an exam timetable may be written as a simulation of the exam
period, but it is more likely to simply iterate over students performing a check
for clashes. The latter code is not a simulation but is equally acceptable as
a definition of the constraints and objective. A native CP interface also goes
beyond stand-alone simulations by allowing the user to model the decision-
making process rather than just the operational process. That is, decisions can
be made on the basis of other decisions. This is not possible in a simulation
where all input parameters are provided upfront.

2.5.5 Combining programming paradigms

The problem of integrating constraint solving into a wider application is one of
combining disparate paradigms. We consider here various options for mixing
constraint programming with other programming paradigms.

Constraint logic programming

Modern constraint programming is the child of constraint logic programming
(CLPI 66). Due to the declarative nature of logic programming languages and
the use of backtracking-based algorithms to solve combinatorial problems, it is
quite natural to combine constraint solving with logic programming. Basically
the logic programming language is extended to allow constraints in the bodies
of clauses. During execution, as constraints are encountered they are added
to a constraint store. If at some point it can be detected that this store is
inconsistent, backtracking is triggered immediately.

The disadvantage of CLP is the lack of flexibility in the algorithm used to
solve the problem, as it is tied to the standard execution algorithm for logic
programs. A native interface to CP for a logic programming language would
overcome this problem by using the code only to define the model and not
to dictate a solving procedure. The entire model would be constructed and
sent off to a separate solver with complete freedom in its solving algorithm.
Actually an approach similar to this was taken in [94], where disjunctions in the
logic program were mapped to 0/1 auxiliary variables, allowing all constraints
to be posted at the root of the search tree. The purpose in this case was to support the integration of a MIP (Mixed Integer Programming) solver to improve the handling of linear constraints, without requiring that solver to accept new constraints incrementally.

An alternative approach is to make adjustments to the standard execution algorithm in order to improve its efficiency. For example, several different techniques have been proposed for extending the more efficient tabling execution strategy for logic programming to be applicable to CLP [33, 55]. The basic idea behind tabling is to avoid unnecessary re-computation by recording the results of calls.

**Constraint imperative programming**

Constraint imperative programming languages attempt to bring constraints into an imperative programming language. The desire to do this grew from interactive graphical applications where it is easier to state declarative constraints on the locations of items than to procedurally enforce these each time an item is moved. The difficulty is in controlling how automatic solving for values interacts with the rest of the program, as it is easy to produce unexpected and confusing behaviour.

Apart from programming style, the main difference between declarative and imperative programming languages is the type of variables used. Declarative languages typically use logical variables which only take one value through the life of the program. Basically a variable is a name for a (possibly unknown) value. Imperative languages typically use mutable variables, which can hold multiple values during their lifetime. Here a variable is a storage location. While a logical variable can be used before its value is decided (e.g. to build a larger term), a procedural variable is meaningless until it holds a concrete value. Any attempt to combine constraint and imperative programming must resolve this basic incompatibility. Several different approaches have been attempted, as discussed below.

One of the earliest constraint imperative programming languages, Kaleidoscope, originally used a refinement model for variables [47]. Essentially a variable had a stream of versions, and constraints were either persistent (applying to all versions of a variable) or transitory (applying only to a particular version). Assignment statements were transitory constraints setting the value of the current version of the variable. Later Kaleidoscope moved to a perturbation model, which was apparently more intuitive for programmers [76]. In this model a variable has a current value (which can be queried and used in the program), but as new constraints are added or other variables assigned new values, the solver can automatically change the variable to hold a new value. To control which variable is changed by the solver when a constraint is violated, each constraint is assigned a strength/priority, and the solver attempts to satisfy stronger constraints first.

Turtle [60] is very similar to later versions of Kaleidoscope, the main difference being that Turtle is not object oriented, instead having more of a functional
flavour. Turtle also uses a perturbation model for variables, but introduces a distinction between normal program variables and ‘constrainable’ variables, whose values the solver can adjust.

In both Kaliedoscope and Turtle, users can use special constraint declarations to define new constraints combining the primitive constraints provided. The Babelsberg object-constraint language framework [43, 44] supports re-use of ordinary methods as constraints. To make the behaviour more predictable, methods used in constraints must be pure (not having side-effects). To be used as a true constraint (which can be propagated in both directions) a method must consist of a single return statement. Babelsberg does not distinguish regular variables from constrainable variables, but does allow variables to be marked as ‘read-only’, to prevent their values being adjusted by the solver.

A constraint imperative programming language more closely related to native interfaces is Kaplan [72], which is an extension of Scala [87]. Scala is a multi-paradigm language combining imperative, functional, and object-oriented features. Since Scala already has features of multiple paradigms, it is relatively easy to support constraints as well. In Kaplan, a constraint is basically a lambda expression giving the function between its input arguments (the decisions) and a Boolean result. The constraint function itself is an object and has methods to find one or all solutions, where a solution is satisfactory input arguments. Complex constraints are formed using function composition, and it is also possible to apply an objective (in the form of another function) to control the order in which solutions will be returned. The constraints are solved via translation to SMT. This part of the language is essentially a native interface (to SMT) similar to the one proposed for Haskell earlier. The main difference is that the constraint itself has the solving functions, and the decisions are the inputs to the function, so their domains are defined by their type. Our decision-making functions allow for more specific domains (although obviously in Kaplan these constraints can be added afterwards). The constraints in Kaplan are restricted to a functional subset of Scala, so the translation to constraints they use would be applicable for a native interface in Haskell, but not in Java.

Kaplan also supports a constraint programming style with the addition of logical variables. These variables use the singular semantics, which means they represent exactly one value. A logical variable with unknown value can be passed around and constrained, but its value is fixed (to be consistent with the constraints) the first time it is queried and cannot subsequently be changed.

Another interesting case is JSetL [96, 95], which attempts to incorporate into Java not just constraints but declarative programming in general. Untyped logical variables, represented by objects with type LVar, and (possibly partially specified) lists and sets, can be constrained and unified in a style approximating logic programming. Constraints are explicitly added to a constraint store, and a solve method is used to convert the current set of constraints to a solved form (or generate an exception if the store is inconsistent). Solving is incremental, and it is also possible to ask for alternative solutions, which will update the values of the involved logical variables.

Although both native CP interfaces and constraint imperative programming
languages involve using constraint technology from within a general-purpose programming language, they actually serve completely opposite purposes. The intention of a constraint imperative language is to support a declarative programming style from within a procedural language, while a native interface attempts to provide access to solver functionality without shifting paradigms.

**Supporting isolated solver calls**

In the previous section we discussed languages allowing solver functionality and imperative code to be intermingled. Here we consider less tightly integrated approaches, designed to support isolated calls to a solver. This is the intention for native interfaces as well as conventional CP libraries.

Section 2.3 gave a detailed comparison between a native Java interface to CP and a conventional CP library for Java (Choco). CP/MP libraries have been created for many other languages as well (e.g [97, 61]). Such libraries typically provide an interface to a constraint solving engine using objects/classes in the host language to represent decision variables, constraints, the model, and the solver. Library functions are provided to create variables with appropriate domains, to add these to the model along with constraints ensuring that invalid combinations of values are disallowed, and finally to solve the model and extract the result. Although the programmer does not have to use a foreign language, it is still necessary to understand the concepts of decision variable and constraint, and to work directly with these to build a declarative representation of the problem.

Squander [83] supports constraint solving in Java using a separate constraint specification language based on first-order relational logic with transitive closure. The constraint problem is defined in an annotation in the Java code, and is allowed to make use of types declared elsewhere in the program. When constraint solving is triggered, the problem definition and the supplied parameters are automatically converted into the basic types supported by the solver. The solution is likewise translated back into the required structured types, and the Java heap is updated appropriately. This process is exactly the same as that proposed for a native interface, and should greatly improve the ease of integration. The difference is that the constraint specification is written in a separate language based on a different paradigm.

**Object oriented constraint languages**

The converse of constraint imperative programming, where constraints are added to imperative languages, is to bring features usually associated with imperative languages into a declarative constraint language. The most obvious example is the application of object-orientation to constraint languages [102]. A concrete example of an object oriented constraint language is the language COMMA [103] (the pre-cursor to s-COMMA, mentioned previously).

The main benefit is the inclusion of structured types, allowing data types to more closely resemble the real-world entities they represent. This should aid
modelling, but all of the usual disadvantages associated with special purpose modelling languages remain.

2.5.6 Language integrated query

A large part of the difficulty of incorporating constraint solving in a wider program is the need to convert between different representations for the same problem. This is not unique to constraint solving. Very similar issues are encountered when interacting with external data stores. For example, the data model and querying techniques used in a relational database do not sit naturally in an object oriented host language. This problem has been recognised and is referred to as the object-relational impedance mismatch [65]. We would be well served to pay attention to the techniques developed to support interaction with a relational database, as they are likely to be at least partly transferable to the problem of supporting interaction with a constraint solver.

One very successful approach is the LINQ (Language INtegrated Query) framework for .NET [81]. LINQ is a general framework for querying data which allows external sources to be treated the same as in-memory objects. LINQ itself provides some syntactic sugar for programmers, and facilities for back-end providers. Then each type of data source has its own back end, which is responsible for performing the necessary conversions between the object-oriented view presented to the programmer and the model used in the data source.

For example, LINQ to SQL allows programmers to query a relational database without leaving the object oriented paradigm. The mapping between object and relational data model is generated automatically, and the backend converts queries defined as lambda expressions into equivalent SQL. The results of these queries are then automatically converted back into the object model. The example below shows a (C#) LINQ query extracting the name of all people owning more than a given number of cars.

```csharp
int numcars = 2;

var peopleWithCars =
    from p in DB.Persons
    where p.Cars.count() >= numcars
    orderby p.LastName
    select p.FirstName + " " + p.LastName;

foreach (String n in peopleWithCars)
    System.Console.WriteLine(n);
```

The `from-where-select` syntax is converted into a method chain which can be passed to the back-end as a lambda expression to be transformed as required. In the example above the generated SQL query would need to join the Person table with the Car table and then use `groupBy` and `count` to find the number of cars owned by each person.
Just as in our case, some LINQ code cannot be translated into SQL, and sometimes it is necessary to write the query directly in order to make use of more advanced database functionality like temporary tables or locking hints. However, for the majority of queries, using LINQ to SQL is both more convenient and more robust compared with using SQL directly.

LINQ to CP

The LINQ framework could easily be used as the basis of a native interface to CP. The implementation would be similar to that used for Kaplan, as LINQ queries are also lambda expressions. Most of the required modelling facilities are already present. The `where` method can be used for constraints, and `orderby` combined with `First` can be used to perform optimisation (as is done in Kaplan). The least obvious aspect is the treatment of decisions. To remain consistent with LINQ (which is intended for querying collections) we would need each decision method to return an `ICollection` representing the domain of possible choices. It would obviously not need to actually populate this collection, but the semantics would be different from a normal native interface which assumes decision methods return a single answer.

See below for a theoretical example of LINQ to CP. Note that the lambda expressions passed to `sum` are an existing feature of C# commonly used in LINQ queries.

```csharp
Set<Item> items = ...;

var solutions =
    from subset in Decisions.chooseSubset(items)
    where subset.sum(item -> item.weight) <= maxweight
    orderby subset.sum(item -> item.value)
    select subset

Set<Item> bestsol = solutions.First();
```

In LINQ to SQL, local methods are not allowed to be used as part of the query. For a true native interface we would remove this restriction. The alternative is to provide a wider set of useful constraint/filtering methods understood by the back-end. The advantage of this approach is that it should be easier to generate good models. Actually, with an appropriate choice of decision and filtering methods we could produce a specification tool equivalent to Essence (note the similarity in syntax and style), but with much improved ease of integration. In fact, LINQ to CP could possibly be implemented via translation to Essence and subsequent use of Conjure.
2.6 Conclusion

This thesis is concerned with the usability of Constraint Programming (CP) tools, specifically for general software developers. The reason we have chosen to target general software developers is because they have the potential to pass on the benefits of CP technology to many more end users by incorporating optimisation functionality into application-specific software.

While expert users appreciate the ease of experimentation and high level of control afforded by modelling languages such as OPL [108] and MiniZinc [86], software developers aiming to incorporate basic optimisation functionality into an application do not have the expertise to benefit from this. Furthermore, it is highly inconvenient to have to learn and use a separate paradigm in order to implement a single application feature. There will always be difficult problems for which an expert is required to find a clever model and tailored solving technique which work together to allow solutions to be found efficiently. However, for the vast majority of practical problems which are not actually computationally difficult to solve, we should focus our efforts on reducing the human time and expertise required to access a suitable solver.

This thesis proposes an alternative interface to CP technology which hides from the user all reference to CP-specific concepts. The problem is defined within the host paradigm, by specifying the procedure or function used to combine individual decisions and compute the outcome. Automatic translation from this definition to a conventional CP model allows the problem to be solved using an external constraint solver. A solution giving optimal values for decision variables is then translated back into a structured representation.

Obviously the automatic conversion into a standard constraint model of reasonable quality is a significant technical challenge. Furthermore, the selection of solving strategies will also need to be automatic, as the user will not know which techniques are applicable, and some may be impossible to specify without referring to the final model. These challenges are the focus of the remainder of this thesis.

The interfaces and examples presented in this chapter are intended to be illustrative and inspiring and are theoretical only. The next chapter describes a proof-of-concept implementation of a slightly modified native CP interface for Java. This version of the interface has some concessions to practicality, and the prototype implementation does not cover all of the code used in examples here (e.g. String operations). The limitations of the prototype are discussed further in section 3.7, along with a discussion of the theoretical limitations for any native CP interface in Java.
Chapter 3

A Prototype for Java

3.1 Introduction

This thesis proposes an alternative interface to CP technology which allows an optimisation problem to be specified through code which constructs a solution using an oracle to make individual decisions, and code which evaluates that solution, determining whether or not it is valid and calculating a measure of its quality. This idea is applicable to any host programming language, although the precise design would obviously depend on the features of the language. We describe in this chapter a proof-of-concept implementation for Java, consisting of a Java library and a plugin for the Eclipse IDE.

Java has been chosen as the host language for two reasons. Firstly, it is a popular programming language for application development, and secondly, there are code analysis and transformation tools available to assist in the development of the required compile-time operation.

3.1.1 The library

The Java library supporting optimisation consists of two interfaces: Optimisable and ChoiceMaker, and a class: Solver. To access optimisation functionality the programmer must create a class implementing the Optimisable interface, writing a build and evaluate method. The build method asks the provided ChoiceMaker to make decisions, using the results to build a candidate solution. The evaluate method evaluates this candidate, throwing an exception if it is invalid, and otherwise computing and returning a measure of its overall value (or cost).

```java
interface Optimisable {
    void build(ChoiceMaker choiceMaker);
    int evaluate() throws Exception;
}
```

The ChoiceMaker is capable of making a variety of decisions, from the very simplest choice of a Boolean value, to complex decisions with many possible
outcomes, such as reordering a list of objects. When interpreting the decision making methods, the programmer may find a random ChoiceMaker easiest to understand; in this case the above-mentioned decision methods represent flipping a coin and shuffling the list respectively.

```java
interface ChoiceMaker {
    boolean chooseBool();
    int chooseInt(int min, int max);
    <T> T chooseOne(Collection<T> options); // choose one object of type T
    ...
}
```

Having implemented the Optimisable interface, the programmer can then use a Solver object to initiate optimisation. The Solver class provides methods to build a minimal, maximal or satisfactory version of an Optimisable object.

```java
class Solver {
    void buildSatisfactory(Optimisable o);
    void buildMaximal(Optimisable o);
    void buildMinimal(Optimisable o);
}
```

When optimisation is triggered by calling one of the Solver methods, the programmer imagines that the Solver takes the provided Optimisable object, and calls its build method, passing a special ChoiceMaker able to make perfect decisions.

```java
// imagined implementation of buildMinimal
void buildMinimal(Optimisable o) {
    o.build(new PerfectChoiceMaker(o, Min));
}
```

The result is that a subsequent call to evaluate will not cause any exceptions, and will compute the best (smallest in the case of buildMinimal) possible value.

---

**Comparison with previously presented native interface for Java**

There are three differences between the prototype we discuss here and the Java library described in the previous chapter.

1. We allow only discrete optimisation problems, by having evaluate return an int value rather than the more generic Comparable, and providing only discrete decision-making methods. This greatly reduces the implementation effort required, and covers typical CP problems.

2. The check method is removed. Instead of having a separate check method, if a solution is invalid the evaluate method should throw an exception. For satisfaction problems the evaluate method can return a fixed value. This often results in simpler code as it can be convenient to check validity at the same time as computing a score.
3. Several extra decision methods have been added. These operate at the object level, and aid modelling considerably.

### 3.1.2 Example of use

As an illustration of the system we consider a simple project planning application.\(^1\) Input to the program is a list of tasks, each of which may have dependencies indicating other tasks which must be scheduled at least a given number of days earlier. The application chooses a day to schedule each task so that all dependencies are satisfied and the project finishes as early as possible. The generated project plan is displayed to the user, who is then allowed to repeatedly reschedule the project after adjusting the tasks and dependencies.

In order to achieve this, the **Optimisable** interface is implemented by a class `ProjectPlan`, whose `build` and `evaluate` methods are shown below. A `ProjectPlan` is initialised with a reference to a list of tasks `alltasks`. The `build` method chooses a day for each task and passes this to the task to be recorded. The `evaluate` method first checks that all task dependencies are satisfied, and then calculates and returns the latest scheduled day as the value of the solution.

```java
void build(ChoiceMaker chooser) {
    for (Task task : alltasks) {
        int day = chooser.chooseInt(1, max);
        task.setScheduledDay(day);
    }
}

int evaluate() throws Exception {
    for (Task task : alltasks)
        if (!task.dependenciesSatisfied())
            throw new Exception();
    return getFinishDay();
}
```

An optimal project plan is obtained by passing a `ProjectPlan` object to the `buildMinimal` method of `Solver` (as we wish to find the solution with smallest evaluation result). After this method is called the tasks recorded in `tasklist` will have scheduled days corresponding to an optimal project plan. The application displays the plan using the `Task` objects, and is then free to make changes to the task list and dependencies (based on user input) before calling `buildMinimal` again, which will update the scheduled days to once again represent an optimal solution given the new tasks and dependencies.

```java
void main(String[] args) {
    // Read initial task list from args, file, database, etc
    List<Task> inittasks = ...;
    ProjectPlan pp = new ProjectPlan(inittasks);
    Solver s = new Solver();
    boolean finished = false;
    while (!finished) {
        s.buildMinimal(pp);
        pp.displayPlan();
        finished = pp.acceptTaskAdjustment(); // allow user to adjust tasks
    }
}
```

\(^1\)Full code for examples is available online: people.eng.unimelb.edu.au/pstuckey/optmodel/

47
3.2 Implementation

To solve the optimisation problem specified by the `build` and `evaluate` methods of an `Optimisable` object, an equivalent constraint model must be constructed and sent to a solver. As the complete problem specification depends on the program state when optimisation is requested (e.g. the tasks in `tasklist`), the constraint model cannot be constructed at compile time. Our approach is to generate at compile time a transformed version of the original code which can be used at run time to create a complete model based on the current state.

The compile-time operation, illustrated in Figure 3.1, is performed by a plugin for the Eclipse IDE. The plugin can be configured to perform the transformation step transparently during program compilation, or as an independent operation. Fortunately it is not necessary to create a transformed version of the entire program: only code used within the `build` and `evaluate` methods is relevant to optimisation. Transformed versions of the relevant methods are added to the project, and then this generated code is compiled along with the original source code using a regular Java compiler.

The code transformation is loosely based on the approach used for CoReJava [25]. The basic idea is to transform the code of the `build` and `evaluate` methods to work with symbolic expressions rather than concrete values. The original version of the code uses the results of decision methods to assign new values to fields and calculate a concrete objective value. The new version instead treats these decision results as unknowns, and builds from these an expression for the objective value and the new value of every potentially modified object property.

At run time, the solver looks up and executes this transformed code. The resulting expressions are decomposed into a collection of constrained solver variables, creating a model which is sent to an external constraint solver (currently G12 CPX [54]). The solution returned by the solver gives optimal decision values, which can be substituted back into the expressions for the new value of every possibly modified object property. By updating these properties accordingly, we give the illusion that the original `build` method has been called.

3.2.1 Compile time

The compile-time phase is a code-to-code transformation. The original code is still required, so we leave the original classes unchanged. For each relevant class
we add a new class containing transformed copies of the relevant methods and fields.

First, we use Eclipse JDT \[46\] to construct the abstract syntax tree for the entire project. At this point a lot of the difficult work, assigning types and linking method invocations with method declarations, is already done. This allows us to search for classes implementing the Optimisable interface, and then to follow chains of method invocations to find all relevant classes, methods, and fields.

- A method is relevant if it is a build or evaluate method, or if it is called from within a relevant method.
- A field is relevant if it is accessed within a relevant method.
- A class is relevant if it contains a relevant field or method.

The next phase is the actual translation phase. We support almost all of the basic language features included in Java 7 (the current version at the time this project started). Specifically, the expression types we allow are field and local variable access, literals, method invocations, class instance creation (constructor invocation), and infix, prefix and postfix expressions (excluding bit operations). Note that lambda expressions were not introduced until Java 8, and we have not considered them.

The supported statement types are while, for, and enhanced for loops; assignments; if-then-else statements; and branching statements return, (unlabelled) break, (unlabelled) continue, and throw (halt with an exception). The most notable omissions are switch and do-while statements. There is nothing inherently difficult about supporting these; it is simply a matter of limited time.

We also need to translate class, method, variable and field declarations. Simple uses of parameterized types (generics) and class inheritance are handled correctly (this is actually not difficult given the work already done by the compiler), but we have not given these aspects of the language much attention, and more complicated uses may cause compile-time errors.\(^2\)

We now explain the translation as applied to this subset of the language.\(^3\)

**Types**

In order for the new code to work with expressions rather than concrete values, it is necessary to change the types of all affected variables, including fields, local variables, and method parameters. We define a new class \texttt{Var} for this purpose. The \texttt{Var} class has a type parameter based on the original type of the variable. This is not strictly necessary, but it allows the compiler to perform better error checking on our generated code.

\(^2\)See section 3.7 for a more detailed discussion of the limitations of the prototype.

\(^3\)Note the code-to-code transformation we describe here is a later version than that described in the paper ‘Optimisation Modelling for Software Developers’, but the resulting code produces identical constraint models. The main difference is the use of instance rather than static classes in the transformed code.
class A {
    boolean field1;
}
class B {
    int method1(A arg1) {
        Boolean localVar1;
        ...
        return 5;
    }
}

(a)

class _A extends Expr {
    Var<_Bool> field1;
}
class _B extends Expr {
    _Int method1(Var<_A> arg1) {
        Var<_Bool> localVar1;
        ...
        return new _Int(5);
    }
}

(b)

Figure 3.2: Type conversion: (a) original code, and (b) transformed code.

We also introduce an abstract class Expr, to be used to represent expressions, including those to be stored in or retrieved from a variable. Each type used in the original code will have a corresponding class extending Expr, to be used to represent an expression of that type. We predefine expression versions of the following primitive types and basic collection types: Boolean, Integer, List, Set, and Map. Expression versions of user-defined classes are generated as part of the transformation. It is these subtypes of Expr which are used as the type parameter for a Var. As a simple naming convention, we prefix the type name with an underscore for the expression version.

Literal values in the original code do not have an explicit declaration of their type, but they also must be transformed to the corresponding expression type. To achieve this all literals are wrapped in a constructor for the appropriate (pre-provided) expression type.

See Figure 3.2 for an illustration of the type conversion. For this example we assume that all included code has been found to be relevant. In summary, the type conversion is performed as follows.

1. For each relevant user-defined class we create a new class to represent an expression of that type.
2. All variables (including fields, local variables and method parameters) are given type Var parameterized with the appropriate Expr type.
3. Methods return the appropriate Expr type.
4. Literal values are converted to (constant) expressions.

Assignments and variable references

The Var class has methods to assign a new value (assign), and to retrieve the current value (lookup). All assignment statements and variable references are replaced with calls to these methods. In both cases the ‘value’ will actually be an expression.
class A {
    int field1;
    boolean field2;
    boolean meth1(int arg1) {
        field1 = arg1;
        return field2;
    }
}

class _A {
    Var<_Int> field1;
    Var<_Bool> field2;
    _Bool meth1(Var<_Int> arg1) {
        field1.assign(arg1.lookup());
        return field2.lookup();
    }
}

Operators

In the original code, operators are used to produce a new value which is a function of the provided arguments. In the transformed version of the code the arguments are no longer concrete values, but Expr objects, and the result type must also extend Expr. We replace each use of an operator with a method building the appropriate expression, as demonstrated below for <. Note that the Less class extends _Bool. The supported operators are: &&, ||, !, >, >=, <, <=, *, /, %, +, - (binary and unary), ==, !=.

boolean meth1(int a, int b) {
    return a < b;
}

_Bool meth1(Var<_Int> a, Var<_Int> b) {
    return new Less(a.lookup(), b.lookup());
}

Method invocations

In Java all methods belong to a class, so since the types have all been adjusted already, a method invocation automatically becomes an invocation of the transformed version of the method. The method itself also takes care of converting the returned value to an Expr type.

However, it is necessary to adjust the arguments. As seen previously, the new parameters have type Var. This is necessary because the parameters of a method are variables. We must be able to assign to and reference these just like other variables. However, the arguments passed when invoking the method are expressions. They may be variable references, but could just as easily be the result of an operator or another method invocation. So it is necessary to wrap each argument in a Var constructor.

Constructor invocations must be adjusted as we wish to construct the expression version of the type rather than the original type. This involves simply adding an underscore to the class name. Otherwise they are treated no differently from other method invocations.

Calls to ChoiceMaker methods are also treated in the same way as other method invocations. However the transformed versions of these methods, which introduce one or more appropriately constrained unknowns, are pre-provided.
Control flow statements

In the ordinary execution of a piece of Java code, not every statement will be executed, and some statements will be executed more than once. Control flow statements determine how execution should flow through the program, based on values computed earlier. Our transformed code cannot skip statements, as it is intended to produce a model of all possible effects of the relevant code. Instead, we adjust the control flow statements to execute all reachable statements, while also recording the conditions under which normal execution would reach each statement. We call this set of conditions the path constraint.

As the simplest example of a control flow statement, consider an if-then-else statement. Usually only the then or else part would be executed. Our translation (illustrated below) instead executes both parts, with the path constraint adjusted appropriately for the duration of each part.

```java
int meth1(int y) {
    int x = 0;
    if(y > 5) {
        x = 1;
    } else {
        x = 2;
    }
    return x;
}

_Int meth1(Var<_Int> y) {
    Var<_Int> x = new Var<_Int>(new _Int(0));
    _Bool c1 = new Greater(y.lookup(),new _Int(5));
    if(c1) {
        x.assign(new _Int(1));
    } else {
        x.assign(new _Int(2));
    }
    return x.lookup();
}
```

Using the original code, the state of the program (the values stored in fields and local variables) after the if-then-else statement will depend on which part was executed. Therefore in the transformed version, the expressions representing this state (which are all obtained using the lookup method of Var) must also depend on the condition expression.
We achieve this by predicating all assignments on the current path constraint. If the current path constraint evaluates to false, that means the assignment statement is not executed, so the value stored in the variable should remain unchanged. If it evaluates to true then the new value replaces the old. This is handled within the Var class, as shown below.

```java
class Var<T extends Expr> {
    T currentVal;
    // constructor
    Var<T>(T init) {
        currentVal = init;
    }
    // assign a new value
    void assign(T newval) {
        currentVal = new IfThenElse(getPathConstraint(), newval, currentVal);
    }
    // get an expression for the current value
    T lookup() {
        return currentVal;
    }
}
```

In the example above, there are three assignments to the variable x; the initialisation assignment (with value 0), an assignment to 1 in the then part, and an assignment to 2 in the else part. Each of these replaces the expression stored in currentVal for x. The final value for x, returned by `meth1`, will be represented by the following expression.

\[
\text{IfThenElse}(c2, 2, \text{IfThenElse}(c1, 1, 0))
\]

where 
\[
c_1 = \text{Greater}(y, 5)
c_2 = \text{Not}(c1)
\]

Note that if the path constraint contains multiple conditions (e.g. inside a nested if statement) the conjunction is used.

**Throw**

In Java, a `throw` statement is used to raise an exception. Within the `build` and `evaluate` methods, a `throw` is to be interpreted as a constraint on the valid execution paths. That is, any execution path which reaches this point is invalid. Therefore each `throw` statement is simply converted into a method which posts a new constraint based on the negation of the current path constraint.

**Return**

A `return` statement provides an expression for the return value of a method, but it also indicates that the remainder of the method should not be executed. To correctly handle methods with multiple `return` statements, a new variable is introduced to hold the method result. At each `return` statement we assign the
returned value to this result variable. We also add a new condition to the path constraint for the remainder of the method ensuring that all further assignments are predicated on not having already returned. The condition added is the negation of the part of the current path constraint falling within the scope of the current method. That is, assuming \( P_m = c_0 \land \ldots \land c_k \) gives the conditions under which we execute the method, and \( P_r = c_0 \land \ldots \land c_k \land \ldots \land c_n \) gives the conditions under which we reach the return statement, the new condition is \( \neg(P_r \text{ given } P_m) = \neg c_{k+1} \lor \ldots \lor \neg c_n \).

Return handling is illustrated by the translation shown below (where \( v \) is a field of the current object).

```c
int exampleMeth() {

_Bool Pm = getPathConstraint();
_Var<_Int> retval;

if(v > 5) {
  _Bool ifcond = new Greater(v.lookup(), new _Int(5));
  addPathCond(ifcond);
  retval.assign(new _Int(5));
  _Bool Pr = getPathConstraint();
  _Bool noret = new Not(Pr.given(Pm));
  addPathCond(noret);
  removePathCond(ifcond);
  v = v + 1;
  v.assign(new Plus(v.lookup(), new _Int(1)));
  retval.assign(v);
  removePathCond(noret);
  return retval.lookup();
}
}
```

In the example above, the path constraint inside the if statement (\( Pr \)) is \( P_m \land \neg ifcond \). This is the condition used for any assignments in this scope, including the assignment to the result variable. It also defines the conditions under which this return statement will cause execution to jump out of the method. To reflect this, the remaining statements have a new condition \( noret = \neg ifcond \). The path constraint when we reach the assignment to \( v \) will therefore be \( P_m \land \neg ifcond \).

**Loops**

A loop whose exit condition does not depend on the results of ChoiceMaker methods can be left unchanged, because the number of iterations will be fixed at run time. The loop will achieve the desired goal of executing the transformed loop body once for each iteration. Loops whose exit conditions do depend on the results of ChoiceMaker methods are discussed later in Section 3.4.

The loop exit statements `break` and `continue` are supported using the same technique as return statements. A `break` statement adds a condition to the path constraint which is removed at the end of the loop. A `continue` statement adds a condition which is removed at the end of the current loop body.
3.2.2 Run time

At run time, optimisation is triggered by a call to the Solver method buildMinimal or buildMaximal, with an object implementing the Optimisable interface passed as an argument. The Solver first uses reflection techniques to find the transformed build and evaluate methods corresponding to the type of the received Optimisable object, and then executes these methods.

Executing this code produces an expression for the objective value (returned by the transformed evaluate method), plus expressions for the new values of each affected object property (recorded as currentVal in the corresponding Var object). We may also have some constraints originating from throw statements, each of which takes the form of a _BOOLEAN representing a negated path constraint. The next step is to convert these expressions into a constraint model.

Constructing the model

We choose to express the problem in MiniZinc [86] to allow flexibility in the choice of constraint solver. Only the objective and constraint expressions are necessary for constructing the model. The translation of these expressions into MiniZinc is mostly obvious. As mentioned previously, the translated versions of ChoiceMaker methods introduce constrained unknowns. These can be directly included in the model by declaring decision variables with appropriate domains and adding any required constraints. All other expressions are constants, or (possibly nested) applications of operators and conditional assignments. The supported operators all have equivalents in MiniZinc, which only leaves the ifThenElse expression type, for which we introduce a predicate varUpdate:

\[
\text{predicate varUpdate(var int: out, var bool: cond, var int: new, var int: old) = } \\
\text{( out = [old,new][bool2int(cond)] + 1 )};
\]

We introduce intermediate variables for subexpressions, sharing these where possible, and also compute initial domains based on the values of constants and the domains of core decisions. By default we declare a simple search strategy, assigning the minimum value for each core decision variable (those introduced by ChoiceMaker methods) in execution order.

Updating the program state

Upon obtaining a solution, the optimal decisions are substituted into the currentVal expression stored in each Var object, and reflection techniques are used to directly update the corresponding object properties accordingly (even if they are private). The complete run time process is illustrated in Figure 3.3.

Figure 3.3: The optimisation process, triggered on demand during execution.
3.2.3 Extended example

Returning to the project planning example from Section 3.1.2, consider the method `getFinishDay`, which is used to compute the return value for `evaluate`. The original and transformed versions of `getFinishDay` are shown in Figure 3.4.

```java
// original
int getFinishDay() {
    int finishDay = 1;
    for(Task task : alltasks)
        if(task.getScheduledDay() > finishDay)
            finishDay = task.getScheduledDay();
    return finishDay;
}

// transformed
_int getFinishDay() {
    Var<_Int> finishDay = new Var<_Int>(new _Int(1));
    for(_Task task : alltasks) {
        _Bool ifcond = new Greater(task.getScheduledDay(), finishDay.lookup());
        addPathCond(ifcond);
        finishDay.assign(task.getScheduledDay());
        removePathCond(ifcond);
    }
    return finishDay.lookup();
}
```

Figure 3.4: ProjectPlan `getFinishDay` method, called from within `evaluate`.

When optimisation is triggered at run time, we execute the transformed versions of `build` and `evaluate`, which will in turn call the transformed version of `getFinishDay`. Recall that the scheduled day for each task is assigned in the `build` method using `chooseInt`, so the expression for each scheduled day is not fixed. In the original code for `getFinishDay`, we check the scheduled day of each task against the current value of `finishDay`, and update `finishDay` only if this task has been assigned a later day. In the transformed code we call `assign` for every task, each time creating a new expression for the current value of `finishDay`. This is an `IfThenElse` expression which is equal to the assigned expression (this task’s scheduled day) if the path constraint (the greater-than comparison) evaluates to `true`, and otherwise the previous value. The expression finally returned by `getLatestDay` (our objective expression) is a chain of nested `IfThenElse` expressions.

When converting this objective expression into MiniZinc we introduce an intermediate variable for the result of each `IfThenElse`, and constrain it using the `varUpdate` predicate. An example MiniZinc model for a project with 3 tasks is shown in Figure 3.5. The three `day` variables in this model are core decision variables introduced by calls to `chooseInt`. The `finishDay` variables are the intermediate variables used to represent the value stored in the Java `finishDay`
variable. The final value (finishDay35) is constrained via a chain of varUpdate constraints to equal the greatest assigned day. The other two constraints enforce task dependencies. These originate from the throw statement in the evaluate method, each being the negation of the path constraint when this statement was reached in the transformed version of evaluate.

```plaintext
var 1..6: day0;
var 1..6: day1;
var 1..6: day2;
var {1,2,3,4,5,6}: finishDay31;
var {1,2,3,4,5,6}: finishDay33;
var {1,2,3,4,5,6}: finishDay35;
constraint (not (day1 < day0));
constraint (not (day2 < (1 + day1))) \ (not (day2 < (2 + day0)));
constraint varUpdate(finishDay31, (day0 > 1), day0, 1);
constraint varUpdate(finishDay33, (day1 > finishDay31), day1, finishDay31);
constraint varUpdate(finishDay35, (day2 > finishDay33), day2, finishDay33);
solve :: int_search([day0, day1, day2], input_order, indomain_split, complete)
  minimize finishDay35;
```

Figure 3.5: MiniZinc model for project planning example.

After generating the model, we find a solution using an external solver, and then update the program state as required. In this case the values for day0, day1 and day2 are used to set the scheduled day field of the three tasks. It appears to the programmer as though the original build method has been called with a ChoiceMaker able to choose an optimal day for each task. Execution can then continue as normal.

### 3.2.4 Refining the translation

Many values computed within the build and evaluate methods are unaffected by the results of decisions. Measures to take this into account can reduce unnecessary complexity in the model. At compile time, translation is only required for a method if at some time it is passed a non-constant argument, or if the code within the method uses decision procedures provided by the ChoiceMaker, changes the state of some object, or reads a field which is updated elsewhere in translated code. Within a method, a variable that is never assigned to a value that depends on the outcome of decisions, and is never assigned conditionally depending on the outcome of a decision, does not need its type changed.

At run time, if the condition for a branching statement is constant, only the corresponding branch is executed. Also, if the current path constraint is constant, an assignment can overwrite the previous value of a variable unconditionally. Actually, it is only required that the part of the path constraint which falls within the variable’s declaration scope is constant. If the part of the path constraint outside this scope evaluates to false, then normal execution would
never reach the declaration of the variable, making its value irrelevant. As an example, consider the code below.

```java
if(x) {
    int sum = 0;
    for(int item : list) {
        sum = sum + item;
    }
    s = sum;
}
```

Let $x$ be the Boolean expression giving the value of the variable $x$. This condition $x$ is added to the path constraint for the duration of the then block. The assignment to $s$ (a variable declared outside the if statement) is conditional on $x$, but all assignments to $sum$ are unconditional.

This improvement is achieved via a minor adjustment in the `Var` class. The `Var` constructor records the path constraint in effect when the variable is declared ($P_d$), and then instead of using the full path constraint, the `assign` method excludes any conditions in $P_d$.

### 3.3 Object Variables

For real-world problems, natural code will almost always involve decisions at the object level. We describe here our support for this, including a decision method to choose one object from a collection. Note that this is a major extension over CoJava, which restricts non-determinism to primitive typed variables only.

Let us consider a new version of the project planning application. This time, resources are no longer infinite. Instead only a given number of hours are available on each day. We introduce a `Day` class, with fields recording the day number and the maximum number of hours available, as well as the number of hours currently assigned. Each `Task` now also has a duration, and we need to ensure that the total duration of all tasks assigned to a day does not exceed the hours available.

The new `build` method for `ProjectPlan` is shown below. For each task a `Day` object is chosen, and the task is assigned to this `Day` using the `addTask` method. This method updates the hours assigned to the day, returning `false` if the total is now greater than the available hours. If the return value is `false` an exception is thrown to indicate that the solution is not acceptable.

```java
public void build(ChoiceMaker chooser) throws Exception {
    for(Task task : allTasks) {
        Day chosenDay = chooser.chooseFrom(allDays);
        if(!chosenDay.addTask(task))
            throw new Exception("Failed to add task");
    }
}
```

The object represented by `chosenDay` depends on the outcome of the `chooseFrom` decision. This means that in the transformed version of the code we need to be able to call `addTask` without knowing which `Day` object is the target.
In general, we wish to allow variables with an object (not primitive) type to take a value which depends on the decisions, and we wish to be able to use these variables as the target for field references and assignments as well as method invocations. The following sections describe how this is achieved, using the short code snippet below as an illustrative example (this code uses a simplified \texttt{Day} class, shown in Figure 3.6).

```java
1 Day d = chooser.chooseFrom(days);
2 int oldnum = d.numhours;
3 d.numhours = 2;
4 d.setNumHours(2);
```

### 3.3.1 Modelling object decisions

To support this sort of code, we first need to be able to represent the choice of an object using primitive solver variables. This is achieved by assigning an integer key to each distinct object. An expression with a non-primitive type has a domain of possible objects, which can be translated into a corresponding integer domain. This representation allows straightforward equality comparisons between object expressions. The \texttt{chooseFrom} method simply creates a new expression whose domain is given by the provided collection, with a corresponding integer decision variable.

**Example**  
Our first line of code will be translated as follows.

```
Day d = chooser.chooseFrom(days);
```

↓

```
Var<_Day> d = new Var<_Day>(chooser.chooseFrom(days));
```

Let us assume that the \texttt{days} collection contains three concrete \texttt{Day} objects \(D_1\), \(D_2\) and \(D_3\), which have been assigned keys 1, 2 and 3 respectively. The call \texttt{chooser.chooseFrom(days)} creates a new decision variable \(x\) with domain \(\{1, 2, 3\}\), and returns a \texttt{Day} with equivalent domain \(\{D_1, D_2, D_3\}\). This \texttt{Day} object is then assigned as the initial value for \(d\).

### 3.3.2 Field references

The next consideration is field access. As with single objects, each relevant field of an object expression is assigned a \texttt{Var} object. This \texttt{Var} must be able to produce an expression for the current value of the field, and to accept an expression for a newly assigned value. For object expressions, a special type of \texttt{Var} is used which has a reference to the \texttt{Var} representing this field for each concrete object possibly represented by the expression. It also maintains an integer expression for the index of the chosen object. When an expression for the current value is requested, an intermediate solver variable is created and constrained to equal the \texttt{currentVal} expression stored in the \texttt{Var} corresponding to the chosen object. This is a simple element constraint.
Example  Consider line 2 of our example code, translated as shown below.

\[
\text{int oldnum = d.numhours;}
\]

\[
\text{Var<\_Int> oldnum = new Var<\_Int>(d.lookup().numhours.lookup());}
\]

The call \(d.lookup()\) returns the current \(\_\text{Day}\) expression assigned to \(d\). This is simply the \(\_\text{Day}\) created by \(\text{chooseFrom}\) on the previous line. We then retrieve the \(\text{numhours}\) field of this \(\_\text{Day}\), which is a \(\text{Var<\_Int>}\) (see Figure 3.6b). This \(\text{Var}\) object, created on construction of the \(\_\text{Day}\), is of the special type described above. It has a reference to the \(\text{numhours}\) \(\text{Var}\) for \(D_1\), \(D_2\) and \(D_3\), and to the \(x\) variable which indicates the chosen day. When we subsequently call \(\text{lookup}\) on this \(\text{Var}\), it in turn calls \(\text{lookup}\) on each of its stored \(\text{Var}\) objects, obtaining for each concrete day \(D_i\) an expression \(hrs_i\) giving the current value of \(\text{numhours}\). It then creates and returns a new \(\_\text{Int}\) object with value \(y\), constrained as follows.

\[
\text{constraint } y = [hrs_1, hrs_2, hrs_3][x]
\]

This new \(\_\text{Int}\) with value \(y\) is then assigned as the initial value for \(\text{oldnum}\).

### 3.3.3 Field assignments

A field assignment for an object expression must update the corresponding field for every object in the expression’s domain. A new intermediate variable is created for each, and a new MiniZinc predicate \(\text{fieldUpdate}\) is used to ensure that all except the one at the correct index are equal to the previous values, while the one at the correct index is equal to the assigned expression if the current path constraint holds, or the old value otherwise.

Example  Line 3 of our example code is translated as follows.

\[
d\text{.numhours = 2;}
\]

\[
d\text{.lookup().numhours.assign(new \_Int(2));}
\]

The call \(d\text{.lookup().numhours}\) retrieves the same special \(\text{Var}\) used for a lookup in the previous line. This time we call \(\text{assign}\), passing an expression for the new value. Inside \(\text{assign}\), the current value stored in the \(\text{Var}\) associated with each concrete \(\_\text{Day}\) \(D_i\) is replaced with a new expression \(\text{newhrs}_i\). These new expressions are then constrained as follows (where \(P\) is the current path constraint).

\[
\text{constraint fieldUpdate(2, x, P,}
\]

\[
[hrs_1, hrs_2, hrs_3, [newhrs_1, newhrs_2, newhrs_3]];
\]

\[
\text{predicate fieldUpdate(var int: assignedVal, var int: index, var bool: cond,}
\]

\[
\text{array[int] of var int: oldVals, array[int] of var int: newVals) =}
\]

\[
\text{forall(i in index.set(oldVals)) (}
\]

\[
\text{newVals[i] = [oldVals[i], assignedVal][bool2int(cond \&\& index=i) + 1]}
\]

\[
)};
\]
3.3.4 Method invocations

We also need to handle the calling of methods on object expressions. Fortunately, the only way the target object affects the outcome of a method invocation is through the values stored in its fields. This means that all uncertainty can be pushed down to the field level. Since a non-constant object expression already has a Var object for each field of the special type discussed above, no further changes are required.

Example  Consider line 4 of our example, with the following translation.

\[
\text{d.setNumHours}(2); \quad \downarrow \quad \text{d.lookup().setNumHours(new Var<\_Int>(new \_Int(2))));}
\]

As before \text{d.lookup()} returns our \text{Day} object, upon which we call \text{setNumHours}. Inside the \text{setNumHours} method (see Figure 3.6), we have an assignment to the \text{numhours} field. As intended, this assignment has the same effect as the direct assignment to \text{numhours} on line 3 of our example code, as it uses exactly the same Var object.

3.4 Variable Collections

Combinatorial problems commonly involve collections such as sets or lists. It is therefore valuable to allow the use of these in the Optimisable code. It should be possible not only to store variable objects in collections, but also to make arbitrary changes to the collection in variable contexts, so that the resulting size and composition of the collection depends on the outcome of decisions. Furthermore, it should be possible to iterate over these variable collections.

Returning to the project planning example, imagine we are now allowed to hire an external contractor to perform some tasks, paying an hourly rate plus a callout fee for each day the contractor’s services are required. Instead of finishing as early as possible we wish to minimise cost while meeting a deadline. Consider the following extracts from the revised \text{build} and \text{evaluate} methods.
Day day = chooser.chooseFrom(days);
if (chooser.chooseBool()) {
    contractedTasks.add(task);
    contractorDays.add(day);
}

int cost = dayFee * contractorDays.size();
for (Task t : contractedTasks) {
    cost += hourlyRate * t.duration();
}
return cost;

Note first that the chosen day for the task, a decision variable, is added to the contractorDays set. Second, this set and the list of contracted tasks are both updated conditionally depending on whether or not this task is to be contracted out (as decided by the chooseBool method). The crucial aspect here is that the for loop in the evaluate code iterates over a collection whose size depends on the values of decision variables.

We have implemented a special purpose translation for the Set, List and Map interfaces in order to support this kind of code. The Set, List and Map classes provide specialised transformed versions of (almost) all methods included in these collection interfaces, using a special-purpose representation for the state of the collection.

3.4.1 Internal representation for sets, lists, and maps

The Set class represents a set using a list of possible members of the set, and a corresponding list of Bool expressions indicating whether or not each item is actually in the set. Each possible item is also an expression which may represent a choice between several actual objects. When a Set is initialised, all items are constant expressions, and the Boolean conditions are true. It is only through operations on the set that variability is introduced.

As an example of an operation we consider add. This method is supposed to add the given item to the set if it is not already present, and return true if the set has changed. The pseudo-code below defines the effect of calling add on a set s having current state \( v_s = \langle n, x_1..n, c_1..n \rangle \), where \( n \) is the number of possible members of the set, \( x_i \) is the possible member at index \( i \), and \( c_i \) is the Boolean condition indicating whether or not \( x_i \) is actually in the set.

\[
result = s.\text{add}(y);
\]

\[
a := \bigwedge_{i=1..n} \neg(x_i = y \land c_i) \\
b := P \land a \quad / / P: \text{path constraint} \\
v_s := \langle n + 1, \langle x_1..n, y \rangle, \langle c_1..n, b \rangle \rangle \\
result := a
\]

The Boolean expression \( a \) represents the condition that \( y \) was not already in the set. This expression is also used as the return value. It is possible for the set to remain unchanged even if \( y \) was not already present, as the current path constraint may not be satisfied. However, in this case execution would not reach this method call, so the return value is irrelevant.

Clearly it is important to handle constants well, as otherwise the model becomes unnecessarily complex. For example, the add method does not add a
new possible item if one of the existing possible items is identical to the added item. In this case the existing item’s Boolean condition is updated instead, and if this is already constant and true, no change is required.

The List class maintains a list of possible members in the same fashion as Set, but instead of a Boolean expression indicating whether or not the item is present, an integer expression for each possible item indicates its (0-based) index in the list, with all items not actually in the list having indices greater than the length of the list. A separate integer expression is maintained for the current length of the list.

The Map class is implemented as an extension of the Set class, with an added expression for each possible key giving its currently assigned value.

### 3.4.2 Iteration

Iteration in Java is performed using the Iterator interface, with two methods: hasNext to check whether there are remaining items, and next to retrieve the next item. We support iteration over variable collections using a VariableIterator class which implements the transformed versions of these operations. That is, the hasNext method returns a Bool expression which evaluates to true if at least one of the remaining items is actually in the collection, while the next method returns an expression for the next item.

Enhanced for loops (for example the one iterating over contractedTasks in our example) are converted into the equivalent while loop using an explicit iterator. At the beginning of each loop iteration the hasNext method is called, and the resulting Boolean expression is added to the path constraint, to be removed at the end of the loop body. When the hasNext method returns an expression which is constant and false (as it does when it runs out of possible members of the collection), the loop is terminated.

Although the Set state includes a list of expressions for possible members of the set, the iterator cannot simply return these in order. To correctly reflect loop exit logic all items which are actually in the set must be returned before any which are not. For this reason, each item returned by the VariableIterator is actually a new expression which may represent any of the possible members of the set. An integer variable is created for each returned item, giving the corresponding index into the list of possible members. These indices are constrained to ensure that an item which is not in the set is never returned before an item which is in the set, and further (to avoid symmetry) that within these two groups items are returned in order of index.

---

4Actually, it is sufficient to ensure that any iteration using an item not actually in the set has no effect. In later implementations the iterator returns (in an arbitrary order) each concrete value which may be included in the set. Instead of exiting the loop as soon as it is determined that the maximum number of iterations has been reached, a contains query is used as an additional path constraint for each individual execution of the loop body, to ensure that if the item is not actually present then this iteration will have no effect.
Example  Consider the loop iterating over a variable set in the above example. This loop is translated as follows.

```java
for(Task t : contractedTasks) {
    cost += hourlyRate * t.duration();
}
```

```
VariableIterator it = new VariableIterator(contractedTasks);
while(true) {
    _Bool cond = it.hasNext();
    if(cond.isFalse())
        break;
    addPathConstraint(cond);
    Var<_Task> t = new Var<_Task>(it.next());
    cost.assign(new Plus(cost.lookup(),
        new Mult(hourlyRate.lookup(), t.lookup().duration())));
    removePathConstraint(cond);
}
```

Let us assume the `contractedTasks` set has \( n \) possible members with values given by the expressions \( m_1..m_n \) and membership conditions \( c_1..c_n \). The iterator \( it \) consecutively returns \( n \) `_Task` objects with values \( t_1..t_n \). Each expression \( t_i \) is assigned an index variable \( ind_i \) to indicate which member it represents. The following constraints are added to the model.

```plaintext
var int: size; % expression for size of set
array[int] of var int: members = [m_1, m_2, .., m_n];
array[int] of var bool: conds = [c_1, c_2, .., c_n];
array[int] of var int: itemsreturned = [t_1, t_2, .., t_n];
array[int] of var int: indices = [ind_1, ind_2, .., ind_n];

% each returned item takes a value according to its index variable
constraint forall(i in 1..n) (itemsreturned[i] = members[indices[i]]);

% items with true condition occur before those with false condition,
% and the number with true condition is given by size
constraint forall(i in 1..n) (conds[indices[i]] = (i <= size));

% symmetry breaking: consecutive items are in order,
% except for the changeover between those in and not in the set
constraint forall(i in 1..n-1) ((size != i) -> (indices[i] < indices[i+1]));
```
Obviously constant detection is very important to avoid excessive complexity. We have implemented some simplifications, such as returning all items which are definitely in the set first, and excluding entirely any which are definitely not in the set, but further improvements are possible.

Iteration over lists is implemented similarly except that the order in which items are returned is determined by the indices stored as part of the List state.

For both lists and sets, the Boolean expression returned by hasNext is simply a comparison between the number of items already returned and an expression for the size of the collection. When eventually we have returned a number of items equal to the maximum size of the collection, this expression will be constant and false.

3.5 Complex Decisions

With support for variable collections, it becomes possible to provide more complex decision procedures allowing decisions to be specified at a higher level. As an illustration, let us return to the project planning example. Imagine that instead of choosing a single worker for each task, we assign a team of workers. For each task there are a set of allowed team sizes, and the task duration varies according to the size chosen. Furthermore, tasks may be performed across multiple days, as long as an integral number of hours is assigned to each day and the total number of hours matches the task duration.

Below is a build method appropriate for this situation, making use of complex decision procedures.

```java
public void build(ChoiceMaker chooser) {
    for (Task task : allTasks) {
        int teamSize = chooser.chooseFrom(task.allowedTeamSizes());
        int taskDuration = task.getDuration(teamSize);
        Set<Worker> team = chooser.chooseSubset(allWorkers, teamSize);
        Map<Day, Integer> chosenDays = chooser.allocate(taskDuration, allDays);
        for (Worker worker : team)
            worker.assignTask(task, chosenDays);
    }
}
```

The procedure first chooses a team size and a corresponding team. Then the total task duration is allocated to days using the allocate method. This method decides how much of the given quantity (in this case the task duration) should be allocated to each object in the given collection (in this case the list of days). Finally, the workers’ schedules are updated appropriately.

Not only do complex decision procedures greatly simplify the build method, they also provide opportunities to make use of global constraints. For example, the allocate method can make use of a global sum constraint to ensure that the total quantity allocated is correct.
Implementation example

The following pseudo-code outlines the implementation of the `allocate` method.

```java
allocate(quantity, items) {
    m = new _Map();
    foreach(concreteItem in items.allPossibleConcreteMembers()) {
        q = newIntVar(0, max(quantity)); // args define range domain
        allocatedQuantities.add(q);
        addConstraint(q <= quantity);
        addConstraint((q > 0) -> items.contains(concreteItem));
        m.addPossibleKey(concreteItem, q, q > 0); // key, value, condition
    }
    addConstraint(quantity = sum(allocatedQuantities));
    return m;
}
```

In general, a complex decision method creates a number of new variables, adds appropriate constraints, and then sets up a convenient data structure whose state is based on the newly created variables, and returns this to be manipulated further in user-written code.

3.6 Experimental Results

Having developed a working system, our next concern is performance. We present in Table 3.1 preliminary experimental results demonstrating the relative performance of the system compared with equivalent hand-written models. It is unrealistic to expect that automatically generated models will be able to compete with models produced by an expert. Our aim is to be able to handle problems arising for small businesses or individuals. The results show that we still have some work to do to achieve this goal. Note that compilation time is not shown as this was insignificant: the entire suite compiles in 20 seconds (with around 15 seconds spent performing code transformation).

For most problems the vast majority of the total time is spent solving the model (rather than generating it). This suggests the potential to greatly decrease the running time by improving the model. Our initial analysis of the automatically generated models led to the identification of two easily detected programming patterns for which stronger constraints are available. The table below shows the original Java code for each of these patterns, the constraints which would be generated using the standard transformation, and the alternative stronger constraints.

<table>
<thead>
<tr>
<th>Java code</th>
<th>Standard translation</th>
<th>Special translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(c) x++;</td>
<td>var x' = [x, x+1][bool2int(c)+1];</td>
<td>var x' = x + bool2int(c);</td>
</tr>
<tr>
<td>if(a&gt;x) x = a;</td>
<td>var x' = [x, a][bool2int(a &gt; x)+1];</td>
<td>var x' = max(x, a);</td>
</tr>
</tbody>
</table>
After adding a step during the model generation phase to automatically detect just these two programming patterns and replace the constraints, we observed a significant improvement in performance on several benchmarks. As will be discussed in the following chapters, further gains can be made by identifying other patterns for which straightforward model refinements are beneficial. We have also achieved significant performance improvements through more substantial changes to the generated constraints and improved representations for collection operations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Size</th>
<th>Total</th>
<th>Solving</th>
<th>Improved</th>
<th>Hand</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>project planning 1</td>
<td>250 tasks</td>
<td>1.13</td>
<td>78.1%</td>
<td>0.69 (39%)</td>
<td>0.16</td>
<td>4.4</td>
</tr>
<tr>
<td>project planning 2</td>
<td>18 tasks</td>
<td>16.22</td>
<td>97.6%</td>
<td>13.31 (18%)</td>
<td>1.18</td>
<td>11.3</td>
</tr>
<tr>
<td>project planning 3</td>
<td>14 tasks</td>
<td>32.38</td>
<td>96.6%</td>
<td>26.38 (19%)</td>
<td>0.27</td>
<td>96.9</td>
</tr>
<tr>
<td>bin packing</td>
<td>8 items</td>
<td>8.67</td>
<td>98.2%</td>
<td>5.81 (33%)</td>
<td>0.34</td>
<td>17.2</td>
</tr>
<tr>
<td>golomb ruler</td>
<td>7 ticks</td>
<td>3.45</td>
<td>33.6%</td>
<td>3.41 (0.9%)</td>
<td>2.03</td>
<td>1.7</td>
</tr>
<tr>
<td>knapsack (0-1)</td>
<td>30 items</td>
<td>1.97</td>
<td>80.4%</td>
<td>1.95 (0.9%)</td>
<td>0.15</td>
<td>13.2</td>
</tr>
<tr>
<td>knapsack (bounded)</td>
<td>30 items</td>
<td>6.40</td>
<td>95.2%</td>
<td>6.36 (0.6%)</td>
<td>1.10</td>
<td>5.8</td>
</tr>
<tr>
<td>routing (pickup-del)</td>
<td>8 stops</td>
<td>6.48</td>
<td>96.5%</td>
<td>6.45 (0.6%)</td>
<td>0.33</td>
<td>19.7</td>
</tr>
<tr>
<td>social golfers</td>
<td>9 golfers</td>
<td>2.65</td>
<td>75.0%</td>
<td>2.60 (1.9%)</td>
<td>0.14</td>
<td>18.3</td>
</tr>
<tr>
<td>talent scheduling</td>
<td>8 scenes</td>
<td>39.47</td>
<td>99.7%</td>
<td>20.87 (47%)</td>
<td>2.20</td>
<td>9.5</td>
</tr>
</tbody>
</table>

Table 3.1: Experimental results for project planning example and various well-known problems. In order the figures give the total time (secs) for the optimisation step, the percentage of this total used by the solver, the new total time using basic model refinement with percentage improvement in brackets, the solving time for an equivalent hand-written model, and the number of times faster this hand-written model is compared with the improved total time. Timing figures are the average over 30 instances.

3.7 Limitations

The prototype we have presented has some limitations in terms of the code which is allowed to be used as part of the problem specification. Most notably we do not currently support arrays, bit operations, characters/strings, floats or doubles except as constants. We have also excluded switch statements and do-while. These things are all technically feasible, it would just take a lot more time to support all of them. It should not be difficult to extend support to arrays, as these are really just primitive lists. The unsupported primitive types would require a corresponding solver representation, and appropriate constraints for each operator. Ideally standard library classes such as String would be supported using a manually translated class (e.g. _String) with a special purpose internal representation, as we have done for various collection classes (List, Set and Map). The excluded statements could be supported via a simple translation to supported statements, but a direct implementation would probably be better. Below we discuss more significant limitations.
Bounded execution path

Currently we assume an execution path with bounded length. Only bounded loops are supported. Similarly, any recursion in the problem definition must reach a point where we can detect through simple forwards propagation of domains that no further calls will be made. We do not attempt to check this condition at compile time, but if it is violated the expression construction code will not terminate. Obviously in a production system it would be desirable to detect cases where such behaviour may occur and give either a warning or error at compile time.

These restrictions mean that all pieces of code we consider have an execution path which can be visualised as a stick with edges skipping some nodes (e.g. skipping the then or else part of an if statement, or exiting a loop early), which is how we get around the path explosion problem. Unbounded loops, unbounded recursion, and less structured control flow statements like goto break this assumption. It might be possible in future to support unbounded loops using the approach suggested in [38], which basically involves unwinding loops lazily during the solving process as it is determined that an extra iteration is required. In conjunction with appropriate transformation techniques to convert recursion into an equivalent loop, this approach could possibly be used for recursion as well. Java does not have a goto statement, but theoretically it might be possible to support these by combining an explicit representation of the execution path (as used for loop untangling, see chapter 5) with lazy expansion of this graph in the style of lazy unwinding.

Compile-time code selection

Method overloading (e.g. in the form of inheritance), or function passing (e.g. via lambda expressions), can theoretically be supported as long as we know in advance which pieces of code may be executed and can produce at runtime an expression whose value indicates the chosen option. We can then simply replace the original call with a switch statement using this expression as the switch and having a case for each option. Currently however, we only handle these things when it can be determined at compile time which method will be called.

Theoretical limitations

There are certain assumptions that are necessary in general for a native interface. At compile time we must be able to compute the complete set of relevant code, and have access to this code in order to transform it. This means relevant code must not call an external library which is not either available for translation or pre-translated. We also rely on the assumption that the code defines a function between the decisions and the outcome. Therefore non-deterministic code, including user input, is not allowed. These restrictions can usually be overcome by moving the offending code out of the build or evaluate method and into the constructor of the Optimisable class or an extra initialisation method to be called before performing optimisation.
Final notes

It is important to note that all of the limitations discussed here (both theoretical restrictions and those just for our prototype) apply only to code defining an optimisation problem (the relevant code, as defined earlier). No restrictions are imposed on the rest of the program.

If this system were to be commercialised, it would be essential to have good error reporting alongside or even instead of a detailed specification of the allowed code. When expecting the user to make use of library types it is important to specify exactly what usages are allowed and what steps are required, because the user has no initial understanding of the new types. When programmers are using their own types the situation is different. In this case it is better to provide good error reporting so that the programmer can start with whatever approach is most obvious to them, and then work around any incompatibilities as they come up. The user does not need to understand exactly what code will be acceptable ahead of time; they are better served by help adjusting their specific attempt until it is acceptable.

3.8 Conclusion

We have designed and implemented a prototype native Java interface to CP, which aims to be intuitive and convenient for software developers. The necessary translation between paradigms is automated, allowing the programmer to work exclusively with a native Java definition of the optimisation problem. This significantly reduces the burden on the programmer and allows straightforward integration of optimisation functionality within a wider application.

A natural coding style is allowed with support for object variables, variable collections, and high level decision procedures, building on and significantly extending techniques used to implement the language CoJava.

Preliminary experiments show that further work is required to achieve satisfactory performance, but that there are gains to be made using very simple local adjustments to the model. The following chapters discuss our progress in improving the translation.
Chapter 4

Modelling Destructive Assignments

4.1 Introduction

Symbolic reasoning has been the crux of many software applications such as verifiers, test-case generation tools, and bug finders since the seminal papers of Floyd and Hoare [45, 62] in program verification and King [71] in symbolic execution for testing. Common to these applications is their translation of the program or some abstraction of it into equivalent constraints which are then fed into a constraint solver to be checked for (un)satisfiability.

The principal challenge for this translation is effective handling of destructive state changes. These both influence and depend on the flow of control, making it necessary to reason disjunctively across possible execution paths. In object oriented languages with field assignments, the disjunctive nature of the problem is further compounded by potential aliasing between object variables.

In this chapter we introduce a new, demand-driven technique for modelling destructive assignments, designed specifically to be effective for the difficult case of field assignments. The key idea is to view the value stored in a variable not as a function of the current state, but as a function of the relevant assignment statements. This allows us to avoid maintaining a representation of the entire program state, instead only producing constraints for expressions which are actually required.

The particular application we consider for our new technique is the native Java interface to CP introduced in Chapter 3. Experimental results using examples from this tool demonstrate that our new technique for modelling destructive assignments is superior to previous approaches, and can produce optimisation models comparable in efficiency to a simple hand written model for the same problem.
4.1.1 Running example

As a running example throughout this chapter we consider a smartphone application for group pizza ordering. Each member of the group nominates a number of slices and some ingredient preferences. The app automatically generates a joint order of minimum cost which provides sufficient pizza for the group, assuming that a person will only eat pizza with at least one ingredient they like and no ingredients they dislike. After approval from the user, the order is placed electronically.

Our focus is on the optimisation aspect of the application: finding the cheapest acceptable order. We assume that for each type of pizza both a price per pizza and a price per slice is specified. The order may include surplus pizza if it is cheaper to buy a whole pizza than the required number of individual slices.

Figure 4.1 shows a Java method defining this optimisation problem, called `buildOrder`. The problem parameters are the contents of the `people` list and the details stored in the `menu` object when `buildOrder` is called. Each call to the method `choiceMaker.chooseFrom` indicates a decision to be made, where the possible options are the `OrderItem` objects included in the list `pizzas` (the `Order` constructor creates an `OrderItem` for each pizza on the menu, all initially for 0 slices). The objective is to minimise the return value, which is the total cost of the order.

```java
int buildOrder() {
    order = new Order(menu);
    for (Person person : people) {
        // Narrow down acceptable pizzas
        pizzas.clear();
        for (OrderItem item : order.items)
            if (person.willEat(item))
                pizzas.add(item);
        // Choose from these for each slice
        for (int i=0; i<person.slices; i++) {
            OrderItem pizza =
                choiceMaker.chooseFrom(pizzas);
            pizza.addSlice();
        }
        return order.totalCost();
    }
}

class Order {
    List<OrderItem> items;
    int totalCost() {
        int totalcost = 0;
        for (OrderItem item : items)
            totalcost += item.getCost();
        return totalcost;
    }
}

class OrderItem {
    int pizzaPrice;
    int slicePrice;
    int fullPizzas = 0;
    int numSlices = 0;
    void addSlice() {
        numSlices = numSlices + 1;
        if (numSlices==slicesPerPizza) {
            numSlices = 0;
            fullPizzas = fullPizzas + 1;
        }
    }
    int getCost() {
        int cost = fullPizzas*pizzaPrice;
        if(numSlices > 0) {
            int slicesCost =
                numSlices * slicePrice;
            if(slicesCost > pizzaPrice)
                slicesCost = pizzaPrice;
            cost = cost + slicesCost;
        }
    }
}
```

Figure 4.1: A Java simulation of a pizza ordering optimisation problem.
4.1.2 Translating code into constraints

As mentioned previously, to evaluate different possible translations from procedural code to constraints we use examples from the tool described in Chapter 3. This tool actually performs the translation on demand at run-time (not as a compile time operation), which complicates the translation process somewhat. For the purpose of this chapter we will ignore such implementation details, using the following abstraction to simplify the description of the different translations.

We consider the translation to be split into two phases. In the first phase the code is flattened into a linear sequence of assignment statements, each of which has some conditions attached. We describe this transformation briefly in Section 4.2. In the second phase, which is the main focus here, the flattened sequence of assignments is translated into constraints.

4.2 Flattening

In the Java programming language only the assignment statement changes the state of the program. All other constructs simply influence which other statements will be executed. It is therefore possible to emulate the effect of a piece of Java code using a sequence of assignment statements, each with an attached set of conditions controlling whether or not it should be executed. The conditions reflect the circumstances under which this statement would be reached during the execution of the original code.

The flattening process involves unrolling loops, substituting method bodies for method calls, and removing control flow statements after adding appropriate execution conditions for the child statements. As an example, consider the method `getCost` shown in Figure 4.1. To flatten an `if` statement we simply add the `if` condition to the execution conditions of every statement within the `then` part. The body of `getCost` can be flattened into the following sequence of conditional assignment statements.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Variable</th>
<th>Assigned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>cost</td>
<td><code>fullPizzas \times pizzaPrice</code></td>
</tr>
<tr>
<td>2. <code>numSlices &gt; 0</code></td>
<td>slicesCost</td>
<td><code>numSlices \times slicePrice</code></td>
</tr>
<tr>
<td>3. <code>(numSlices &gt; 0, slicesCost &gt; pizzaPrice)</code></td>
<td>slicesCost</td>
<td><code>pizzaPrice</code></td>
</tr>
<tr>
<td>4. <code>numSlices &gt; 0</code></td>
<td>cost</td>
<td><code>cost + slicesCost</code></td>
</tr>
</tbody>
</table>

Note that each assignment statement applies to a specific variable. This may be a local variable identified by name (as above), or an object field `o.f` where `o` is a variable storing an object, and `f` is a field identifier. We call an assignment to an object field a field assignment. The value of the object variable `o` may

---

1 Our tool only supports loops with exit conditions unaffected by the decisions, or iteration over bounded collections. This means the number of loop iterations is always bounded. For unbounded loops partial unrolling can be performed, and the final model will be an under-approximation of the behaviour of the program.
depend on the decisions, so the concrete object whose field is updated by a field assignment is not necessarily known.

An important optimisation is to consider the declaration scope of variables. For example, if a variable is declared inside the then part of an if statement (as is the case for the slicesCost variable above), assignments to that variable need not depend on the if condition. In any execution of the original code where the if condition does not hold, this variable would not be created, and therefore its value is irrelevant. This means assignments 2 and 3 above do not need the condition numSlices > 0.

We also need to record the initial program state. For variables which exist outside the scope of the code being analysed, we add an unconditional assignment at the beginning of the list setting the variable to its initial value. We call this an initialising assignment. For object fields we add an initialising assignment for each concrete object.

Figure 4.2 shows the sequence of assignments produced by flattening our example function buildOrder for an instance with two people and three pizza types. Note that calls to ChoiceMaker methods are left untouched (these represent the creation of new decision variables), and expressions which do not depend on the decisions are calculated upfront. For example, the code used to find acceptable order items for each person does not depend on any decisions, so rather than including assignments originating in this part of the code in the flattened sequence, we simply calculate these lists and then use them as constants. Where these expressions are used as if conditions or loop exit conditions we exclude from the translation any unreachable code.

In the following sections, we assume our input is this flattened list of conditional assignment statements. We also use the notation Dom(v, i) to refer to the set of possible values for variable v at (just before) assignment i. This is easily calculated from the list of assignments. A conditional assignment adds values to the domain of the assigned-to variable, while an unconditional assignment replaces the domain.

### 4.3 Modelling Assignments: Existing Techniques

Using the flattening transformation described above and a straightforward translation of mathematical and logical expressions, we reduce the problem of representing Java code by constraints to that of modelling (conditional) assignment statements. In this section we describe two existing approaches to this, while in the next section we introduce a new proposed approach.

#### 4.3.1 Typical CP approach

One obvious technique for modelling assignments, and that used both in the previous chapter and in [27, 34], is to create a new version of the assigned-to variable for each assignment, and then use the latest version whenever a variable is referred to as part of an expression. If the assignment has some conditions,
<table>
<thead>
<tr>
<th>Cond</th>
<th>Object</th>
<th>Field/Var</th>
<th>Assigned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Veg</td>
<td>fullPizzas := 0</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Marg</td>
<td>fullPizzas := 0</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Mush</td>
<td>fullPizzas := 0</td>
<td></td>
</tr>
<tr>
<td>4-12.</td>
<td>other initialisation assignments (for numSlices, pizzaPrice, slicePrice)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>pizzas1 := [Veg, Marg]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>pizza1 := chooseFrom(pizzas1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>pizza1 . numSlices := pizza1.numSlices + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>b1 := pizza1.numSlices == slicesPerPizza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>(b1) : pizza1 . numSlices := 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td>(b1) : pizza1 . fullPizzas := pizza1.fullPizzas + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19-23.</td>
<td>repeat assignments 14-18 for 2nd slice (using vars pizza2 and b2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.</td>
<td>pizzas2 := [Marg, Mush]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25.</td>
<td>pizza3 := chooseFrom(pizzas2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.</td>
<td>pizza3 . numSlices := pizza3.numSlices + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27.</td>
<td>b3 := pizza3.numSlices == slicesPerPizza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28.</td>
<td>(b3) : pizza3 . numSlices := 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29.</td>
<td>(b3) : pizza3 . fullPizzas := pizza3.fullPizzas + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30-34.</td>
<td>repeat assignments 25-29 for 2nd slice (using vars pizza4 and b4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35-39.</td>
<td>repeat assignments 25-29 for 3rd slice (using vars pizza5 and b5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40.</td>
<td>totalcost := 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41.</td>
<td>cost1 := Veg.fullPizzas × Veg.pizzaPrice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42.</td>
<td>b6 := Veg.numSlices &gt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43.</td>
<td>slicesCost1 := Veg.numSlices × Veg.slicePrice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44.</td>
<td>b7 := slicesCost1 &gt; Veg.pizzaPrice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.</td>
<td>(b7) : slicesCost1 := Veg.pizzaPrice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.</td>
<td>(b6) : cost1 := cost1 + slicesCost1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47.</td>
<td>totalcost := totalcost + cost1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48-54.</td>
<td>repeat assignments 41-47 for 2nd order item (Marg)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55-61.</td>
<td>repeat assignments 41-47 for 3rd order item (Mush)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 62. | objective := totalcost |

Figure 4.2: Flattened version of the buildOrder method. We assume an instance where the menu lists three different types of pizza (vegetarian, margherita and mushroom), meaning the order will contain three OrderItems [Veg, Marg, Mush], and where the people list contains two Person objects, the first willing to eat vegetarian or margherita and requiring two slices, and the second willing to eat margherita or mushroom and requiring three slices. The b variables have been introduced to store branching conditions. Variables from methods called more than once and those used as the iteration variable in a loop are numbered to distinguish between the different versions.
the new version of the variable can be constrained to equal either the assigned value or the previous version, depending on whether or not the conditions hold. This is easily achieved using a pair of implications, or alternatively using an element constraint with the condition as the index. The element constraint has the advantage that some propagation is possible before the condition is fixed, so we will use this translation.

The constraint arising from a local variable assignment is shown below, where localvar0 is the latest version of localvar before the assignment, and localvar1 is the new variable that results from the assignment, which will become the new latest version of localvar. Note that we assume arrays in element constraints are indexed from 1. For convenience, in the rest of this chapter we use a simplified syntax for these constraints (also shown below).

| assignment:                               | condition : localvar := expression |
| constraint:                               | element(bool2int(condition)+1, [localvar0, expression], localvar1) |
| simple syntax:                            | localvar1 = [localvar0, expression][condition] |

This translation is only correct for local variables. Field assignments are more difficult to handle due to the possibility of aliasing between objects. However, if the set of concrete objects which may be referred to by an object variable is finite (which is the case for our application), then it is possible to convert all field assignments into equivalent assignments over local variables, after which the translation above can be applied.

For each concrete object, a local variable is created to hold the value of each of its fields. In the following we name these variables using the object name and the field name separated by an underscore. Then every field assignment is replaced by a sequence of local variable assignments, one for each of the possibly affected concrete objects. These new assignments retain the original conditions, and each also has one further condition: that its corresponding concrete object is the one referred to by the object variable. Where necessary to avoid duplication, an intermediate variable is created to hold the assigned expression.

An example of this conversion is shown below, where we assume the assignment is on line n and $\text{Dom}($objectvar, n) = {Obj1, Obj2, Obj3}.

| field assignment:          | condition : objectvar.field := expression |
| assignments:               | condition $\land$ (objectvar = Obj1) : Obj1_field := expression |
|                           | condition $\land$ (objectvar = Obj2) : Obj2_field := expression |
|                           | condition $\land$ (objectvar = Obj3) : Obj3_field := expression |

The final requirement is to handle references to object fields. We need to look up the field value for the concrete object corresponding to the current value of the object variable. To achieve this we use a pair of element constraints sharing an index as shown below, where fieldrefvar is an intermediate variable representing the retrieved value. We assume the same domain for objectvar.
In summary, this approach involves two steps. First the list of assignments is modified to replace field assignments with equivalent local variable assignments, introducing new variables as required. Then the new list (now containing only local variable assignments) is translated into constraints, with special handling for field references. This approach is quite simple, but can result in a very large model if fields are used extensively. To see the result of applying this translation to a portion of our running example, see Figure 4.3(a).

4.3.2 Typical SMT approach

One of the main reasons for the significant advances in program symbolic reasoning (e.g. verification and testing) during the last decade has been the remarkable progress in modern SMT solvers (see Section 1.2.4 for an introduction to SMT).

When using SMT, local variable assignments can be translated in the same way as for the CP approach (adding a new version of the variable for each assignment), but using an if-then-else construct (ite below) instead of an element constraint.

assignment: condition : localvar := expression

formula: localvar1 = ite(condition, expression, localvar0)

For field assignments, it is more convenient to use the theory of arrays. This theory extends the theory of uninterpreted functions with two interpreted functions read and write. McCarthy proposed [79] the main axiom for arrays:

\[ \forall a, i, j, x \text{ (where } a \text{ is an array, } i \text{ and } j \text{ are indices and } x \text{ is a value )} \\
\quad i = j \rightarrow \text{read}(\text{write}(a, i, x), j) = x \\
\quad i \neq j \rightarrow \text{read}(\text{write}(a, i, x), j) = \text{read}(a, j) \]

Note that since we are not interested in equalities between arrays we only focus on the non-extensional fragment.

Following the key idea of Burstall [28] and using the theory of arrays, we define one array variable for each object field. Conceptually, this array contains the value of the field for every object, indexed by object. Note however that there are no explicit variables for the elements.

An assignment to a field is modelled as a write to the array for that field, using the object variable as the index. The result is a new array variable representing the new state of the field for all objects. This is much more concise and efficient than creating an explicit new variable for each concrete object.
We still need to handle assignments with conditions. If the condition does not hold all field values should remain the same, so we can simply use an `ite` to ensure that in this case the new array variable is equal to the previous version.

**field assignment:**

\[
\text{cond : objectvar.field := expression}
\]

**formula:**

\[
\text{field1 = ite}(\text{cond}, \text{write}(\text{field0, objectvar, expression}), \text{field0})
\]

A reference to an object field is represented as a read of the latest version of the field array, using the object variable as the lookup index.

**field reference:**

\[
\text{objectvar.field}
\]

**formula:**

\[
\text{read}(\text{field0, objectvar})
\]

For a more complete example, see Figure 4.3(b). This example clearly demonstrates that the SMT formula can be much more concise than the CP model arising from the translation discussed in the previous section. Its weakness is its inability to reason over disjunction (compared to `element` in the CP approach). These two approaches are compared further in Section 4.4.3.
4.4 A New Approach to Modelling Assignments

The main problem with the CP approach presented earlier is the excessive number of variables created to store new field values for every object possibly affected by a field assignment. Essentially we maintain a representation of the complete state of the program after each execution step.

A complete representation of the state is not actually necessary. Our only real requirement is to ensure that the values retrieved by variable references are correctly determined by the assignment statements. Maintaining the entire state is a very inefficient way of achieving this, since we may make several assignments to a field using different object variables before ever referring to the value of that field for a particular concrete object. To take advantage of this observation, we move away from the state-based representation, instead simply creating a variable for each field reference, and constraining this to be consistent with the relevant assignments.

4.4.1 The general case

We first need to define which assignment statements are relevant (i.e. may affect the retrieved value) for a given variable reference. Let $a_i$ be the assignment on line $i$ of the flattened list, and $o_i$, $f_i$ and $c_i$ be the object, field identifier and set of conditions for this assignment. For a reference to variable obj.field occurring on line $n$, assignment $a_j$ is relevant iff the following conditions hold.

$$j < n \text{ and } f_j = \text{field}$$  
(occurs before the reference, uses correct field)

$$\text{Dom}(obj, n) \cap \text{Dom}(o_j) \neq \emptyset$$  
(assigns to an object which may equal obj)

$$\not\exists u : o_u = \text{obj}, f_u = \text{field}, c_u = \emptyset, j < u < n$$  
(not overwritten by unconditional assignment)

As an example, consider the reference to Veg.fullPizzas on line 41 in Figure 4.2. Of the eight assignments to the fullPizzas field (all of which occur before this reference), the following three are relevant. The others cannot affect the retrieved value as they use object variables (e.g. pizza3) whose domains do not include Veg.

1. Veg.fullPizzas := 0
2. pizza1.fullPizzas := pizza1.fullPizzas + 1
3. pizza2.fullPizzas := pizza2.fullPizzas + 1

For a correct model we need constraints ensuring that the retrieved value (Veg.fullPizzas) corresponds to the most recent assignment which updated the read variable. To achieve this we introduce a new integer variable indexvar whose value indicates which of the relevant assignments this is. We use three element constraints to ensure that the selected assignment applies to the correct object, has true execution conditions, and assigns a value equal to the result.

```
element(indexvar, [Veg, pizza1, pizza2], Veg)
element(indexvar, [true, b1, b2], true)
element(indexvar, [0, pizza1.fullPizzas + 1, pizza2.fullPizzas + 1], Veg.fullPizzas)
```
Note that pizza1_fullPizzas and pizza2_fullPizzas are the variables introduced for the field references used as part of the assigned values. These would be constrained using their own list of relevant assignments.

The only remaining requirement is that we must choose the latest applicable assignment. Using the natural order for the arrays this corresponds to the greatest index. We therefore add constraints stating that if at index $i$ the object variables are equal and the execution condition is true, then the selected index must be no less than $i$.

$$(b1 \land pizza1 = \text{Veg}) \rightarrow \text{indexvar} \geq 2$$
$$(b2 \land pizza2 = \text{Veg}) \rightarrow \text{indexvar} \geq 3$$

The general form of the constraints used for field references is shown below. References to local variables are treated as field references where the object variable is the same as that used for all relevant assignments. When this is the case the first element constraint is not required (as it is trivially satisfied), and the implications can be simplified. Other obvious simplifications are also applied.

$$
\text{field reference: } \quad \text{queryobj.field} \\
\text{relevant assignments: } \quad \text{cond1 : obj1.field := expr1} \\
\quad \quad \quad \quad \ldots \\
\quad \quad \quad \quad \text{condn : objn.field := exprn} \\
\text{constraints: } \quad \text{element(indexvar, [obj1, ..., objn], queryobj)} \\
\quad \quad \quad \text{element(indexvar, [cond1, ..., condn], true)} \\
\quad \quad \quad \text{element(indexvar, [expr1, ..., exprn], queryobj.field)} \\
\quad \quad \quad \quad \text{(cond2 \land queryobj = obj2) \rightarrow \text{indexvar} \geq 2} \\
\quad \quad \quad \quad \ldots \\
\quad \quad \quad \quad \text{(condn \land queryobj = objn) \rightarrow \text{indexvar} \geq n}
$$

As an optimisation, when the code contains more than one reference to some variable $v$, we insert an unconditional assignment to $v$ at the time of the earlier read, using the read result as the assigned value. This will become the earliest relevant assignment for the later read, which helps to avoid duplication of expressions and constraints.

There is some similarity between our extraction of relevant assignments and the dynamic slicing technique proposed in [64]. Note however that slicing is used only to reduce the number of statements to be translated into constraints; the actual translation still uses the standard approach.

4.4.2 Special cases

It is sometimes possible to detect a pattern to the relevant assignments which allows us to use a better translation. We look for three special cases: Boolean variables, sequences of assignments representing a sum calculation, and sequences of assignments representing a maximum or minimum calculation. When one of these cases is discovered, we automatically use a specialised translation.
Boolean variables

When the referenced variable is of type bool, we can define a Boolean expression for the retrieved value rather than using element constraints and implications. The expression (shown below) is true if some assignment with a true value applies to this variable and no later assignment with a false value applies.

field reference: \( q . \text{field} \)
assignments: \( c_i : o_i . \text{field} := e_i \quad i \in 1..n \)
expression: \[
\bigvee_{i \in 1..n} \left( c_i \land e_i \land (o_i = q) \land \bigwedge_{j \in i+1..n} (e_j \lor \neg c_j \lor (o_j \neq q)) \right)
\]

Local variables are handled in the same way except without the object equalities. Using this constraint instead of the generic constraint eliminates the need to introduce an index variable, and allows simplifications to be performed when objects are known to be equal, an assigned value is fixed, or an assignment is unconditional.

Sum calculations

Computations often involve taking the sum of a set of numbers. In a procedural language, sums are commonly calculated by iteratively adding each number to a variable representing the total. This coding pattern results in a sequence of writes where each written value is an addition of the previous value of this variable and some other number. When this pattern is detected, we can replace the usual constraints with a sum constraint.

relevant assignments: \[
\begin{align*}
\text{total} & := 0 \\
\text{total} & := \text{total} + \text{value1} \\
\text{total} & := \text{total} + \text{value2} \\
\text{total} & := \text{total} + \text{value3}
\end{align*}
\]
constraint: \[
\text{finaltotal} = \text{sum}(\{0, \text{value1}, \text{value2}, \text{value3}\})
\]

In the example above all assignments were unconditional and used exactly the same variable (the local variable total). It is also possible to use a general form of this constraint for sequences of assignments which do not represent a pure sum but a related calculation, such as counting the objects which satisfy some condition. Consider again the relevant writes for the reference to Veg.fullPizzas discussed earlier. A better constraint for Veg.fullPizzas is shown below.

assignments: \[
\begin{align*}
\text{Veg} \cdot \text{fullPizzas} & := 0 \\
(b1) & : \text{pizza1} \cdot \text{fullPizzas} := \text{pizza1.fullPizzas} + 1 \\
(b2) & : \text{pizza2} \cdot \text{fullPizzas} := \text{pizza2.fullPizzas} + 1
\end{align*}
\]
constraint: \[
\begin{align*}
\text{Veg.fullPizzas} = \\
\text{sum}(\{0, \text{bool2int}(b1 \land \text{pizza1}=\text{Veg}), \text{bool2int}(b2 \land \text{pizza2}=\text{Veg})\})
\end{align*}
\]
A major advantage of this constraint is that it removes the need to create and constrain the variables `pizza1_fullPizzas` and `pizza2_fullPizzas`. These can be excluded from the model entirely as they were only used to re-assign to the same variable, and are now not required to define the values retrieved from this field.

The general form of the alternative constraint used for sums is given below. The first assignment gives the initial value of `qobj.field`. If not already present it is created from the initialisation assignments for objects in the domain of `qobj`.

**field reference:** `qobj.field`

**assignments:**

- `qobj.field := init`
- `cond_i : obj_i.field := obj_i.field + expr_i, i ∈ 1..n`

**constraint:**

```plaintext
q.field = sum([init] + [expr_i * bool2int(cond_i ∧ obj_i = qobj) | i ∈ 1..n])
```

### Max/min calculations

Another common coding pattern is to calculate a maximum or minimum by iterating through a list of values overwriting a variable each time a smaller/larger value is found. As with sum, we can detect this pattern when building the constraints for the final read of the variable. This time in order for the alternative constraint to apply, every non-initialisation assignment must have a condition which compares the current value of the variable with the assigned value. When there are no other assignment conditions, and the variable is a local variable (or the assigned-to object is known to equal the read object for all relevant assignments), we can use a max/min constraint as shown below.

**assignments:**

- `max := init`
- `(value1 > max) : max := value1`
- `(value2 > max) : max := value2`

**constraint:**

```plaintext
finalmax = max([init, value1, value2])
```

We can again extend this to apply to field assignments and assignments with additional conditions. When extra conditions are present, we are calculating the maximum or minimum value for which these additional conditions hold. For a maximum, we constrain the result to be no less than any value for which the extra conditions hold, and to equal one of the values for which the conditions hold. Minimum is handled similarly.

**field reference:** `queryobj.field`

**assignments:**

- `(cond_i ∧ value_i > obj_i.field) : obj_i.field := value_i, i ∈ 1..n`

**constraints:**

```plaintext
\lor_{i ∈ 1..n}(cond_i ∧ (obj_i = queryobj) ∧ (queryobj.field = value_i))
\land_{i ∈ 1..n}(cond_i ∧ obj_i = queryobj) → queryobj.field ≥ value_i
```
4.4.3 Comparison with earlier approaches

We compared the three presented translation techniques experimentally, using the pizza ordering example plus two benchmarks from Chapter 3 (the others require support for collection operations, as discussed in the next section). We used 30 instances for each of several different sizes to evaluate scaling behaviour. For the original (orig) and new (new) CP approaches we show the effect of adding special cases (orig+ and new+). Special cases can be detected in the original method, but only for local variables. Using the new translation makes these cases also recognisable for fields. As a reference we also include a fairly naive hand written model for each problem (hand). The Java code defining the problems and all compared constraint models are available online at http://people.eng.unimelb.edu.au/pstuckey/optmodel/.

The CP models were solved using the lazy clause generation solver Chuffed. The SMT versions were solved using Z3 [37]. Z3, like most SMT solvers, does not have built-in support for optimisation. We used a technique similar to [101] to perform optimisation using SMT: in incremental mode, we repeatedly ask for a solution with a better objective value until an unsatisfiable result is returned.

Table 4.1 shows average solving time and failures for the different models. The small numbers indicate timeouts (> 600s). These were included in the average calculations. The results show that while the SMT approach does compete with the original approach, with special case treatment it does not. The new approach is quite superior and in fact has a synergy with special cases (since more of them are visible). In fact, new+ competes with hand except for pizza where it appears that the treatment of the relationship between slices and full pizzas used in hand is massively more efficient than the iterative approach in the simulation.

<table>
<thead>
<tr>
<th></th>
<th>Time (secs)</th>
<th>Failures (000s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>smt orig</td>
<td>orig+ new new+ hand</td>
</tr>
<tr>
<td>proj1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>2.2 23.0 0.1</td>
<td>12.1 0.1 0.1 56 0 34 0 0</td>
</tr>
<tr>
<td>225</td>
<td>2.4 3.2 0.1 1.5 0.1 0.1 9 0 4 0 0</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>1.6 61.9 0.1</td>
<td>61.7 3 0.1 99 0 127 0 0</td>
</tr>
<tr>
<td>proj2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>115.6 84.8 42.7 51.7 23.7 7.6</td>
<td>39 31 110 35 22</td>
</tr>
<tr>
<td>24</td>
<td>221.1 286.9 167.6 170.9 129.2 92.2 92 89 368 239 280</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>262.7 376.2 10 293.3 10 255.9 11 137.9 128.9 11 120 144 583 251 452</td>
<td></td>
</tr>
<tr>
<td>pizza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>56.0 37.4 51.25</td>
<td>7.0 3.1 2.0 175 118 30 14 0</td>
</tr>
<tr>
<td>4</td>
<td>226.4 180.9 175.7 138.7 138.7 79.3 2.1 544 541 377 252 1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>480.9 22 411.8 18 407.5 18 343.4 13 298.3 12 2.2 1170 1216 865 945 7</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Comparing three approaches to modelling destructive assignments.
4.5 Collection Operations

The code for the pizza ordering example makes use of collection classes from the Java Standard Library: `Set`, `List` and `Map`. In this case no special handling is required as all collection operations are independent of the decisions, but often it is more natural to write code where that is not the case. For example, say we wished to extend our application to choose between several possible pizza outlets, each with a different menu. We could do this by adding one extra line at the beginning of the `buildOrder` function.

```java
menu = chooseFrom(availableMenus);
```

This change means the contents of the `OrderItem` list in the `Order` class will depend on the decisions, so the `for` loop iterating over this list (in `buildOrder`) will perform an unknown (though bounded) number of iterations, and the result of any query operation on this list will also depend on the decisions.

In the previous chapter, collection operations were supported by introducing appropriately constrained variables representing the state of each collection after each update operation (e.g. `List.add`). Query operations (e.g. `List.get`) were represented as a function of the current state of the relevant collections. This is analogous to the way field assignments were handled, with the same drawbacks.

Fortunately our new technique can also be extended to apply to collection operations, resulting in a much more efficient representation. Where previously the flattened list of state changing operations contained only assignments, we now also include collection update operations. Then every query operation on a collection is treated analogously to a field reference. That is, a new variable is created to hold the returned value, and constraints are added to ensure that this value is consistent with the relevant update operations.

Below we provide details of the constraints used for `List`, `Set` and `Map` operations. We then give experimental results using collection-related benchmarks from Chapter 3.

4.5.1 List

For the `List` class we support update operations `add` (at end of list) and `replace` (item at index), and query operations `get` (item at index) and `size`.

A code snippet containing one of each operation type is shown below. Also shown are the assumed possible variable values and initial list contents, and the flattened list of collection update operations.

```java
if(cond) {
    list1.add(A);
    list1.replace(0, item);
}
if(list2.size() > ind) item = list2.get(ind);
```

(a) Code

<table>
<thead>
<tr>
<th>Cond</th>
<th>List</th>
<th>Index</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>list1</td>
<td>size1</td>
<td>A</td>
</tr>
</tbody>
</table>

(b) Variables

<table>
<thead>
<tr>
<th>Cond</th>
<th>List</th>
<th>Index</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1[0] : list1[0] := A (add)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1[1] := B (add)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2[0] := C (add)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) Update Operations
Each update operation has an associated condition, list, index and item. For the add operation, the index is a variable \texttt{size1} holding the current size of \texttt{list1}. The first three operations in the table reflect the original contents of the lists.

**Constraints for size queries**

With this limited set of supported update operations (which is nevertheless sufficient to cover all code used in the benchmarks from Chapter 3), the size of a list is simply the number of preceding add operations applying to this list and having true execution conditions. Note that the replace operation is not relevant to size.

query: \[
sizeresult := \text{list2.size()}
\]

constraint: \[
sizeresult = \text{sum([ bool2int(list2=L1), bool2int(list2=L1),}
\text{ bool2int(list2=L2), bool2int(list2=list1 \land \text{cond} )])}
\]

**Constraints for get queries**

A get query is treated almost exactly like a field reference. The value returned must correspond to the most recent update operation with true execution condition applying to the correct list and index. There is however one extra complication to be considered. Constraining the get result to correspond to an update operation has the effect of forcing the index to be less than the size of the list. This is only valid if the get query is actually executed.

In the constraints shown below, the final element of each array has been added to leave the index unconstrained and assign an arbitrary value \texttt{A} to our result variable when the get would not be executed (\texttt{sizeresult>ind} is false). Without this the constraints would force \texttt{ind} to correspond to an operation on \texttt{list2} regardless of whether or not the get query is actually executed, incorrectly causing failure when \texttt{list2} is empty. We fix the result rather than leaving it unconstrained to avoid searching over its possible values.

query: \[
\text{getresult := list2.get(ind)}
\]

constraints: \[
\text{element(indexvar, [L1,L1,L2,list1,list1,list2], list2)}
\text{element(indexvar, [0,1,0,size1,0,ind], ind)}
\text{element(indexvar, [true,true,true,cond,cond,\neg(sizeresult>ind)], true)}
\text{element(indexvar, [A,B,C,A,item,A], getresult)}
\text{(list2=L1) \land (ind=1)} \rightarrow \text{indexvar} \geq 2
\text{(list2=L2) \land (ind=0)} \rightarrow \text{indexvar} \geq 3
\text{(list2=list1) \land (ind=size1) \land \text{cond}} \rightarrow \text{indexvar} \geq 4
\text{(list2=list1) \land (ind=0) \land \text{cond}} \rightarrow \text{indexvar} \geq 5
\neg(sizeresult>ind) \rightarrow \text{indexvar} \geq 6
\]

**4.5.2 Set**

For the Set class we support the update operation add, and query operations contains and size. The example below shows the flattened list of update oper-
ations produced for a short snippet of set manipulation code. We assume that the sets are initially empty (if not then initialisation updates would be added to the flattened list as for the List example above).

```
set1.add(A);
set2.add(item1);
if(cond1) set1.add(B);
boolean c = set1.contains(item1);
int s = set1.size();
```

(a) Code

```
set1 ∈ {S1,S2,S3}   set2 ∈ {S1,S2,S3}   cond1 ∈ {true,false}
item1 ∈ {A,B,C}     set1 .add A
```

(b) Variables

```
cond1 : set1 .add B
```

(c) Update Operations

### Constraints for contains queries

A contains query should return true if an add operation using the specified set and value has true execution condition.

**query:**

```
containsresult := set1.contains(item1)
```

**constraint:**

```
containsresult = ((item1=A) ∨ (set2=set1 ∧ item1=B))
```

### Constraints for size queries

Unlike for List, the size of a Set is not necessarily equal to the number of preceding add operations, as the same item may have been added more than once. So for a Set size query, we insert a contains query for each possibly contained item, and then use a Boolean sum constraint to constrain the size to equal the number of these queries returning true.

**query:**

```
sizeresult := set1.size()
```

**constraints:**

```
containsA = true
containsB = ((set2=set1 ∧ item1=B) ∨ cond1)
containsC = (set2=set1 ∧ item1=C)
bool_sum([containsA, containsB, containsC], sizeresult)
```

Note that in our example the query set1.contains(A) can be simplified to true, as there is an unconditional add operation using these exact arguments. This means our size query result will have an immediate lower bound of 1.

### 4.5.3 Map

The Map class combines the functionality of the Set and List classes. The supported operations are put (value at key), get (value at key), containskey, and size (of key set). The get query is treated like a List get. For a List get, the result is the value assigned by the latest add/replace with matching list and index. For a Map get the result is the value assigned by the latest put with matching map and key. The containskey and size queries are implemented in exactly the same way as the Set contains and size queries. For these queries the values associated with put operations are ignored. Each put is simply interpreted as an add using the key as the item.

85
4.5.4 Comparison on benchmarks with collections

Table 4.2 compares the various translation approaches (excluding smt and orig which were shown to be not competitive in Table 4.1) and hand written models, using problems involving collections. It is clear that the new translation substantially improves on the old in most cases, and is never very much worse (bins, knap1). With the addition of special case treatment the new translation is often comparable to the hand written model, though certainly not always (proj3, route). In a few instances it is superior (bins, golf), this may be because it uses a sequential search based on the order decisions are made in the Java code, or indeed that the intermediate variables it generates give more scope for reusable nogood learning.

4.6 Conclusion

Effective modelling of destructive assignment is essential for any form of reasoning about procedural code. We have developed a new encoding of assignment and state that gives effective propagation of state-related information. We demonstrate the effectiveness of this encoding for the automatic generation of optimisation models from simulation code, showing that the resulting model can be comparable in efficiency to a hand-written optimization model.

In the future we plan to investigate the use of this encoding for applications such as test generation. The main difference is the lack of a known initial state. This will require the creation of variables to represent unknown initial field values, with constraints ensuring that if a pair of object variables are equal then their corresponding initial field variables are also equal. Uncertainty about the initial state will also affect the number of relevant assignments for field references. For a query object with unbounded domain all assignments to the same field occurring prior to the read are relevant, unless one of these is an unconditional assignment using this exact variable. These differences may mean that redundant constraints relating reads to each other (which we have not discussed in this chapter due to their lack of impact for our application) become more important for effective propagation.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (secs)</th>
<th>Failures (000s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2.6</td>
<td>8.1</td>
</tr>
<tr>
<td>14</td>
<td>82.8</td>
<td>95.4</td>
</tr>
<tr>
<td>16</td>
<td>327.2</td>
<td>315.1</td>
</tr>
<tr>
<td>golf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,3</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>4,4</td>
<td>3.4</td>
<td>3.4</td>
</tr>
<tr>
<td>5,2</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>golomb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.3</td>
<td>10.8</td>
</tr>
<tr>
<td>9</td>
<td>14.0</td>
<td>55.4</td>
</tr>
<tr>
<td>10</td>
<td>161.5</td>
<td>281.1</td>
</tr>
<tr>
<td>knap1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>2.1</td>
<td>33.3</td>
</tr>
<tr>
<td>80</td>
<td>7.5</td>
<td>95.7</td>
</tr>
<tr>
<td>90</td>
<td>14.2</td>
<td>180.2</td>
</tr>
<tr>
<td>knap2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>20.9</td>
<td>247.8</td>
</tr>
<tr>
<td>80</td>
<td>88.4</td>
<td>935.2</td>
</tr>
<tr>
<td>90</td>
<td>223.6</td>
<td>2263.9</td>
</tr>
<tr>
<td>knap3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>26.2</td>
<td>14.3</td>
</tr>
<tr>
<td>50</td>
<td>81.1</td>
<td>25.0</td>
</tr>
<tr>
<td>60</td>
<td>295.2</td>
<td>58.7</td>
</tr>
<tr>
<td>proj3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>153.9</td>
<td>289.3</td>
</tr>
<tr>
<td>12</td>
<td>509.4</td>
<td>778.5</td>
</tr>
<tr>
<td>14</td>
<td>600.0</td>
<td>807.3</td>
</tr>
<tr>
<td>route</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>34.2</td>
<td>34.0</td>
</tr>
<tr>
<td>6</td>
<td>338.3</td>
<td>195.8</td>
</tr>
<tr>
<td>7</td>
<td>600.0</td>
<td>263.2</td>
</tr>
<tr>
<td>talent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,8</td>
<td>11.1</td>
<td>25.9</td>
</tr>
<tr>
<td>4,9</td>
<td>170.8</td>
<td>159.7</td>
</tr>
<tr>
<td>4,10</td>
<td>545.5</td>
<td>459.7</td>
</tr>
</tbody>
</table>

Table 4.2: Further comparison for benchmarks with variable collections.
Chapter 5

Loop Untangling

5.1 Introduction

In this chapter we consider the treatment of loops (for loops and while loops) during the translation from procedural code into equivalent constraints. We focus on bounded loops, where a limit on the number of iterations is assumed or can be computed. Bounded loops arise in bounded model checking (e.g. [34]) and other forms of symbolic execution, as well as our motivating application of allowing optimisation problems to be defined natively in Java.

The typical approach to handling bounded loops is loop unwinding, which involves flattening the loop by creating a copy of the body for each potential iteration. This was the approach used in previous chapters of this thesis and also in e.g. [27, 34, 8, 21].

The key insight of this chapter is that the iterations of a loop do not necessarily need to be identified by the order of execution. That is, when creating copies of the loop body we do not have to label them as the iteration reached by execution first, second, and third, as is done in standard loop unwinding. Instead we can choose a different way of identifying each potential iteration, and link them together using a separate representation of the execution order.

We describe here a new technique called loop untangling which does just that. Instead of execution order, iterations are identified by the value taken by a key expression within the loop body. Using this approach it is possible to vastly simplify the constraints for each copy of the loop body, because the value of this key expression is known. As shown in Section 5.4, this can result in greatly improved solver performance.

5.2 Motivating Examples

We give here two example programs where standard loop unwinding produces a particularly inefficient model, and sketch how loop untangling can provide a better translation. We will later show how this can be achieved automatically.
First example: routing

Our motivating examples come from the prototype native Java interface to CP
introduced in Chapter 3. The first example (below) is a routing problem which
was one of the original benchmarks from that chapter. Given a set of jobs, each
of which has a pickup stop and a delivery stop, the problem is to choose the
shortest Hamiltonian route visiting all stops, with no delivery stop visited before
the corresponding pickup. Recall that the ChoiceMaker argument provides non-
deterministic decision making methods. In this case the method chooseOrder is
used, which returns a permutation of the given list.

```java
int buildRoute(ChoiceMaker chooser) {
  List<Stop> route = chooser.chooseOrder(allStops);
  // compute arrival times
  int currentLocation = startLocation;
  int currentTime = 0;
  for(Stop stop: route) {
    int nextLocation = stop.getLocation();
    currentTime += travTime(currentLocation, nextLocation);
    stop.arrivalTime = currentTime;
    currentLocation = nextLocation;
  }
  tripFinishTime = currentTime + travTime(currentLocation, startLocation);
  // check no pickup is after the corresponding delivery
  for(Job j : jobs) {
    if(j.pickupStop.arrivalTime > j.deliveryStop.arrivalTime)
      throw new Exception();
  }
  return tripFinishTime;
}
```

Figure 5.1: Java code defining a routing problem.

Consider the first loop, which computes the arrival time for each stop. Let
us assume the list allStops contains three stops [A,B,C], which means these three
stops also occur exactly once in the route list, but in an unknown order. Using
standard loop unwinding we would create a copy of the loop body for the first,
second, and third iteration. For each of these, the value of stop may be A, B, or
c. All of the other variables depend on stop, so their values are also unknown.
Furthermore, when we later look up the arrival time for the pickup and delivery
stop for each job, the value retrieved could be the currentTime value computed
in any of the three iterations.

Figure 5.2(a) shows the (idealized) MiniZinc [86] produced by loop unwind-
ing. The decisions are the permutation of the stops, enforced by alldifferent.
Expressions computed within the loop body are represented using arrays in-
dexed by iteration time Ite = 1..n (where n is the number of iterations), or
Ite0 = 0..n for those having a version before the loop. Constraints simulate
the calculation within the loop, using the locations and dist arrays to look up
parameter values referenced within the getLocation and travTime methods.

89
To constrain the final arrival time for each stop $s$ we need to determine which iteration was the last where we changed the arrivalTime field of the stop $s$, encoded using which. We then can lookup the currentTime in that iteration to give the arrivalTime.

```plaintext
array[ite] of var Stop: route;
constraint alldifferent(route);
array[ite] of var Location: nextL;
array[ite0] of var Location: currL;
array[ite0] of var int: travT;
constraint currL[0] = startL;
constraint currT[0] = 0;
constraint forall (i in ite) (nextL[i] = locations[route[i]] ∧ travT[i] = dist[currL[i-1],nextL[i]] ∧ currT[i] = currT[i-1] + travT[i] ∧ currL[i] = nextL[i]);
array[Stop] of var Stop0: prevS;
constraint path(prevS,...,0);
array[Stop] of var Location: nextL;
array[Stop0] of var Location: currL;
array[Stop] of var int: travT;
array[Stop0] of var int: currT;
constraint currT[0] = 0;
constraint currL[0] = 0;
constraint forall (j in Job) (currT[which[pickup[j]]] <= currT[which[delivery[j]]]);
constraint forall (i in ite) (nextL[i] = locations[route[i]] ∧ travT[i] = dist[currL[i],nextL[i]] ∧ currT[i] = currT[i-1] + travT[i] ∧ currL[i] = nextL[i]);
constraint forall (s in Stop) (which[s] = max(i in ite) (i*bool2int(route[i] = s)));
constraint forall (j in Job) (currT[which[pickup[j]]] <= currT[which[delivery[j]]]);
```
for expressions calculated within the loop are indexed by \texttt{Stop} or \texttt{Stop0} (which includes an artificial initial stop 0). The decisions are \texttt{prevS}, that is for each iteration/stop, what is the previous iteration/stop (or 0 for the first iteration). This is used to look up values that depend on the previous loop iteration (or initialization), while a path constraint ensures that these predecessor variables correspond to a Hamiltonian path starting anywhere and ending at the artificial stop 0. The \texttt{arrivalTime} for a stop is now simply equal to the \texttt{currentTime} computed in the iteration corresponding to that stop.

**Second example: pizza ordering**

Our second example is a pizza ordering problem which was the running example from Chapter 4. The pertinent code is shown in Figure 5.3. The task is to find the cheapest pizza order which will satisfy a group of discriminating pizza eaters. The code computes the acceptable pizzas for each person, then chooses from these for each slice up to the number the person requires. Once the slices are chosen the cost of the order is calculated taking into account a discount for ordering whole pizzas.

```java
1 int buildOrder() {
2     order = new Order(menu);
3     for (Person person : people) {
4         // Find acceptable pizzas
5         pizzas.clear();
6         for (OrderItem item : order.items)
7             if (person.willEat(item))
8                 pizzas.add(item);
9         // Choose type for each slice
10        for (int i=0;i<person.slices;i++) {
11            OrderItem pizza = chooser.chooseOne(pizzas);
12            pizza.addSlice();
13        }
14     }
15     return order.totalCost();
16 }
```

The loop we consider this time is the one on lines 10–14, within which we make the decisions and tally up the number of slices and pizzas for each pizza type. Here the order of iterations is actually irrelevant to the final result (the cost of the order). It is only the number of times each type of pizza is chosen which matters. Unwinding the loop introduces symmetries and also creates a lot of added uncertainty as the pizza type whose \texttt{numSlices} and \texttt{numPizzas} field is changed in each iteration is unknown. It would be much better to create a variable giving the number of times each type of pizza is chosen, and then constrain the final value of \texttt{numSlices} and \texttt{numPizzas} for each pizza type to be a function of this variable.
Loop untangling achieves this by labelling the iterations of the loop on lines 10–14 by the return value of `chooseOne`. Note that in this case the label is not unique, so we will need a copy of the body for e.g. the first time `Vegetarian` is chosen, and the second time, up to the maximum number of times this pizza type may be chosen. In each of these iterations the value of `numSlices` and `numPizzas` will be fixed, as these values depend only on the number of times this pizza type has previously been selected, which is known by definition of the iteration. Furthermore, when we later look up these values for a particular pizza type, we know that the result will be the value computed in one of the iterations corresponding to that pizza type. Which iteration will depend only on the number of times that pizza type is chosen. The resulting constraint system is far simpler and propagates much more efficiently.

**Achieving the better translations automatically**

The following sections describe a loop untangling technique which can be applied to any loop, and which when applied to the examples above results in much better performance than standard loop unwinding. It is not necessary to detect explicitly that in the first example the value of `currentLocation` is exactly the value of `nextLocation` from the previous iteration, nor to detect in the second example that the order of iterations is irrelevant. Provided the appropriate choice of labelling scheme for iterations our generalised implementation automatically produces a model which is equivalent to the better model in both cases.

### 5.3 General Loop Untangling Technique

This section explains the process of converting code into equivalent constraints using loop untangling rather than loop unwinding. The assumed translation technique is the query based approach described in Chapter 4. The key feature of this technique is that rather than modelling the current state of the program at each execution step, we constrain the value of each state query to correspond correctly to the preceding state changes. This is a necessary prerequisite for loop untangling because it allows the execution order of state changes to be viewed as a decision.

The translation is broken into two phases. First the code is flattened into a list of basic steps: state changes, state queries, and path control points. Then the result of each state query in this list is constrained to correspond correctly to the changes and control points.

In the previous chapter, it was assumed that loops would be unwound as part of the flattening process, and the flattened list contained only state changes with conditions reflecting the effect of path control statements. Here we redefine the flattening process to allow loops to be untangled instead, keeping state queries and path control points explicit in the result. It is also necessary to adjust the constraints used for state queries, as the previously used constraints assumed a known order for state changes.
5.3.1 Programs as ordered state changes and state queries

We consider a Java program to consist of a sequence of basic steps, each of which is a state change, state query, or path control point. At the lowest level all state changes are assignments and all state queries are variable references. However, since our application of interest (defining combinatorial optimisation problems using imperative code) tends to make heavy use of collections (specifically sets, lists and maps), we treat the core collection operations as atomic state changes (e.g. add item to list) and state queries (e.g. length of list). Path control points are points in the code where execution branches or merges. That is, break, continue and return statements, plus the beginning and end of then blocks, else blocks and loop bodies, and the end of methods (if there are multiple return statements).

5.3.2 Flattening

The first step in our translation is to convert the code into a list of basic steps. For example the code in lines 5-12 of the routing example (Figure 5.1) is flattened as shown in Figure 5.4. To improve readability we have not separated compound queries into individual parts. For example stop.getLocation() is actually a query for the value of the stop variable, and then a query for the result of the getLocation method called on that stop variable, which is itself a query for the location field of the stop.

Note that this list is not really flat yet, as items inside loop bodies may occur more than once in an execution of the program. To solve this we need to create copies of the loop body in such a way that each copy is executed at most once.

5.3.3 Creating iterations

When standard loop unwinding is used (as in the previous chapter), we create a copy of the loop body for each potential iteration and label them as the first, second, third etc. The execution order is fixed, but each individual iteration can have a large amount of uncertainty. The idea behind loop untangling is to instead create and label our iterations in a way that reduces the uncertainty within each individual iteration.

Figure 5.4: Flattened loop
The first step is to choose a state query inside the body to be used as the label query. The label query is how we will refer to the loop iteration, and ideally knowing the value of the label query will make the loop body much easier to model. Currently this choice is specified via annotation, although it seems clear that some simple static analysis should give us good choices. In our illustrative examples, we choose the iteration argument \texttt{stop} ($q_2$ in Figure 5.4) and the choice of pizza type assigned to \texttt{pizza} (line 11 in Figure 5.3).

Given a label query $q_L$, we determine the maximum number of times the loop body may be executed ($n$), and for each iteration $i \in 1..n$ we compute the set of possible values $D_i$ which could be taken by $q_L$. This is exactly the same calculation as would be done as part of standard loop unwinding.

\textbf{Example} For the pizza example, let us assume $\text{people} = \{\text{Ant, Bee}\}$, and that Ant wants one slice of Vegetarian or Caprichiosa, and Bee wants two slices which could be Margherita or Vegetarian. Unwinding the loop produces three iterations, with label query domains: $D_1 = \{\text{Veg, Cap}\}$, $D_2 = \{\text{Mar, Veg}\}$, $D_3 = \{\text{Mar, Veg}\}$.

We then create copies of the loop body as follows. For each value $v$ in the union of the domains $D_i$ computed above, we create $k$ copies of the loop body, where $k$ is the number of iterations in which $D_i$ contains $v$. For each copy, we add a constraint that if this iteration is reached by execution then the value of $q_L$ is $v$. This means that we can assume a fixed value for each iteration. If execution reaches the iteration then we know its value will be $v$, and if execution does not reach this iteration then the value of any query contained in it is irrelevant. When multiple copies are created for value $v$ we also impose a fixed execution order on these to eliminate symmetry, and number them accordingly.

\textbf{Example} Using the same assumptions for the pizza example, we create 3 copies of the loop body for Veg, two copies for Mar, and one for Cap. We impose a fixed order on the Veg and Mar iterations, labelling these Veg$_1$, Veg$_2$, Veg$_3$, and Mar$_1$, Mar$_2$. For each iteration we impose a constraint on the chosen pizza type based on the label (e.g. for Veg$_1$ we must choose Veg). Therefore the value held in the \texttt{pizza} variable in each iteration is fixed.

Note that the added constraint setting $q_L$ to take value $v$ does not replace the constraints ordinarily used to define the result of the query based on the preceding state changes (e.g. that the chosen pizza is one of those in the allowed set for the current person). These are still needed but they will now impose a constraint on the (no longer fixed) execution order rather than the query result.

Note also that we may create more than $n$ copies of the loop body. A good choice of label query will remove a lot of uncertainty from individual iterations without introducing too many extra iterations. If for the chosen label query every computed domain $D_i$ contains only a single value, then no uncertainty can be removed and loop untangling is equivalent to loop unwinding.

In the routing example we create a single copy of the loop body for each stop in the \texttt{allStops} list, as we know that each occurs exactly once in \texttt{stopsInOrder} and therefore will occur in exactly one iteration. The new list of basic steps will
have three copies of the body (assuming there are 3 stops A,B,C). We will add subscripts \(a, b, c\) to the listed step ids in Figure 5.4 to refer to them.

For nested loops such as the one in the pizza example, we create copies of the bodies separately. That is, copies of the inner loop body are not associated with a particular outer iteration. However, there will be multiple copies of the start and end loop nodes for the inner loop, and each of these will belong to a particular iteration of the outer loop.

### 5.3.4 Modelling the execution path

When standard loop unwinding is used, path control points can only cause execution to skip state changes. The relative order is known. So previously we have modelled the execution path implicitly, simply calculating a Boolean expression for the conditions under which each state change is executed.

When iterations are identified by something other than execution order, the relative execution order of iterations, and therefore of the basic steps contained in them, is unknown. In order to impose constraints on the order of iterations, and to reason correctly about the relationships between state changes and queries, we now need an explicit representation of the execution path.

Our approach is to compute a graph of the possible execution paths, similar to a control flow graph, but containing a node for every copy of each basic step, rather than a single node for each basic block. The edges are constructed as follows.

- A state query or state change has a single outgoing edge leading to the following step.
- The control point at the beginning of a then or else block has an edge leading to the first step in the block, and another edge leading directly to the end of the block.
- A continue, break, or return control point has a single outgoing edge to the end of the associated loop body, loop, or method respectively.
- A start-loop control point has an edge to its associated end-loop control point, and to every start-body control point for its loop.
- An end-body control point has an edge to the start-body point for each other iteration of that loop, and to every version of the end-loop control point.
- An exception step has no outgoing edges.

Any valid solution must correspond to a path through this graph (between the fixed start and end steps). Note that as required by our application this prevents exception points from being reached.

We create for each basic step \(s\) a variable \(\text{next}[s]\) whose value gives the next step executed after \(s\). Clearly for most steps the value is fixed to the unique
successor step. This next array must be constrained to ensure that the final values do indeed represent a path visiting a subset of the steps. In Chapter 6 we introduce a constraint subpath to be used for this purpose.

Not all paths through the graph represent valid execution paths. The use of certain edges (where execution branches) is conditional on the result of a Boolean state query referred to in that step. The edge leading into a then or else block can only be used if the if condition is true or false respectively. An edge leading into a loop body is only valid if a query for the loop entry condition returns true. In Figure 5.4 the query used to control the use of edges is shown in brackets next to the source node. For loops (both start-loop and end-body control points) if the query shown is false then the edge to the end-loop control point must be used. These constraints are easily imposed by linking values of the next variables with the appropriate conditions.

Example Figure 5.5(a) shows a portion of the execution graph for our routing example with basic blocks collapsed. The start can reach each loop iteration for stop = A, B or C, and these can reach each other and the end of the loop.

As an example of the conditions on edges, consider the edges leaving step $p_{3a}$. Setting $\text{next}[p_{3a}] = p_4$ requires $q_{9a} = \text{false}$, as this edge represents exiting the loop, while $\text{next}[p_{3a}] = p_{2b}$ (or $\text{next}[p_{3a}] = p_{2c}$) requires that $q_{9a} = \text{true}$, as these edges represent re-entering the loop.

Example For the pizza example (Figure 5.5(b)), edges which can be discounted upfront due to false edge conditions or constraints on the label query are not shown. Since Ant runs first it can reach only the first instance of Veg or Cap, and each of these can reach its end since Ant only picks one slice. For Bee the start can reach the first Mar or the first or second Veg. Each of these nodes can reach only the next of the same category or any of the other category, and the end of Bee’s loop. Outside this part of the graph is a mandatory path from the end of Ant’s loop to the start of Bee’s.

Figure 5.5: Path control graphs for: (a) Fig. 5.1 (5–12), (b) Fig. 5.3 (10–14).
As mentioned previously, if we have created multiple iterations for a given value of the label query, we impose a fixed order on these to eliminate symmetry. This means that (as shown in Figure 5.5(b)) edges leading from the start loop control point to the second or later copy of the body for each value of the label query are excluded immediately, and between iterations for the same value we only keep edges leading between successive copies. However, this is not sufficient, as the graph may still contain paths which do not respect the imposed order (e.g. in Figure 5.5(b) the graph does not exclude path $sB - V1 - M2 - V2 - M1 - eB$). To eliminate such paths we impose a constraint $\text{before}(sb_x, sb_y)$ for each successive pair of start-body steps $sb_x, sb_y$ belonging to iterations with the same label value. We use a simple definition of the before function, described below.

We also need further constraints on the edges for nested loops, to ensure that we do not enter the inner loop from one outer iteration and leave through a different outer iteration. For example we cannot enter node $V1$ from $sA$ and leave through $eB$. This is prevented by adding a constraint on the start-body and end-body control points ($s_i$ and $e_i$ for $i$ in the iteration set $I$) for each loop.

$$\forall i, j \in I, \neg(\text{before}(s_i, e_j) \land \text{before}(e_j, e_i))$$

This ensures that no other end-body step from the outer loop can come between a pair of associated start-body and end-body steps for that loop.

**Simple definition of before**

A straightforward implementation of before can be achieved by creating a time variable for each node in the graph of possible execution paths, and adding a constraint for each edge to say that if that edge is used then the time of the destination is one greater than the time of the source. Then before can be defined as follows.

$$\text{before}(a, b) = \text{time}[a] < \text{time}[b]$$

Any step not on the chosen execution path is assigned a time greater than the number of steps.

**5.3.5 Modelling state queries**

A state query is a function of the state changes occurring before it, while the path control points determine which state changes occur before which state queries. To achieve a correct translation from code to constraints we need to constrain each state query in our flattened list to correspond correctly to the state changes (including artificial state changes added at the beginning to set up the initial program state) and path control points.
Lookup queries

Most types of state query (including variable references) are what we call lookup queries, which means they return a value which is a function of only the most recent matching state change. What is meant by matching depends on the specific query type. For lookup queries we create a variable changeID to represent the ID of the most recent matching state change, and then constrain this ID and the retrieved value appropriately.

Previously (in Chapter 4), for field references the following constraints were used. Note that only assignments to the queried field (not other state changes) are relevant, and the expression cond gives the conditions under which assignment i is executed.

\[
\begin{align*}
\text{query} & \quad \text{qobj.field} \\
\text{changes:} & \quad \text{obj}_1.\text{field} := \text{expr}_1 \\
& \quad \vdots \\
& \quad \text{obj}_n.\text{field} := \text{expr}_n \\
\text{variables:} & \quad \text{var } 1..n: \text{changeID}; \text{var int: qresult;}
\end{align*}
\]

\[
\begin{align*}
\text{constraints:} & \quad [\text{obj}_1, ..., \text{obj}_n][\text{changeID}] = \text{qobj} \land \\
& \quad \text{qresult} = [\text{expr}_1, ..., \text{expr}_n][\text{changeID}] \land \\
& \quad [\text{cond}_1, ..., \text{cond}_n][\text{changeID}] = \text{true} \land \\
& \quad (\text{cond}_2 \land \text{qobj} = \text{obj}_2) \rightarrow \text{changeID} \geq 2 \land \\
& \quad \vdots \\
& \quad (\text{cond}_n \land \text{qobj} = \text{obj}_n) \rightarrow \text{changeID} \geq n;
\end{align*}
\]

The first constraint requires the chosen assignment to use the same object as the query, which is the definition of matching for field references. The next constraint sets the result of the query to equal the value from the chosen assignment. These two constraints are both still valid when untangling loops.

The remaining constraints are intended to ensure that the chosen assignment is executed and is not subsequently overwritten before the query. However, they assume that the changes can only occur in the listed order, which is no longer valid when loop untangling is used.

We can fix this problem by using the following alternative constraints, making use of the before predicate introduced earlier. Note that when unwinding loops, before can be defined as before(\text{step}_a, \text{step}_b) = (a < b) \land \text{cond}_a, and then these new constraints reduce to the originals.

\[
\begin{align*}
\text{query} & \quad \text{qstep: qobj.field} \\
\text{changes:} & \quad \text{step}_1: \text{obj}_1.\text{field} := \text{expr}_1 \\
& \quad \vdots \\
& \quad \text{step}_n: \text{obj}_n.\text{field} := \text{expr}_n \\
\text{variables:} & \quad \text{var } 1..n: \text{changeID}; \text{var int: changestep; var int: qresult;}
\end{align*}
\]

\[
\begin{align*}
\text{constraints:} & \quad [\text{obj}_1, ..., \text{obj}_n][\text{changeID}] = \text{qobj} \land \\
& \quad \text{qresult} = [\text{expr}_1, ..., \text{expr}_n][\text{changeID}] \land \\
& \quad \text{cstep} = [\text{step}_1, ..., \text{step}_n][\text{changeID}] \land \\
& \quad \text{before(\text{cstep}, \text{qstep})} \land \\
& \quad \text{forall } (i \text{ in } 1..n) \left( \\
& \quad \text{obj}_i = \text{qobj} \land \text{before(\text{step}_i, \text{qstep})} \right) \rightarrow \text{not before(\text{cstep}, \text{step}_i) } \\
\end{align*}
\]
We now explicitly define the basic step associated with each change and the query, and impose new constraints based on these. We first use \texttt{before} to ensure that a change which is skipped by the execution path or which occurs after the query cannot be chosen. Then the final constraint ensures that we choose the \textit{latest} matching assignment by requiring that no other matching change overwrites our chosen one.

Although these new constraints are correct, if we use the previously described definition of \texttt{before}, the resulting propagation will be weak.

\textbf{Example} Consider the reference to \texttt{currentTime} which forms part of \texttt{q4} in the routing example (Figure 5.4). The options for the \textit{latest matching change} are \texttt{c1} and \texttt{c5} (which has a version for each iteration of the loop). Imagine we are determining \texttt{q4b} and we have already decided that the \texttt{a} iteration is followed by the \texttt{b} iteration (i.e. \texttt{next[p4a] = p4b}). Ideally we should know that \texttt{q4b} can only be reached by \texttt{c5a}. But after this decision we have that \texttt{time[q4b] \in \{26, 42\}}, \texttt{time[c5a] = \{11, 27\}}, \texttt{time[c5c] = \{11, 43\}}, \texttt{time[c1] = 1}. So according to the above constraints each of \{\texttt{c1}, \texttt{c5a}, \texttt{c5c}\} could be the latest matching state change.

Even harder to handle is when we decide that iteration \texttt{b} does not follow \texttt{a}. We know that all iterations have a matching change for \texttt{q4b}. So if \texttt{a} is not immediately before \texttt{b}, then there must be another iteration in between with a matching change, even though no specific change is known to be between them.

In general, the logic we would like to have is that whenever all paths between change \texttt{c} and query \texttt{q} go through another change which is known to match \texttt{q}, then change \texttt{c} cannot be chosen for \texttt{q}. Note that although this is related to dominance it is not pure dominance as we do not require that the same change overwrites \texttt{c} on all paths.

Ideally this reasoning would be achieved using a global constraint, which could reason directly over the execution graph (the \texttt{next} variables), as well as the arguments associated with each state change and query. We investigate such a constraint (including its use when untangling as well as unwinding loops) in Chapter 7. In this chapter, as we are only considering two benchmarks, we are able to approximate the effect of the global constraint by using some of the intended logic in our model simplification phase (see Section 5.3.6). This allows us to evaluate the potential of loop untangling before implementing the global constraint.

\textbf{Aggregate queries}

In contrast to \texttt{lookup queries}, \texttt{aggregate queries} return a value which is a function of all matching state changes occurring before the query. For these we use \texttt{before} to constrain which changes should be included in the aggregate. For example, list length is constrained as follows (the length of the list is the number of matching add item changes before the query).
query: qstep: qlist.length()
changes: step1: list1.add(item1)
        ...
        stepn: listn.add(itemn)
variables: var int: qresult;
constraints: qresult = sum (i in 1..n) (bool2int(listi = qlist ∧ before(stepi, qstep)));

Sometimes a lookup query behaves like an aggregate query. This happens when the most recent relevant change itself depends on the previous changes, either because of its arguments or because earlier changes affect whether or not execution reaches the later changes. In these cases it is still possible to use the standard representation for lookup queries, but much better performance can be achieved using a specialised translation. In previous chapters this was called special cases. When untangling rather than unwinding loops we can use the same specialised translations described Chapter 4, as in each special case discussed there the query result is not affected by the order in which the changes occur.

5.3.6 Optimisations and simplifications

In certain cases it is possible to create a simple expression closest defining the conditions which must hold for change c to be the closest matching change to a given lookup query q. When this is possible, we can change the constraints used for q to the simpler form shown below.

query: qstep: var ref qobj.field
changes: step1: obj1.field := expr1
        ...
        stepn: objn.field := exprn
variables: var 1..n: changelD; var int: qresult;
constraints: [obj1, ..., objn][changelD] = qobj ∧
             [closest1, ..., closestn][changelD] ∧
             qresult = [expr1, ..., exprn][changelD];

Non-overwriting changes

If all changes potentially matching a lookup query are initialisation changes (added at the beginning to set up the initial program state), then only one can match the query, so we can use true as the closest expression for all of them.

Matching change for every predecessor

If for a query q there exists a node in the execution graph n such that every edge leading in to n would create a fixed path between a matching change c for q and q, then we can use the edges into n to define the closest conditions. For example, consider the reference to currentTime discussed previously (part of q4 in Figure 5.4). The query q<sub>4b</sub> has a matching change in each iteration of the loop (c<sub>5a</sub>, c<sub>5b</sub>, c<sub>5c</sub>), and one before the loop (c<sub>1</sub>). All edges into the node p<sub>2b</sub>
(see Figure 5.5) create a fixed path between one of these changes and \( q_{4b} \). So we can constrain \( q_{4b} \) as shown below. Note that since none of the edges leading in to \( p_{2b} \) correspond to the change \( c_{5b} \) we can discount this change immediately.

**query:** \( \text{currentTime} \)

**changes:**
- \( c_1: \text{currentTime} := 0 \)
- \( c_{sa}: \text{currentTime} := q_{4a} \)
- \( c_{sc}: \text{currentTime} := q_{4c} \)

**variables:** \( \text{var 1..3: changeID; var int: qresult;} \)

**constraints:**
- \( \left[ \text{next}[p_1] = p_{2b}, \text{next}[p_{4a}] = p_{2b}, \text{next}[p_{3c}] = p_{2b} \right] \land \text{changeID} \land \text{qresult} = [0, q_{4a}, q_{4c}] \)

The same can be done for query \( q_{10} \) outside the loop using the edges into \( p_4 \). This provides both the positive and negative reasoning discussed in the previous section. If we decide to use an edge then the closest condition for that change will become true and all others will become false (after the subpath constraint enforces the requirement that only one edge leads into a given node). If we decide not to use an edge then that closest condition will become false, excluding the corresponding change. Although these constraints are more verbose than the idealised MiniZinc shown in Section 5.2, they provide the same propagation strength.

**Ordered changes**

We can also take into account the known relationship between iterations with the same value for the label query. If all changes relevant to a query belong to iterations with the same label value, then their relative order is known (as we have fixed this to avoid symmetry), so we can use the original lookup query constraints.

If in addition all possibly matching changes are known to actually match and the path through each iteration with a matching change is fixed, then we can do even better. In this case it is not possible for execution to skip the change in iteration \( i \) without also skipping those in later iterations. So the closest matching change is the one from the last iteration to be executed before the query. Ordering the changes by iteration version, for each change \( c_i \) except the last:

\[
\text{closest}_{c_i} = \text{before}(c_i, q) \land \neg \text{before}(c_{i+1}, q)
\]

The change from the last iteration is the closest whenever it is before the query. If the query is outside the loop, then we know that all iterations not skipped by execution will occur before the query, so we can simplify the condition further:

\[
\text{closest}_{c_i} = \text{in}[c_i] \land \neg \text{in}[c_{i+1}] \quad \text{in}[\text{step}_i] = (\text{next}[\text{step}_i] \neq \text{step}_i)
\]

where \( \text{in}[\text{step}_i] \) means step \( i \) is included on the execution path. This can be defined as shown above since subpath sets unvisited nodes to have themselves as successors.
Further, for queries which are also contained in an iteration with the same label value, we can assume that all changes in earlier iterations for this value are included on the execution path. If not, then the query will not be reached either so its value does not matter. Therefore for these queries the closest matching change is set to the change from the latest iteration before the query iteration.

**Example** The above simplifications are used in the translation of the pizza example. Consider the reference to `numSlices` on line 23 of the pizza code (Figure 5.3), in the first `Veg` iteration. Let us call this query `q_{nv1}`. The relevant changes are the initialisation assignment for `Veg` which sets `numSlices` to 0, and the assignments on lines 23 and 25. All versions of these assignments from non-`Veg` iterations are known not to match `q_{nv1}` as they refer to a different pizza object, and all versions from later `Veg` iterations are known to be after this query. The versions from the current iteration are also after this query, so actually only the initialisation assignment can be chosen. Therefore the value of `q_{nv1}` can be fixed to 0. This in turn will fix the value assigned to `numSlices` on line 23 to 1, and the condition on the following line (24) to false (assuming `slicesPerPizza` is fixed to say 2), which means that the assignment on line 25 is not reached by execution.

Now consider the reference to `numSlices` in the next `Veg` iteration, query `q_{nv2}`. As explained above, since the path through the earlier `Veg` iteration is fixed and the query is also inside a `Veg` iteration, we can simply use the change from the closest iteration to this one (V1). The value of `q_{nv2}` is therefore 1. The value assigned on line 23 will be 2, and the test on line 24 will succeed. Again the path through this iteration is fixed, but it goes through the assignment on line 25, setting `numSlices` back to zero. When we consider `q_{nv3}` applying the same simplification again gives a value of 0.

For the query to `Veg.numSlices` after the loop (inside the `totalCost` method), we can use the definition of closest described above for queries outside the loop, because all matching changes belong to `Veg` iterations (except for the init change) and are known to match, and the path through every `Veg` iteration is fixed. The constraint for this query is therefore:

- **query:** `q`: Veg.numSlices
- **changes:**
  - `c_0`: Veg.numSlices := 0
  - `c_{1..4}`: Veg.numSlices := 1
- **variables:** `1..4: changeID, var int: qresult`
- **constraints:**
  - `¬in[c_{1+1}, in[c_{1+1}] ∧ ¬in[c_{2+2}, in[c_{2+2}] ∧ ¬in[c_{1+1}, in[c_{1+1}][changeID]] changeID] ∧ qresult = [0,1,0,1][changeID];`

This is the constraint described in Section 5.2 which links the final value of `Veg.numSlices` to the number of times `Veg` is chosen (which is `changeID - 1`).

102
5.4 Experimental Results

We show here experimental results for the two examples discussed. The constraint models for unwinding and untangling are produced fully automatically from the input Java code, the only additional information given to untangling was the choice of label query for each loop. Table 5.1 compares untangling to unwinding (the version called new+ in Chapter 4) and to a hand written CP model for the same problem (also identical to the one used previously). Each figure is the average for 30 instances of the stated size. The times shown include instances which reached the timeout of 10 minutes, while the failures figures (shown in thousands) exclude them. All models were solved using G12 CPX on a 3.40GHz Intel i5-4670K with 16GB RAM.

The results clearly show the benefit of untangling. For these problems, we are vastly better off using a simple model for each iteration and deciding the order through them, rather than deciding what happens in the $i^{th}$ loop iteration where we know the order. Note that the simplifications and optimisations discussed in Section 5.3.6 are crucial to achieve these results, and cannot always be applied. We show in Chapter 7 that with a specialized global propagator we can achieve good results without relying on these optimisations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solving Time</th>
<th>Failures (000s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>unwind</td>
<td>untangle</td>
</tr>
<tr>
<td>pizza 4</td>
<td>94.0s</td>
<td>1.3s</td>
</tr>
<tr>
<td>5</td>
<td>320.7s (13)</td>
<td>5.8s</td>
</tr>
<tr>
<td>6</td>
<td>470.1s (22)</td>
<td>149.7s (6)</td>
</tr>
<tr>
<td>routing 5</td>
<td>12.9s</td>
<td>1.5s</td>
</tr>
<tr>
<td>6</td>
<td>102.8s</td>
<td>8.1s</td>
</tr>
<tr>
<td>7</td>
<td>569.3s (20)</td>
<td>40.8s</td>
</tr>
</tbody>
</table>

Table 5.1: Comparative performance of unwinding and untangling.

5.5 Related and Further Work

Loop untangling is related to other forms of program analysis that reason about loops. For example, automatic parallelisation of code needs to reason about when iterations can be reordered [91]. We could improve loop untangling by co-opting methods from this area to detect cases where the execution order of iterations can be fixed arbitrarily. The technique described in [6] for detecting commutativity could be a good starting point as a similar query-based viewpoint is taken when considering whether or not reordering iterations changes the outcome.

Loop untangling could also be improved by employing more general forms of program analysis. Typically optimisations performed by compilers are designed to simplify the remaining code, which would in turn simplify our translation to the constraint model. For example, loop untangling implicitly requires
reasoning about reaching definitions, and can also be simplified by constant propagation. While our tool does a basic form of reaching definition analysis and constant propagation these could be improved by full program analysis techniques (e.g. [1]).

An interesting direction for future work is developing a program analysis which would automatically select the label query. By examining the reaching definitions graph and understanding what data in the program is dependent on decisions and what is not we can choose a label query that, when fixed, fixes much of the computation of the loop body. We would however need to trade this off against the number of iterations required.

Finally our method, and indeed most methods based on symbolic execution, currently only handles bounded loops. Unbounded loops can be approximated by putting an artificial limit on the number of iterations, but otherwise they require techniques to generate loop invariants including interpolation [36] or abstract interpretation [35]. Loop untangling could possibly be extended to handle unbounded loops using an approach similar to that in [38]. There constraints were added for each iteration of the loop lazily as needed when it became known that the previous iteration was entered. In a similar fashion, it may be possible to lazily add nodes to our execution path graph.

5.6 Conclusion

Standard loop unwinding unnecessarily ties the actual execution order of iterations with the way an individual iteration is identified. The idea behind loop untangling is to decouple these two things by modelling the execution path explicitly. The labelling scheme can then be used to reduce the uncertainty in each copy of the loop body. Although it may be necessary to create more copies of the loop body than through standard loop unwinding, with a good choice of labelling scheme this is far outweighed by the relative simplicity of the constraints required for each copy. The final result is a model which is much easier to solve.
Chapter 6
Explaining Circuit Propagation

6.1 Introduction

The circuit($\vec{v}$) global constraint requires the variables in its argument list $\vec{v}$ to take values such that each variable’s value indicates the index of its successor in a tour visiting all variables. If we consider the graph $G$ with a vertex $u_i$ for each variable $v_i$ and edges $(u_i, u_j)$ where $j$ is in the domain of $v_i$, a solution to the circuit constraint is a Hamiltonian cycle of $G$. The circuit constraint is a special case of the cycle constraint [59]. The constraint cycle($n, \vec{u}$) holds if each node in $\vec{u}$ has a distinct successor, and the number of different (including self) cycles is $n$, hence circuit($\vec{v}$) = cycle(1, $\vec{v}$).

The circuit constraint has a number of closely related variants: path looks for a Hamiltonian path in a graph; subcircuit looks for a simple cycle in a graph not including all nodes; while subpath looks for a path in the graph. The propagation algorithms for these constraints are highly related (and indeed we can use the propagators for circuit and subcircuit for path and subpath respectively). As discussed in the previous chapter, we require a subpath constraint in order to support loop untangling. Circuit and its variations are also important for various other kinds of tour-finding, path-finding and graph problems.

In this chapter we examine how to integrate the circuit constraint, and its variants, into a lazy clause generation solver.\(^1\) To do so we must extend the constraint to explain its propagation. We consider various propagation algorithms for circuit and examine how best to explain each of them. We also compare the effectiveness of the different propagation algorithms once we use explanation, since adding explanation changes the trade-off between propagation complexity and power. Simpler propagators, although less powerful, may produce more reusable explanations. Even though the most powerful propagation algorithm

\(^1\)See Section 1.2 for an introduction to propagation-based constraint solving and lazy clause generation.
we consider for \textit{circuit} and its variants creates huge explanations, we find that explanation is highly advantageous for solving problems involving this kind of constraint.

Clearly the definition of \textit{circuit} implies that the \textit{alldifferent} constraint must also hold for \textit{circuit} to be satisfied, since every variable must have a different successor in the circuit. Propagators for the \textit{circuit} constraint typically re-use \textit{alldifferent} propagation algorithms, using a separate algorithm to prevent subtours. For the purpose of this chapter we focus on this circuit-specific part of the propagation, as \textit{alldifferent} propagation is already well studied, including an investigation into its interaction with lazy clause generation (see [110] and [69, 39]).

The next section discusses modelling with \textit{circuit} and introduces the example used for experiments. Section 6.3 discusses variants on \textit{circuit} and their relationship with \textit{circuit}. Section 6.4 introduces different propagators for \textit{circuit} and its variants, and explores how best to explain the resulting propagation. In Section 6.5 we examine the trade-off between propagation complexity and pruning strength, and compare the propagators with and without explanation. Section 6.6 discusses other constraints related to \textit{circuit}. Finally Section 6.7 concludes.

6.2 Modelling with \textit{circuit}

The \textit{circuit} constraint has a Zinc/MiniZinc \cite{zinc, minizinc} definition of the form

\begin{verbatim}
predicate circuit(array[int] of var int: succ);
\end{verbatim}

where the array of integer variables \texttt{succ} represents nodes in a graph numbered from 1 to \(n\), where \(n\) is the length of the array. The initial domain of variable \texttt{succ}[i] represents the possible successors of node \(i\), and hence is a subset of \(1..n\). The constraint ensures that the edges represented by the values of the successor variables form a Hamiltonian cycle of the nodes, that is a complete circuit visiting every node once.

Consider the problem of designing a tour of a set of locations. We imagine a tour company has selected important sites which should be included in the tour, and wishes to find an order to visit the sites so that each site is visited exactly once, and the length of the longest leg is minimised (as their clients do not like sitting in a bus for a long time). We assume a transport network which is not complete. That is, it is not always possible to travel from one site to another directly, without passing through one or more other sites. Even passing through an already visited site is not allowed, as this would be frustrating to clients.

A MiniZinc \cite{minizinc} model for this problem is shown in Figure 6.1. The only global constraint is the \textit{circuit} constraint. All other constraints are (reified) binary constraints. We shall use this model to explore design alternatives for \textit{circuit} and its variants. Since it is dominated by the global \textit{circuit} constraint, it is an effective model for such evaluation.
The underlying graph used for the transport network is important. Completely random graphs do not make very realistic transport networks, because there is no consistency in the distances between nodes, and in which nodes are connected to each other. We have used a more realistic technique to generate networks for our benchmarks. We first randomly distributed the locations in a two dimensional space, and calculated the Euclidean distance between each pair. Edges were then added so that every node was connected to its seven closest neighbours. To keep the instances more similar in difficulty we then added extra edges to ensure the existence of at least one Hamiltonian circuit (to make the instance satisfiable). We achieved this by performing a random walk of the graph, adding a randomly chosen new edge whenever all existing edges from the current node lead to already visited nodes, and then adding an edge from the final node back to the initial node. All data files used in our experiments are available online.\footnote{http://people.eng.unimelb.edu.au/pstuckey/circuit}

```plaintext
include "circuit.mzn";
int: n; % number of locations
set of int: Locations = 1..n;
int: maxLegLen; % length of longest edge in network

% travel times between locations
% -1 means no direct connection exists
array[Locations,Locations] of int: travelTime;

% successor variables
array[Locations] of var Locations: succ;

% only use allowed legs
constraint forall(loc1, loc2 in Locations)
  ( travelTime[loc1,loc2] < 0 -> succ[loc1] != loc2 );

% successors must form a circuit
constraint circuit(succ);

% variable for the length of the longest leg
var 1..maxLegLen: maxleg;
constraint forall(loc1, loc2 in Locations)
  ( succ[loc1] == loc2 -> maxleg >= travelTime[loc1,loc2] );

solve minimize maxleg;

Figure 6.1: MiniZinc model for tour design problem
```
6.3 Variations of circuit

In this work we consider not only the circuit constraint, but close variations, each of which is introduced below along with a corresponding variation to our example problem. The propagation algorithms for all variants are similar. Figure 6.2 provides an illustration of solutions satisfying each variation of the constraint.

6.3.1 The path constraint

The circuit constraint can be trivially extended to implement a path constraint. The path constraint ensures that when the value of each variable is interpreted as the index of its successor, the result is a single path including all variables.

The path constraint is defined as

\[
\text{predicate path(array[int] of var int: succ, var int: start, var int: end) =}
\]

\[
\text{circuit(succ ++ [start])} \land
\text{succ[end] = length(succ)+1;}
\]

where the succ array is a successor relationship, start is the number of the first node in the path, and end is the number of the last node in the path. Each variable ranges over 1..n+1 where n is the length of the array succ. The successor relationship defines a single path from node start to node end. This is achieved by adding a dummy node (with index n+1) to the graph. Its successor, given by variable start, is added at the end of the array of original variables (using MiniZinc syntax succ ++ [start]), and we then apply the standard circuit constraint. In a solution, the value of the start variable gives the start of the path, and the successor of the final node in the path end is the dummy node (so index n+1, as this is the index of the start variable).

Note that this path constraint is a special case of the path constraint listed in the global constraints catalog [59]. The more general constraint also accepts an argument which is the number of paths required (in our case we assume 1).

![Figure 6.2: Solutions to variations of the circuit constraint.](image)
Returning to our tour design example, the path constraint is applicable when the tour is not required to start and finish in the same location. The start and end locations for the tour may be unconstrained, they may be fixed, or there may be a limited number of options (e.g. cities with international airports). For simplicity we consider the case where the start and end locations are completely open, so the MiniZinc model is exactly the same, except that the call to circuit is replaced with a call to path with start and end unconstrained, and the domain of the succ variables extended by one.

6.3.2 The subcircuit constraint

The circuit constraint is only applicable when all variables are required to participate in the circuit. We next consider a variation of circuit, which we call subcircuit, which drops this requirement.

The subcircuit constraint has a Zinc/MiniZinc [78, 86] definition of the form

\[
\text{predicate subcircuit(array\[int\] of var int: succ);
}
\]

where the argument succ is the same as for circuit.

In subcircuit, a variable excluded from the circuit is required to take its own index as its value, hence it in effect appears in a self cycle. This means that the alldifferent constraint still holds for subcircuit and prevents a variable not included in the circuit from becoming the successor of another variable, since a variable v having its own index as a value prevents any other variable having v as a successor.

Changes are required to the circuit propagation algorithms (and therefore also the explanations generated) to make them applicable for subcircuit. These will be discussed in Section 6.4 where we describe propagation algorithms.

The subcircuit constraint applies to our tour design problem when instead of visiting every location in the network, a tour is required to visit some subset of the locations satisfying further constraints. For example, say we have a set of activities required to be included in the tour (e.g. shopping, beach visit, museum), and each activity is only available in some locations. The task is to find a tour covering at least one location providing each activity, while minimising the length of the longest leg. This is the problem we will use to evaluate subcircuit propagation and explanation. Our MiniZinc implementation is provided in Figure 6.3.

6.3.3 The subpath constraint

The subpath constraint is an extension of subcircuit equivalent to the path extension for circuit. The successor variables are required to form a path, and any node not in the path must have itself as its successor. As with path, subpath is implemented by adding a dummy node n + 1 with successor variable start, and then applying the subcircuit constraint. For subpath we also need to ensure that the dummy node is included in the circuit. This is easily achieved by limiting the domain of the start variable to only the original nodes.
The \textit{subpath} constraint is defined analogously to the \textit{subcircuit} constraint.

\texttt{predicate subpath(array[int] of \texttt{var int}: succ, \texttt{var int}: start, \texttt{var int}: end) = subcircuit(succ ++ [start]) /\}
\texttt{start <= length(succ) /\}
\texttt{succ[end] = length(succ)+1;}

For a version of the tour design problem appropriate for \textit{subpath}, we again simply drop the requirement that the tour must start and finish in the same location.

\texttt{include "subcircuit.mzn";}\n\texttt{int: n; \% number of locations} \n\texttt{set of int: Locations = 1..n;}\n\texttt{int: m; \% number of activities} \n\texttt{set of int: Activities = 1..m;}\n\texttt{int: maxLegLen; \% length of longest edge in network} \n\texttt{\% travel times between locations} \n\texttt{\% -1 means no direct connection exists} \n\texttt{array[Locations,Locations] of \texttt{int}: travelTime;} \n\texttt{\% activity locations} \n\texttt{array[Activities,Locations] of \texttt{bool}: activityAvailable;} \n\texttt{\% successor variables} \n\texttt{array[Locations] of \texttt{var Locations}: succ;} \n\texttt{\% only use allowed legs} \n\texttt{constraint forall(loc1, loc2 in Locations)} \n\texttt{( travelTime[loc1,loc2] < 0 -> succ[loc1] != loc2 );} \n\texttt{\% visit at least one location with every activity} \n\texttt{constraint forall(act in Activities)} \n\texttt{( exists(loc in Locations where activityAvailable[act,loc])} \n\texttt{ ( succ[loc] != loc ) );} \n\texttt{\% successors must form a circuit} \n\texttt{constraint subcircuit(succ);} \n\texttt{\% variable for the length of the longest leg} \n\texttt{var 1..maxLegLen: maxleg;} \n\texttt{constraint forall(loc1, loc2 in Locations)} \n\texttt{( succ[loc1] == loc2 -> maxleg >= travelTime[loc1,loc2] );} \n\texttt{solve minimize maxleg;} \n
Figure 6.3: MiniZinc model for tour design problem using \textit{subcircuit}.  

110
6.4 Propagating and Explaining \textit{circuit}

In the following sections we consider three complementary algorithms for propagating \textit{circuit}, in order of increasing complexity. For each we define the algorithm and then examine various alternatives for adding explanations, using experimental results to justify our final decisions.

In order to reduce the risk of making design decisions which are only justified for a specific search strategy, the experiments are repeated using two different search strategies. The first is in-order labelling of variables using minimum values first. This simple fixed strategy allows us to more easily isolate the effect of different design decisions, as the interaction between the design decision and the search strategy is relatively easy to understand. It also allows us to compare versions of the propagator with and without learning. The second strategy is activity based dynamic search. This is the most effective autonomous search for lazy clause generation solvers, and is highly dynamic.

All experiments in this chapter were carried out on a 2.8GHz AMD 6-Core Opteron 4184 CPU with 64GB of memory, using (except where otherwise stated) the lazy clause generation solver Chuffed [32].

Throughout this section we use the following notation: \( V \) is the set of all nodes (or variable indices), \( n = |V| \), \( x_i \) where \( i \in V \) is the variable holding the successor of node \( i \), and \( value(x_i) \) is the (fixed) value of variable \( x_i \) in the current domain \( D \).

Note that all of the algorithms discussed below assume the \textit{alldifferent} constraint has already been propagated. We use the existing implementation of domain consistent \textit{alldifferent} for all experiments. Since it is not possible to explain only some propagations in Chuffed, whenever explanations are used for \textit{circuit} they are also used for \textit{alldifferent} (and all other constraints in the model).

6.4.1 The \textit{check} algorithm

A very simple and cheap propagator for the \textit{circuit} constraint fires only on variable fixing, and simply follows the chain of fixed variables starting at this newly fixed variable, until an unfixed variable is found, or a loop is detected. If a loop is detected with length less than the number of nodes, the propagator reports a conflict. We call this propagation algorithm \textit{check}.

For \textit{subcircuit} (and \textit{subpath}), a cycle which excludes some nodes is allowed if all excluded nodes have themselves as successor. Therefore when a cycle of fixed variables is found in the \textit{subcircuit} version of the \textit{check} algorithm, instead of reporting failure we set the value of all excluded variables to their own indices. If this is not possible for some variable, then a conflict is reported.

Explanations for \textit{check}

We consider two alternative explanations for \textit{check} propagation, shown below. Here \( C \) is the set of nodes included in the small cycle.
\[ \bigwedge_{i \in C} [x_i = \text{value}(x_i)] \rightarrow \text{false} \quad (6.1) \]

\[ \bigwedge_{i \in C, j \in V \setminus C} [x_i \neq j] \rightarrow \text{false} \quad (6.2) \]

Clause 6.1 says that the successor variables for nodes in the cycle taking their current values leads to failure. The second option is more general (but also larger - \(O(n^2)\) rather than \(O(n)\)), indicating that the fact that no node inside the cycle has a successor outside the cycle is sufficient to cause failure.

For \textit{subcircuit} we can use very similar clauses. The two options are shown below.

\[ \bigwedge_{i \in C} [x_i = \text{value}(x_i)] \rightarrow [x_k = k], \quad k \in V \setminus C \quad (6.3) \]

\[ \bigwedge_{i \in C, j \in V \setminus C} [x_i \neq j] \rightarrow [x_k = k], \quad k \in V \setminus C \quad (6.4) \]

Table 6.1 shows experimental results comparing these two alternatives for each variation of \textit{circuit}. It seems clear from the table that the second alternative (clauses 6.2 and 6.4) is better for all variations of our problem, as for both search strategies the number of failures, number of propagations, and execution time are all much smaller. This result demonstrates that general clauses can be more effective than smaller, more specific clauses. We use clauses 6.2 and 6.4 in the remainder of our experiments.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Clause</th>
<th>Inorder Search</th>
<th>VSIDS Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fails</td>
<td>Props</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{circuit}</td>
<td>1</td>
<td>2071</td>
<td>14567</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>399</td>
<td>2088</td>
</tr>
<tr>
<td>\textit{subcircuit}</td>
<td>3</td>
<td>5179</td>
<td>39758</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>657</td>
<td>3495</td>
</tr>
<tr>
<td>\textit{path}</td>
<td>1</td>
<td>2301</td>
<td>15584</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>505</td>
<td>2993</td>
</tr>
<tr>
<td>\textit{subpath}</td>
<td>3</td>
<td>4606</td>
<td>28863</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>871</td>
<td>4297</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison of alternative explanation clauses for the \textit{check} algorithm. Each figure is the average for 500 instances with 50 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.
Choosing an explanation for failure

When not using explanation, it makes sense to exit a propagator as soon as failure is detected. However, when using lazy clause generation the explanation we give for failure will affect the clause which is subsequently learned. Therefore if a constraint is violated in multiple ways it may be worth discovering all of these and using a heuristic to choose which violation to report.

For subcircuit, when a small cycle is found we report failure if there exists a node outside that cycle which cannot be a self cycle (that is, a node whose successor variable’s current domain does not include its own index). By default we have reported failure using the first such node encountered. However, it is possible to instead collect all nodes for which this condition holds, and then make a deliberate selection.

The clause we produce to explain subcircuit check conflicts is shown below, where \( C \) is the set of nodes in the cycle, and \( k \) is the chosen node outside the cycle.

\[
\left( \left[ x_k \neq k \right] \land \bigwedge_{i \in C, j \in V \setminus C} \left[ x_i \neq j \right] \right) \rightarrow false
\]

Note that for each node we could choose, the clause will include a different first literal but will otherwise be the same. We can use the properties of the corresponding literals to make a good choice for \( k \).

We know that this clause will cause a conflict, and at that point the solver will compute a learned clause and backjump to some earlier part of the search tree. The position in the search tree we jump to will depend on the levels at which literals involved in the failure became fixed. We would like to backjump as far as possible, as this way we exclude more of the search tree. Therefore it makes sense to choose to include in our explanation of failure the literal which became fixed highest (earliest) in the search tree. This information is already available as it is used when deriving learned clauses.

We tested this theory experimentally, comparing four different selection heuristics for the node to be used to explain a subcircuit check conflict:

1. The first applicable node. Note that this option entails less overhead as we can report failure immediately upon finding an appropriate node.
2. The last applicable node.
3. The node \( k \) whose corresponding literal \( x_k \neq k \) became fixed highest in the search tree. In other words, the node whose successor as itself was excluded the earliest.
4. The node \( k \) whose corresponding literal became fixed lowest in the search tree (the node whose successor as itself was excluded most recently).
The results are shown in Table 6.2. As expected, choosing the literal which was fixed highest (earliest) in the search tree is the best option for both subcircuit and subpath, and choosing the literal which was fixed lowest in the tree is worse than all other options for both problems.

When using inorder search for subcircuit, choosing the first applicable node is better than choosing the last. This is probably because an inorder search means successor variables for nodes earlier in the list will often become fixed higher in the search tree.

For subpath, choosing the last node works quite well (and better than choosing the first). This is surprising until you realise that the last node in the list is the dummy node (start), which is never allowed to have itself as a successor. This means that whenever a small cycle is found which does not include the dummy node, choosing the last node gives an ideal explanation for the failure, because the corresponding literal is fixed at the root node.

In all further experiments, we use the highest node heuristic to choose a literal for subcircuit and subpath failure explanations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Heuristic</th>
<th>Fails</th>
<th>Props</th>
<th>Time (sec)</th>
<th>Fails</th>
<th>Props</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>subcircuit</td>
<td>first</td>
<td>1220</td>
<td>6564</td>
<td>274.5 (159)</td>
<td>7</td>
<td>82</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>last</td>
<td>1259</td>
<td>6733</td>
<td>430.7 (288)</td>
<td>7</td>
<td>82</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td>1183</td>
<td>6030</td>
<td>481.0 (330)</td>
<td>16</td>
<td>139</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>highest</td>
<td>922</td>
<td>5737</td>
<td>212.8 (123)</td>
<td>5</td>
<td>70</td>
<td>1.0</td>
</tr>
<tr>
<td>subpath</td>
<td>first</td>
<td>1095</td>
<td>5541</td>
<td>501.2 (359)</td>
<td>9</td>
<td>108</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>last</td>
<td>1041</td>
<td>5325</td>
<td>458.0 (321)</td>
<td>5</td>
<td>68</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td>1189</td>
<td>5488</td>
<td>561.2 (430)</td>
<td>21</td>
<td>174</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>highest</td>
<td>922</td>
<td>5419</td>
<td>397.3 (260)</td>
<td>4</td>
<td>59</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 6.2: Comparison of alternative heuristics for choosing an explanation of failure detected by the check algorithm for subcircuit and subpath. Each figure is the average for 500 instances with 55 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

6.4.2 The prevent algorithm

We now consider a slightly stronger propagation algorithm, described in [30], which we call prevent. This algorithm finds the start and end of each chain of nodes with fixed successors, and removes the first node of each chain from the domain of the successor variable for the end node of that chain (unless the chain includes all variables). This prevents the chain from becoming a subcycle.

The circuit propagator in the open source constraint solver Gecode (3.5.0) [97] uses a clever technique to find distinct chains, exploiting the fact that a chain must start with a node which is not the fixed value of any other successor.
variable. Assuming that \textit{alldifferent} has been propagated, the set of all possible starts of chains is the union of the domains of the unfixed successor variables. We used this method in our implementation as well, with the only drawback being that because chains that are already cycles are not explored, this algorithm is not complete and must be accompanied by either the \textit{check} algorithm from above, or the stronger algorithm described in the next section. In our experiments, the \textit{prevent} algorithm is always accompanied by \textit{check}.

\textbf{Applying prevent to subcircuit}

The \textit{subcircuit} version of the \textit{prevent} algorithm cannot perform any propagation unless there exists outside the chain a node \( k \) which must be included in the circuit (because its successor variable’s current domain does not include itself). We call this node the \textit{evidence} node, and for \textit{evidence} node \( k \) we refer to \( x_k \) as the \textit{evidence} variable, and \([x_k \neq k]\) as the \textit{evidence} literal.

\textbf{Explaining prevent}

For the \textit{circuit} version of \textit{prevent} we again have two different options for explanations. In the following, \( C \) is the set of nodes in the fixed chain, \( a \) is the first node, and \( z \) is the last node in the chain (so its successor variable is not fixed).

\begin{align*}
\bigwedge_{i \in C, i \neq z} [x_i = \text{value}(x_i)] \rightarrow [x_z \neq a] & \quad (6.5) \\
\bigwedge_{i \in C, i \neq z, j \in V \setminus C} [x_i \neq j] \rightarrow [x_z \neq a] & \quad (6.6)
\end{align*}

The first clause (6.5) says that the last node is not allowed to have the first node as a successor because of the current choice of successor for the other nodes in the chain. The second clause (6.6) says that the last node is not allowed to have the first as a successor because none of the other nodes in the chain have a possible successor outside the chain. If none of the other nodes lead outside the set included in the chain, then the final node must do so, and therefore it cannot have the first node in the chain as a successor.

For the \textit{subcircuit} version of the algorithm, we need to also specify that one of the nodes outside the chain cannot have itself as a successor. The two possible explanations are therefore the same as above but with an extra literal on the left hand side \([x_k \neq k]\), where \( k \) is a node outside the chain. This is the \textit{evidence} literal mentioned previously.

Clauses 6.5 and 6.6 have the same size complexity as the \textit{check} explanation clauses - \( O(n) \) and \( O(n^2) \) respectively.

The results of experiments evaluating the two options are shown in Table 6.3. In contrast with the results for \textit{check} propagation, this time the second clause

115
was not more effective. Instead for all problems and for both search strategies the first clause (6.5) resulted in roughly equal or slightly better performance.

Although the clauses presented for prevent seem analogous to the check clauses, the second prevent explanation (6.6 above) is less general than the corresponding check explanation would be if the chain was later closed into a circuit. The prevent explanations both refer specifically to the final successor variable being equal to the first node, whereas only the first check explanation does this. The second check explanation would simply state that no node inside the cycle reaches any node outside. As stated previously we always use prevent in combination with check as it is not complete on its own, and in these experiments we used the best discovered options for check, which means the more general explanations. Perhaps this is why the more general explanation for prevent did not perform all that well.

Since there was very little difference between the two options, and the first option is simpler, we chose to use the first option (Clause 6.5) in all further experiments.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Clause</th>
<th>Inorder Search</th>
<th>VSIDS Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fails Props Time</td>
<td>Fails Props Time</td>
</tr>
<tr>
<td>circuit</td>
<td>6.5</td>
<td>354 2434 72.8 (50)</td>
<td>0.3 11.8 0.25</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
<td>316 2148 75.0 (50)</td>
<td>0.3 12.0 0.28</td>
</tr>
<tr>
<td>path</td>
<td>6.5</td>
<td>533 4402 212.0 (145)</td>
<td>0.4 18.2 0.37</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
<td>490 3960 220.1 (151)</td>
<td>0.4 18.5 0.44</td>
</tr>
<tr>
<td>subcircuit</td>
<td>6.5</td>
<td>677 5250 166.6 (86)</td>
<td>1.5 38.3 0.55</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
<td>708 5458 181.4 (94)</td>
<td>1.5 38.5 0.56</td>
</tr>
<tr>
<td>subpath</td>
<td>6.5</td>
<td>863 6262 377.8 (236)</td>
<td>1.2 36.7 0.67</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
<td>835 5935 384.3 (236)</td>
<td>1.2 37.0 0.77</td>
</tr>
</tbody>
</table>

Table 6.3: Comparison between alternative explanation clauses for the prevent algorithm. Each figure is the average for 500 instances with 55 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

Choosing an evidence literal

For the subcircuit version of the prevent algorithm, we need to choose an evidence literal to include in our propagation explanation. We experimented with the same four options used for choosing a literal to include in check failure clauses. That is, the first appropriate literal, the last, the literal fixed highest (earliest) in the search tree, and the literal fixed lowest (most recently) in the search tree. The results are shown in Table 6.4.

Although the differences are much smaller than those observed in the check case, we again find that the best option is to choose the literal highest in the
search tree. Recall that when using inorder search we expect the first and highest options to perform similarly. In future experiments we use the highest selection method to choose evidence literals for prevent explanation.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Heuristic</th>
<th>Inorder Search</th>
<th>VSIDS Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fails</td>
<td>Props Time</td>
<td>Fails Props Time</td>
</tr>
<tr>
<td>subcircuit</td>
<td>first</td>
<td>677 5250 166.6 (86)</td>
<td>1.5 38.3 0.55</td>
</tr>
<tr>
<td></td>
<td>last</td>
<td>708 5375 172.2 (92)</td>
<td>1.5 37.9 0.53</td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td>706 5470 181.2 (92)</td>
<td>1.5 38.1 0.54</td>
</tr>
<tr>
<td></td>
<td>highest</td>
<td>668 5211 166.5 (85)</td>
<td>1.4 37.7 0.53</td>
</tr>
<tr>
<td>subpath</td>
<td>first</td>
<td>863 6262 377.8 (236)</td>
<td>1.2 36.7 0.67</td>
</tr>
<tr>
<td></td>
<td>last</td>
<td>878 6351 392.6 (243)</td>
<td>1.2 36.9 0.66</td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td>893 6348 399.6 (244)</td>
<td>1.3 37.3 0.67</td>
</tr>
<tr>
<td></td>
<td>highest</td>
<td>868 6233 375.8 (240)</td>
<td>1.1 36.2 0.64</td>
</tr>
</tbody>
</table>

Table 6.4: Comparison of alternative heuristics for selecting an evidence literal for subcircuit prevent explanations. Each figure is the average for 500 instances with 55 locations. Failure and propagation counts are given in thousands, while times are in seconds. The number of timeouts (10 mins) is shown in brackets.

### 6.4.3 The scc algorithm

As mentioned previously, a solution to the circuit constraint is a Hamiltonian cycle of the graph $G$ where each node $v$ has an edge to all nodes $u \in D(x_v)$. This implies circuit is only feasible if $G$ has a single strongly connected component.

The final propagation algorithm we consider, which we call scc, takes advantage of this property. The algorithm is based on Tarjan’s depth first search algorithm for finding strongly connected components [106]. If multiple strongly connected components are discovered, the algorithm reports failure. With only minor changes to Tarjan’s algorithm, it is also possible to discover further propagation in some cases. As discussed in [99], depending on the current domains of variables and the root node chosen, depth first search may explore multiple disjoint subtrees below the root. Figure 6.4 shows an example of this, with nodes numbered in the order they are visited, and the subtrees shown as triangles.

When this happens, reasoning can be applied to prune links between subtrees which cannot form part of a valid solution, and to enforce links which must be part of any solution.

### Propagation rules

The following two observations are made in [99].

1. There must be an edge from each subtree to its predecessor subtree, and an edge from the first subtree to the root. Hence no edge to the root from a subtree other than the first can be used.
2. Edges between non-adjacent subtrees are not allowed. That is, if A, B and C are subtrees such that A was visited before B and B was visited before C, then any edge that leads from C to A is forbidden, because if such an edge were used in the circuit there would be no way to get in and out of B without visiting the root twice.

We add two more observations, both of which provide further opportunity for propagation with only minor alterations to the algorithm.

3. Any solution must include an edge leading from the root node to the last subtree (as there is no other way to reach this subtree). Therefore, since a solution can only include one edge originating at the root node (because the root successor variable can only take one value), any edge leading from the root to earlier subtrees can be pruned.

4. At any node x within the search tree, if exploration of x’s first child a does not reach any node above x, then the edge leading from x to a can be pruned. This is because the only way out of the subtree rooted by a is through x, so the circuit must not enter this subtree through x (as x cannot be visited twice). The circuit must instead enter the subtree rooted at a via a back edge from a later subtree.

In the remainder of this chapter we will refer to these new pruning rules as prune root (rule 3) and prune within (rule 4). Figure 6.5 illustrates all four rules.

Algorithm description

We now describe the scc algorithm. Pseudo-code for this algorithm is included in Figure 6.6. The key idea is to keep track of the search index of the first and last node in the previous subtree explored, in order to be able to detect for every back edge found whether the destination is within the current subtree, in the
Figure 6.5: (a) The SCC exploration graph for circuit starting from root. At least one (thick) edge from A to the root, from D to C, C to B, and B to A must exist (rule 1). Backwards (dotted) edges to the root from B, C or D cannot be used (rule 1). The (thin-dashed) edges from C to A and D to B cannot be used (rule 2). The (thick-dashed) edges leading from root to A, B and C cannot be used (rule 3). (b) Illustration of prune-within (rule 4). The edge from x to a cannot be used otherwise we cannot escape the subtree rooted at a (dark grey).

previous subtree, or in an earlier subtree. Any edge to an earlier subtree can be pruned. Edges to the previous subtree are counted and the most recently found is stored. After a subtree has been explored a conflict is reported if there were no back edges, and if there was only one then this edge can be enforced. Removal of edges leading from the root to subtrees before the last is done after exploration is complete, while prune within is handled by testing the lowpoint of the first child at each node, and if this lowpoint is not less than the index of the current node, the edge to that child is pruned. Obviously if a strongly connected component is found below the root, or if the search does not reach all variables, a conflict is reported.

**Adjusting the scc algorithm for subcircuit**

Converting the scc algorithm to apply to subcircuit is a little more difficult than for the previous two algorithms. Much of the reasoning for this algorithm depends on the fact that every subtree must be visited. Our general approach is to perform the algorithm as per circuit, but before a propagation is performed or conflict reported, an extra checking step is required as follows.

- For subcircuit, it is not necessarily true that there must be an edge from each subtree to its predecessor subtree. This is only true if both of these subtrees are required to be included in the circuit (at least one node from each). Similarly, we only require an edge from the first subtree to the root if there exist nodes both inside and outside that first subtree that must
Figure 6.6: Pseudo-code for the scc propagation algorithm.
be in the circuit. Otherwise it doesn’t matter if there is no way out of the first subtree. So in each of these situations, before pruning or fixing any edges we ensure that we can find an evidence node (whose successor variable’s current domain does not include itself) within these subsets of the nodes.

- The rule concerning edges skipping subtrees is similarly qualified. Such edges are only prohibited if the skipped subtrees contain a node which is required to be included in the circuit. Note that there is no need to ensure that the origin and destination subtrees of the edge in question are required to be included. If either of these subtrees is not included, then no edge between them can be used.

- Edges from the root to subtrees before the last can be allowed if no node in the last subtree is required to be included in the circuit. It would be possible to extend this to say that an edge from the root to a subtree can be pruned if any of the later subtrees contains a node which must be included in the circuit, but we did not implement that extra reasoning.

- At a node \( x \) within the search tree, if exploration of \( x \)’s first child \( a \) does not reach any node above \( x \), then the edge from \( x \) to \( a \) can be pruned only if nodes both inside and outside the part of the tree rooted at \( a \) are required to be included in the circuit.

- If the search discovers a strongly connected component below the root, a conflict is reported only if there exist nodes both inside and outside that component which must be included in the circuit.

- If the search does not reach all nodes, instead of reporting a conflict, we first check that at least one reached node is required to be included in the circuit, and if such a node can be found we fix all successor variables of nodes not reached by the search to form self-cycles.

The only other change required to the algorithm is that self-cycle edges must be handled carefully. These edges are ignored when finding the children of a node, and edges from the root to itself are not removed as prune root propagations.

**Explaining scc propagation**

In this section we provide the explanation clauses used for scc propagation. All of the explanations for this algorithm have size complexity \( O(n^2) \), as they all include at least one statement that no (or only one) edge exists between two subsets of the nodes.

When discussing subcircuit we use the notation \( \text{in}(a) \) to mean that node \( a \) must be included in the circuit, making it a possible evidence node (i.e. \( a \not\in D(x_a) \) for the current domain \( D \)).
There are several different propagation rules to consider.

1. A strongly connected sub-component exists.
   For circuit, on discovery of a strongly connected component made up of a strict subset of the nodes $S$, a conflict is reported with explanation
   \[
   \bigwedge_{i \in S, j \in V \setminus S} [x_i \neq j] \rightarrow false.
   \]
   For subcircuit, if there exists a node $a \in S$ where $in(a)$ holds, then for each node $b \in V \setminus S$ we set $x_b = b$ with explanation
   \[
   \left( [x_a \neq a] \land \bigwedge_{j \in V \setminus S} [x_i \neq j] \right) \rightarrow [x_b = b].
   \]

2. Only one edge leads from the first subtree to the root.
   Let $r$ be the root node, $a$ be the unique node reaching the root from the first subtree, and $A$ be the set of all nodes in the first subtree.
   Then for circuit the clause generated is
   \[
   \bigwedge_{i \in A, j \in V \setminus A, i \neq a \lor j \neq r} [x_i \neq j] \rightarrow [x_a = r].
   \]
   For subcircuit it is
   \[
   \left( [x_b \neq b] \land [x_c \neq c] \land \bigwedge_{i \in A, j \in V \setminus A, i \neq a \lor j \neq r} [x_i \neq j] \right) \rightarrow [x_a = r],
   \]
   where $b \in A, c \in V \setminus A, in(b)$ and $in(c)$.

3. Only one edge leads from subtree $C$ to the previous subtree $B$.
   In this case the reason the edge is required depends on the structure of the tree. Let $B$ be the set of nodes in $B$, and $C$ be the set of nodes in $C$. Also let $A$ be the set of nodes in subtrees before $B$, and $D$ be the set of nodes which were included in subtrees after $C$ or not reached in the search at all. Let $c$ be the unique node in subtree $C$ that reaches node $b$ in subtree $B$. 
For circuit the clause is
\[
\left( \bigwedge_{i \in A, j \in B \cup C \cup D} [x_i \neq j] \land \bigwedge_{i \in B, j \in C \cup D} [x_i \neq j] \land \bigwedge_{i \in C, j \in B \cup D, i \neq c \land i \neq j} [x_i \neq j] \right) \rightarrow [x_c = b].
\]

For subcircuit, where \( p \in B \) and \( q \in C \), \( \text{in}(p) \) and \( \text{in}(q) \), the clause is
\[
\left( [x_p \neq p] \land [x_q \neq q] \land \bigwedge_{i \in A, j \in B \cup C \cup D} [x_i \neq j] \land \bigwedge_{i \in B, j \in C \cup D} [x_i \neq j] \land \bigwedge_{i \in C, j \in B \cup D, i \neq c \land i \neq j} [x_i \neq j] \right) \rightarrow [x_c = b].
\]

4. No edges lead from sub-tree \( C \) to the previous sub-tree \( B \).
This case is very similar to the above case.
For circuit the clause is
\[
\left( \bigwedge_{i \in A, j \in B \cup C \cup D} [x_i \neq j] \land \bigwedge_{i \in B, j \in C \cup D} [x_i \neq j] \land \bigwedge_{i \in C, j \in B \cup D} [x_i \neq j] \right) \rightarrow \text{false}.
\]

For subcircuit, where \( p \in B \) and \( q \in C \), \( \text{in}(p) \) and \( \text{in}(q) \), the clause is
\[
\left( [x_p \neq p] \land [x_q \neq q] \land \bigwedge_{i \in A, j \in B \cup C \cup D} [x_i \neq j] \land \bigwedge_{i \in B, j \in C \cup D} [x_i \neq j] \land \bigwedge_{i \in C, j \in B \cup D} [x_i \neq j] \right) \rightarrow \text{false}.
\]

5. An edge skips one or more subtrees.
In this case reasoning again depends on the structure of the tree. Let \( c \) be the origin of the edge and \( a \) its destination. Take \( A \) as the set of nodes in the same or an earlier sub-tree to that of \( a \), \( B \) as the set of nodes in sub-trees between that of \( a \) and \( c \) (of which there is at least one), and \( C \) as the set of nodes in the same or later sub-tree as that of \( c \) plus nodes not reached by the search.
For circuit the clause generated is
\[
\left( \bigwedge_{i \in A, j \in B \cup C} [x_i \neq j] \land \bigwedge_{i \in B, j \in C} [x_i \neq j] \right) \rightarrow [x_c \neq a].
\]
For subcircuit it is
\[
\left( \left[ x_b \neq b \right] \land \bigwedge_{i \in A, j \in B \cup C} \left[ x_i \neq j \right] \land \bigwedge_{i \in B, j \in C} \left[ x_i \neq j \right] \right) \rightarrow \left[ x_c \neq a \right],
\]
where \( b \in B \) and \( \text{in}(b) \).

6. Edges leading from the root to a subtree other than the last.

Let \( E \) be the set of nodes in subtrees before the last, \( L \) be the set of nodes in the last subtree or not reached by the search, and \( r \) be the root node. An edge from \( r \) to \( e \) where \( e \in E \) is pruned with the following explanation.

For circuit,
\[
\bigwedge_{i \in E, j \in L} \left[ x_i \neq j \right] \rightarrow \left[ x_r \neq e \right].
\]

For subcircuit,
\[
\left( \left[ x_l \neq l \right] \land \bigwedge_{i \in E, j \in L} \left[ x_i \neq j \right] \right) \rightarrow \left[ x_r \neq e \right],
\]
where \( l \in L \) and \( \text{in}(l) \).

7. Non-viable edge discovered within a subtree.

This is the case where the first child of a node within the traversal tree does not reach any node above its parent. The edge from parent to child is pruned, because there is no way to reach outside the child’s part of the tree without going through the parent node. The explanations are therefore as follows, where \( c \) is the child node, \( C \) is the set of nodes in the subtree rooted at \( c \), \( p \) is the parent node, and \( A = V \setminus (C \cup \{ p \}) \) is the set of all nodes excluding that of the parent and nodes in the subtree rooted at the child.

For circuit,
\[
\bigwedge_{i \in C, j \in A} \left[ x_i \neq j \right] \rightarrow \left[ x_p \neq c \right],
\]
and for subcircuit,
\[
\left( \left[ x_a \neq a \right] \land \left[ x_b \neq b \right] \land \bigwedge_{i \in C, j \in A} \left[ x_i \neq j \right] \right) \rightarrow \left[ x_p \neq c \right],
\]
where \( a \) and \( b \) are nodes such that \( a \in A, b \in C, \text{in}(a) \) and \( \text{in}(b) \).
Root node selection

In [99], it was shown that the choice of root node can have a significant impact on the performance of the scc algorithm. In particular, choosing a random root was very successful. We are interested in whether or not choosing the root randomly is still beneficial when using explanations. The effectiveness of lazy clause generation depends on the opportunity to make use of the learned clauses, so randomness may be detrimental if it results in a more varied exploration which does not often reach similar nodes.

We consider five options for the root selection strategy.

1. Always choose the first node.
2. Choose the first node with unfixed successor variable.
3. Choose a random node.
4. Choose a random node with unfixed successor variable.
5. Run the algorithm on every node.

If the root’s successor is fixed, then there will be exactly one subtree below it and therefore no opportunities for propagation unless multiple strongly connected components are discovered, or the prune-within rule fires. For this reason it is probably better to choose an unfixed root. By including this option separately we can more accurately judge the benefit of making a random selection.

For subcircuit, the first, third and fifth strategies are modified slightly to avoid choosing a root which is fixed in a self-cycle, as this is guaranteed not to produce any propagation. The other strategies already avoid this by choosing a node with unfixed successor. In the case where all successors are fixed these strategies also choose a non-self-cycle node as the root.

Table 6.5 shows the results of our root selection experiments. In most cases, choosing as root the first node with unfixed successor was better than always choosing the very first node. Surprisingly that didn’t seem to be the case for subcircuit and subpath when using VSIDS search. This could be because a node with fixed successor is guaranteed to be in the circuit (except if it is a fixed self-cycle, but we never choose such a node as the root). Choosing a root which is fixed and therefore in the circuit may allow inconsistencies to be detected earlier, even though little other propagation will be possible. Failing early can be beneficial for VSIDS since it quickly learns how to escape the inconsistency. Note that we can’t make a meaningful comparison between these options for subpath with inorder search as in both cases almost all instances timed out.

Choosing a random root was clearly beneficial for all versions of the problem, and for both search strategies. This is in agreement with the results in [99], which is quite encouraging as it suggests that techniques involving randomness which are effective without explanation can still be beneficial when using explanation. Running the algorithm for every potential root did reduce the search
space significantly, but due to the very high overhead this strategy was much slower than random root selection for all problems.

For circuit and path, choosing randomly from among the nodes with unfixed successors appeared to be the best strategy. For subcircuit and subpath the results are not so clear. For subcircuit using VSIDS search and for subpath using inorder search, it was better to only eliminate nodes fixed in self-cycles, again probably due to the fact that in order for conflicts to be detected by the subcircuit propagator the exploration must reach at least one node which is required to be included in the circuit. However, for subcircuit using inorder search, excluding nodes with fixed successors still gave slightly better results, and for subpath using VSIDS search, although excluding these nodes resulted in higher numbers of failures and propagations, the execution time was slightly shorter.

In the remainder of our experiments we use random root selection. The circuit version of the propagator excludes nodes with fixed successor, while the subcircuit version only excludes self-cycle nodes (as this appeared to be the better choice overall for subcircuit and subpath).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Root</th>
<th>Inorder Search</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fails</td>
<td>Props</td>
<td>Time</td>
<td>Fails</td>
<td>Props</td>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>circuit</td>
<td>always first</td>
<td>196</td>
<td>1099</td>
<td>71.3</td>
<td>(48)</td>
<td>0.9</td>
<td>17.9</td>
<td>0.64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first unfixed</td>
<td>118</td>
<td>678</td>
<td>42.3</td>
<td>(26)</td>
<td>0.8</td>
<td>16.9</td>
<td>0.60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>89</td>
<td>524</td>
<td>33.9</td>
<td>(22)</td>
<td>0.8</td>
<td>16.9</td>
<td>0.60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rand unfixed</td>
<td>77</td>
<td>453</td>
<td>30.4</td>
<td>(21)</td>
<td>0.7</td>
<td>15.7</td>
<td>0.55</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>all roots</td>
<td>45</td>
<td>277</td>
<td>52.5</td>
<td>(33)</td>
<td>0.5</td>
<td>12.5</td>
<td>2.65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>path</td>
<td>always first</td>
<td>350</td>
<td>2242</td>
<td>257.4</td>
<td>(180)</td>
<td>1.5</td>
<td>26.8</td>
<td>1.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first unfixed</td>
<td>313</td>
<td>2027</td>
<td>244.6</td>
<td>(174)</td>
<td>1.3</td>
<td>25.0</td>
<td>0.90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>211</td>
<td>1320</td>
<td>162.2</td>
<td>(103)</td>
<td>1.2</td>
<td>25.0</td>
<td>0.85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rand unfixed</td>
<td>205</td>
<td>1279</td>
<td>152.4</td>
<td>(95)</td>
<td>1.1</td>
<td>23.4</td>
<td>0.79</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>all roots</td>
<td>125</td>
<td>869</td>
<td>211.1</td>
<td>(138)</td>
<td>0.7</td>
<td>18.3</td>
<td>3.98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subcircuit</td>
<td>always first</td>
<td>385</td>
<td>2636</td>
<td>171.6</td>
<td>(165)</td>
<td>3.2</td>
<td>57.7</td>
<td>1.50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first unfixed</td>
<td>311</td>
<td>2144</td>
<td>149.0</td>
<td>(72)</td>
<td>3.3</td>
<td>60.8</td>
<td>1.52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>81</td>
<td>509</td>
<td>37.6</td>
<td>(15)</td>
<td>2.1</td>
<td>36.4</td>
<td>1.09</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rand unfixed</td>
<td>78</td>
<td>487</td>
<td>35.6</td>
<td>(15)</td>
<td>2.2</td>
<td>37.6</td>
<td>1.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>all roots</td>
<td>29</td>
<td>236</td>
<td>79.8</td>
<td>(39)</td>
<td>1.0</td>
<td>25.1</td>
<td>8.88</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subpath</td>
<td>always first</td>
<td>623</td>
<td>4488</td>
<td>596.6</td>
<td>(494)</td>
<td>5.3</td>
<td>74.2</td>
<td>2.48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first unfixed</td>
<td>628</td>
<td>4411</td>
<td>598.2</td>
<td>(492)</td>
<td>5.3</td>
<td>77.7</td>
<td>2.36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>271</td>
<td>1706</td>
<td>220.5</td>
<td>(64)</td>
<td>3.4</td>
<td>50.8</td>
<td>1.76</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rand unfixed</td>
<td>257</td>
<td>1801</td>
<td>251.2</td>
<td>(95)</td>
<td>3.8</td>
<td>53.3</td>
<td>1.68</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>all roots</td>
<td>120</td>
<td>1034</td>
<td>286.3</td>
<td>(140)</td>
<td>1.6</td>
<td>34.8</td>
<td>10.90</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Comparison of root selection strategies for the scc algorithm. Each figure is the average for 500 instances with 65 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.
Additional pruning rules

We would also like to discover the impact of the two extra pruning rules we have suggested (prune root and prune within). In the previous experiments we used the original version of the scc algorithm which excludes these pruning rules. Table 6.6 shows the impact of adding one or both of the additional rules for each version of our problem.

It is clear that for all versions of our problem both prune within and prune root are beneficial. For circuit and path, prune root was more effective, reducing the average execution time by around 30-40%, while prune within gave a more modest improvement of between 7% and 30%. For subcircuit and subpath however, prune within performed better than prune root, most obviously in the case of subpath using inorder search where adding this rule reduced the execution time by almost 85%.

Combining the two rules was the best (or tied best) option in all cases, whether considering failures, propagations, or execution time. Therefore in the remainder of this paper we include both prune within and prune root whenever scc is used.

The decision of whether or not to use prune root will clearly affect the choice of root selection strategy. We conducted further experiments to verify that using both additional propagation rules with random root selection is indeed the best combination for our problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Pruning Rules</th>
<th>Inorder Search</th>
<th>VSIDS Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fails</td>
<td>Props</td>
</tr>
<tr>
<td>circuit</td>
<td>original scc</td>
<td>77</td>
<td>453</td>
</tr>
<tr>
<td></td>
<td>scc+within</td>
<td>128</td>
<td>696</td>
</tr>
<tr>
<td></td>
<td>scc+root</td>
<td>75</td>
<td>447</td>
</tr>
<tr>
<td></td>
<td>scc+within+root</td>
<td>77</td>
<td>412</td>
</tr>
<tr>
<td>path</td>
<td>original scc</td>
<td>205</td>
<td>1279</td>
</tr>
<tr>
<td></td>
<td>scc+within</td>
<td>251</td>
<td>1606</td>
</tr>
<tr>
<td></td>
<td>scc+root</td>
<td>192</td>
<td>1206</td>
</tr>
<tr>
<td></td>
<td>scc+within+root</td>
<td>148</td>
<td>968</td>
</tr>
<tr>
<td>subcircuit</td>
<td>original scc</td>
<td>81</td>
<td>509</td>
</tr>
<tr>
<td></td>
<td>scc+within</td>
<td>47</td>
<td>280</td>
</tr>
<tr>
<td></td>
<td>scc+root</td>
<td>46</td>
<td>299</td>
</tr>
<tr>
<td></td>
<td>scc+within+root</td>
<td>26</td>
<td>186</td>
</tr>
<tr>
<td>subpath</td>
<td>original scc</td>
<td>271</td>
<td>1706</td>
</tr>
<tr>
<td></td>
<td>scc+within</td>
<td>71</td>
<td>484</td>
</tr>
<tr>
<td></td>
<td>scc+root</td>
<td>274</td>
<td>1745</td>
</tr>
<tr>
<td></td>
<td>scc+within+root</td>
<td>62</td>
<td>425</td>
</tr>
</tbody>
</table>

Table 6.6: Additional propagation rules for the scc algorithm. Each figure is the average for 500 instances with 65 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.
Choosing evidence literals

As with prevent explanations, the subcircuit versions of clauses used to explain scc propagation require the inclusion of evidence literals (literals \( x_k \neq k \)) for a node \( k \) whose successor variable’s current domain does not include \( k \)). In many cases there are several appropriate literals which could be chosen. We experimented with the same options for selecting evidence literals as discussed for propagation of prevent.

As can be seen in Table 6.7, in most cases choosing the literal which became fixed highest in the search tree made very little difference compared with simply choosing the first applicable literal. However, when using VSIDS search for subpath it did appear to be beneficial. Since the highest heuristic was never far from the best option, and was in most cases significantly better than the opposite strategy of choosing the literal fixed lowest in the search tree, we decided to use the highest heuristic for scc evidence literal selection in further experiments (making this the same as prevent evidence literal selection).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Heuristic</th>
<th>Inorder Search</th>
<th>VSIDS Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fails</td>
<td>Props</td>
</tr>
<tr>
<td>subcircuit</td>
<td>first</td>
<td>26</td>
<td>186</td>
</tr>
<tr>
<td></td>
<td>last</td>
<td>26</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td>36</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td>highest</td>
<td>23</td>
<td>160</td>
</tr>
<tr>
<td>subpath</td>
<td>first</td>
<td>62</td>
<td>425</td>
</tr>
<tr>
<td></td>
<td>last</td>
<td>81</td>
<td>554</td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td>113</td>
<td>758</td>
</tr>
<tr>
<td></td>
<td>highest</td>
<td>61</td>
<td>418</td>
</tr>
</tbody>
</table>

Table 6.7: Comparison of heuristics for evidence literal selection for the scc algorithm. Each figure is the average for 500 instances with 65 locations. Failure and propagation counts are given in thousands, while times are in seconds. Where at least one instance reached the time limit of 10 minutes, the number of timeouts is shown in brackets.

6.5 The Effect of Explanation

In this section we explore the effect of explaining circuit. We first investigate which propagation algorithm performs the best with and without explanation, and then go on to compare explaining and non-explaining propagators.

6.5.1 Propagation complexity trade-off

In all propagators there is a trade off between the complexity of the algorithm and its power. In lazy clause generation propagators we also need to consider the size and generality of the explanations produced. When using explanation,
a weakly propagating algorithm which produces short highly reusable explanations, may be able to compete with a stronger propagator whose explanations are much more complex and large and not very reusable.

We therefore wish to investigate whether adding explanation changes the relative effectiveness of the different circuit propagation algorithms. We consider four different versions of the propagator, as follows.

1. The check algorithm only.

2. Both check and prevent (recall prevent cannot be used alone).

3. The scc algorithm alone.

4. All three algorithms in combination.

When multiple algorithms are used, we apply the least expensive first, and then continue with each more expensive algorithm if no conflict has been discovered. For scc we used random root selection and both extra propagation rules. Experimentation showed that just as this is the best combination when using explanation, it is also the best combination when not using explanation.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Algorithm</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit</td>
<td>check</td>
<td>14.9</td>
<td>355.6</td>
<td>567.7</td>
</tr>
<tr>
<td></td>
<td>check + prevent</td>
<td>7.2</td>
<td>244.0</td>
<td>536.0</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>11.4</td>
<td>289.4</td>
<td>556.2</td>
</tr>
<tr>
<td></td>
<td>check + prevent + scc</td>
<td>9.2</td>
<td>277.2</td>
<td>542.3</td>
</tr>
<tr>
<td></td>
<td>Gecode</td>
<td>0.1</td>
<td>4.4</td>
<td>33.5</td>
</tr>
<tr>
<td>path</td>
<td>check</td>
<td>25.7 (5)</td>
<td>259.4 (143)</td>
<td>548.0 (426)</td>
</tr>
<tr>
<td></td>
<td>check + prevent</td>
<td>15.1 (1)</td>
<td>236.5 (127)</td>
<td>521.0 (398)</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>22.4 (4)</td>
<td>258.9 (144)</td>
<td>546.2 (421)</td>
</tr>
<tr>
<td></td>
<td>check + prevent + scc</td>
<td>20.2 (3)</td>
<td>265.5 (148)</td>
<td>536.7 (414)</td>
</tr>
<tr>
<td></td>
<td>Gecode</td>
<td>0.3</td>
<td>25.8 (11)</td>
<td>136.0 (76)</td>
</tr>
<tr>
<td>subcircuit</td>
<td>check</td>
<td>15.1 (1)</td>
<td>409.9 (269)</td>
<td>578.9 (470)</td>
</tr>
<tr>
<td></td>
<td>check + prevent</td>
<td>10.0</td>
<td>318.0 (189)</td>
<td>555.7 (439)</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>17.0 (1)</td>
<td>396.9 (254)</td>
<td>575.8 (463)</td>
</tr>
<tr>
<td></td>
<td>check + prevent + scc</td>
<td>12.5</td>
<td>359.2 (224)</td>
<td>564.8 (452)</td>
</tr>
<tr>
<td>subpath</td>
<td>check</td>
<td>24.1 (2)</td>
<td>375.7 (237)</td>
<td>576.0 (462)</td>
</tr>
<tr>
<td></td>
<td>check + prevent</td>
<td>17.6 (3)</td>
<td>327.2 (194)</td>
<td>556.3 (440)</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>29.9 (5)</td>
<td>394.9 (253)</td>
<td>576.3 (464)</td>
</tr>
<tr>
<td></td>
<td>check + prevent + scc</td>
<td>28.2 (4)</td>
<td>363.8 (224)</td>
<td>570.9 (456)</td>
</tr>
</tbody>
</table>

Table 6.8: Comparison of propagation algorithms without explanation, using inorder search. Each figure shown is the average execution time (secs) over 500 instances of the given size. Where at least one instance reached the 10 minute time limit, the number of timeouts is given in brackets.
Comparing algorithms without explanation

Table 6.8 shows the results without explanation. For all problem versions the prevent algorithm is an improvement over check alone. The scc algorithm used alone performs better than check for circuit and path, but worse than check for subcircuit and subpath, and always worse than check plus prevent. It appears that the scc algorithm is too expensive to pay off, as using all algorithms together is better than scc alone, but still slower than check and prevent without scc.

For circuit and path we include execution times for Gecode (version 4.0.0) using the default circuit implementation which includes all three algorithms (although obviously without our modifications to scc). It is clear that without explanation the Gecode implementation is much faster than Chuffed.

Comparing algorithms with explanation

We now consider the results using explanation shown in Table 6.9. The addition of prevent to check was still clearly beneficial. The striking difference is that with explanation the scc algorithm actually performs very well. This is interesting as it suggests that the explanations produced by scc, although large, are sufficiently general to be effective. In most cases when using inorder search, scc alone was the best performing algorithm, but for the hardest problems (large sizes of subpath) it was better to use all three algorithms together. For VSIDS search, using all three algorithms together seemed to be the best option.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Algorithm</th>
<th>Inorder Search</th>
<th>VSIDS Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>55 60 65</td>
<td>55 60 65</td>
</tr>
<tr>
<td>circuit</td>
<td>check</td>
<td>99 (66) 132 (96) 166 (122)</td>
<td>0.42 0.63 2.37 (1)</td>
</tr>
<tr>
<td></td>
<td>check+prev</td>
<td>73 (51) 114 (86) 128 (96)</td>
<td>0.26 0.31 0.35</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>5 (2) 10 (4) 17 (9)</td>
<td>0.24 0.27 0.35</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>11 (8) 19 (13) 22 (14)</td>
<td>0.25 0.30 0.31</td>
</tr>
<tr>
<td>path</td>
<td>check</td>
<td>265 (184) 305 (227) 353 (271)</td>
<td>0.69 1.00 2.69 (1)</td>
</tr>
<tr>
<td></td>
<td>check+prev</td>
<td>214 (146) 274 (201) 314 (233)</td>
<td>0.37 0.48 0.56</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>48 (22) 48 (20) 76 (40)</td>
<td>0.47 0.41 0.52</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>44 (22) 73 (39) 97 (56)</td>
<td>0.33 0.40 0.46</td>
</tr>
<tr>
<td>subcircuit</td>
<td>check</td>
<td>216 (127) 195 (100) 270 (154)</td>
<td>1.04 1.74 2.19</td>
</tr>
<tr>
<td></td>
<td>check+prev</td>
<td>169 (86) 155 (72) 242 (133)</td>
<td>0.54 0.74 0.94</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>3 (1) 2 7 (3)</td>
<td>0.45 0.58 0.73</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>7 (3) 2 9 (2)</td>
<td>0.41 0.50 0.61</td>
</tr>
<tr>
<td>subpath</td>
<td>check</td>
<td>400 (262) 410 (250) 511 (371)</td>
<td>1.17 1.71 2.64</td>
</tr>
<tr>
<td></td>
<td>check+prev</td>
<td>382 (245) 393 (245) 494 (343)</td>
<td>0.63 0.85 1.17</td>
</tr>
<tr>
<td></td>
<td>scc</td>
<td>19 (3) 19 (4) 29 (5)</td>
<td>0.64 0.82 1.05</td>
</tr>
<tr>
<td></td>
<td>all</td>
<td>19 (3) 17 (1) 26 (6)</td>
<td>0.58 0.78 1.04</td>
</tr>
</tbody>
</table>

Table 6.9: Comparison of propagation algorithms when using explanation. Each figure is the average execution time (secs) over 500 instances of the given size. The number in brackets gives instances reaching the time limit of 10 minutes.
6.5.2 Benefit of explanation

Having investigated the best choice of algorithm when using and not using explanation, we now consider the impact of explanation on the performance of our circuit propagator. In order to make a direct comparison, we need to use the same circuit algorithm with and without explanation. When not using explanation, the best choice of algorithm was `check` plus `prevent` without `scc`, but when using explanation the `scc` algorithm was vital for good performance. We have chosen to use all three algorithms together, since this gives reasonable performance both with and without explanation, and is the best option when using VSIDS search. We also include results for Gecode, which was found earlier to be much faster than Chuffed without explanation regardless of the choice of algorithm.

Table 6.10 shows the average execution time and number of failures for increasing problem sizes, giving a comparison between Chuffed without explanation (inorder search), Chuffed with explanation using inorder search, Chuffed with explanation and VSIDS search, and Gecode (inorder search). Comparing Chuffed with explanation to Chuffed without explanation, the smallest improvement was for `circuit` problems. For the smallest size (15) these were solved almost 80 times faster when using explanation. For size 30 these problems were solved around 1000 times faster. Looking at size 15, the improvement was greatest for `subpath` problems, which are also the hardest. For `subpath` problems of this size, using explanation was already more than 700 times faster than not using explanation. For the larger sizes too many instances timed out when not.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solver</th>
<th>Execution Time (secs)</th>
<th>Fails (000s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>circuit</td>
<td>No expl</td>
<td>9.16</td>
<td>594.4 (492)</td>
</tr>
<tr>
<td></td>
<td>Expl</td>
<td>0.12</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>VSIDS</td>
<td>0.04</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>Gecode</td>
<td>0.09</td>
<td>111.1 (75)</td>
</tr>
<tr>
<td>path</td>
<td>No expl</td>
<td>20.19 (3)</td>
<td>597.3 (495)</td>
</tr>
<tr>
<td></td>
<td>Expl</td>
<td>0.05</td>
<td>2.6 (1)</td>
</tr>
<tr>
<td></td>
<td>VSIDS</td>
<td>0.04</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>Gecode</td>
<td>0.33</td>
<td>344.3 (249)</td>
</tr>
<tr>
<td>subcircuit</td>
<td>No expl</td>
<td>12.47</td>
<td>598.6 (498)</td>
</tr>
<tr>
<td></td>
<td>Expl</td>
<td>0.04</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>VSIDS</td>
<td>0.04</td>
<td>0.1</td>
</tr>
<tr>
<td>subpath</td>
<td>No expl</td>
<td>28.19 (4)</td>
<td>598.2 (497)</td>
</tr>
<tr>
<td></td>
<td>Expl</td>
<td>0.04</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>VSIDS</td>
<td>0.04</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 6.10: Experiment showing the effect of explanation, comparing Chuffed without explanation, Chuffed with explanation, Chuffed with explanation using VSIDS search, and Gecode. Each figure is the average for 500 instances of the given size. The number of timeouts (>10 mins) is shown in brackets.
using explanation to accurately estimate the improvement factor. Using VSIDS rather than inorder search was a further improvement over just adding explanation, and this improvement also increased with increasing problem size. For example, solving circuit problems using VSIDS search was 3, 6 and 60 times faster than inorder search (with explanation) for sizes 15, 30 and 60 respectively. Although Gecode was much faster than Chuffed without explanation, it was able to compete with Chuffed with explanation only for the smallest size of circuit problems. By size 60, Gecode timed out at 10 minutes for 75% of circuit problems and almost all path problems, while Chuffed with explanation took on average 19 and 72 seconds, and Chuffed with explanation and VSIDS averaged less than half a second for both problems.

The failure counts also given in Table 6.10 show a similar pattern to the times.

These results make it very clear that explanation is highly effective for circuit (and subcircuit) propagation. It is also apparent that larger and harder problems benefit more. That is, those instances with greater average search time without explanation also have a larger improvement factor. This is probably because when more search is required, there are more opportunities to make use of learned clauses.

6.6 Related Work

The purpose of this chapter is to investigate how explanation can be used, and its effect on the circuit constraint and its variants. We are unaware of any other work on circuit with explanation. However, for context we briefly describe some related constraints and propagation algorithms.

In this work we considered three propagation algorithms for circuit. Another (incomplete) circuit propagation algorithm was suggested in [70]. This algorithm uses graph separators to detect nonhamiltonian edges which can then be removed. While of theoretical interest, the algorithm is very complex and appears very slow to propagate, which is why we have not included it in our study.

The circuit constraint can clearly be implemented using the more general cycle constraint[10], by fixing the number of cycles to one. This is also possible for subcircuit, although an extra constraint is required to ensure that the total number of cycles is one more than the number of self-cycles, and this means it will not propagate as strongly as the propagator we describe. Since path and subpath are implemented using circuit and subcircuit respectively, these can be implemented using cycle as well.

We have already mentioned the more general version of path which enforces $n$ disjoint paths where $n$ is a variable. A further generalisation of this is the tree constraint, introduced in [11] and extended in [12]. The extended version of tree includes precedence, incomparability and degree constraints, and can be used to implement subpath as well as path constraints.
Another graph-based constraint of particular relevance is the DomReachability constraint, which uses reasoning based on node dominance and reachability and can be used to solve the ordered simple path with mandatory nodes problem [93]. This problem is equivalent to that solved by subpath with the additional requirement of enforcing an order between certain pairs of nodes (which is not possible using our circuit-based implementation). This additional functionality could be useful for the application of subpath discussed in the previous chapter (constraining the graph of execution paths when using loop untangling). We have not investigated the DomReachability constraint further because the work discussed in this chapter was actually completed before the idea of loop untangling was conceived. A study comparing the effectiveness of circuit-based versus dominance/reachability-based propagators for loop untangling would be interesting future work.

While this work focuses on propagation algorithms designed specifically for circuit, using a simple transformation to allow these to be applied to path problems, it is also possible to make the opposite transformation and implement circuit using a path constraint (or a tree constraint with the ability to restrict the degree of nodes). This is achieved by selecting an arbitrary node to be the start and end of the path, and splitting this node into two - the start node keeps all outgoing edges and the end node keeps the incoming edges.

The same technique can be used for subcircuit as long as there is at least one required node. A general propagator for subcircuit using the path formulation is possible, but it would need to wait until at least one node became mandatory before it could begin propagating, using this node as the start and end of the path.

Actually, the scc algorithm is closely related to a propagation algorithm for path called Reduced Path [42]. This algorithm finds strongly connected components of the graph and enforces that they form a chain with exactly one edge between neighbouring components and no edges skipping components. If the root node from the scc algorithm were the one to be split during the conversion to a path formulation, then each subtree would become one or more strongly connected components in this chain. The propagations performed by scc are therefore a subset of those performed by Reduced Path.

The prune within improvement to scc is also covered by existing path propagation algorithms. This rule is actually a restricted form of dominator based pruning, similar to that discussed in [41] but only detected when it is convenient to do so as part of the scc algorithm.

Although path propagators can potentially remove more edges than the scc algorithm, there is a disadvantage to using a path formulation for circuit, which is that the node to split is chosen up front. This is equivalent to always choosing the same root node for the scc algorithm, and our experimental results show that it is beneficial to choose the root randomly. An interesting avenue for future work would be a propagator for circuit based on path propagation algorithms, but which does not commit to a node to split up front, instead selecting this node each time propagation is performed, perhaps randomly as we have done for the scc algorithm.
6.7 Conclusion

We have investigated how best to add explanation to the global constraint circuit and its variants. The results show that explanation is highly beneficial for problems involving these constraints. The resulting propagators compare very favourably against the state-of-the-art circuit implementation in Gecode, one of the fastest available constraint programming solvers.

Somewhat surprisingly the complex scc propagator which creates very large explanations, is not dominated by the cheaper propagators, whose explanations are typically very much smaller. Indeed it is certainly not always the case that once learning is used that simpler propagators are preferable. But just as without learning, sometimes weaker propagators are preferable to strong propagators. A full CP system must support both kinds.

Perhaps more surprising is the fact that adding randomness to an explaining propagator (the choice of root for scc) is beneficial. Usually explaining propagators want to be deterministic so that they tend to reuse earlier explanations again and again. It appears for circuit that the benefit of random root selection is substantial enough to overcome this disadvantage.
Chapter 7

The Reaching Definitions Global Constraint

7.1 Introduction

This chapter introduces a new global constraint, called reaching definitions, to be used when modelling procedural code. The purpose of this constraint is to ensure that the values of state queries (most commonly variable references) correspond correctly to the state changes (assignments) and the execution path.

The reaching definitions constraint replaces the decomposition based on element constraints and implications used in Chapter 4. It is motivated by a need to drop the assumption of a known order for changes and queries implicit in that approach, in order to facilitate loop untangling as discussed in Chapter 5. In this chapter we describe our first attempt at implementing the reaching definitions global constraint in the lazy clause generation solver Chuffed. We show that this initial implementation successfully maintains the propagation strength of the decomposition, and also discuss two extensions providing stronger propagation in some cases.

The next section revisits the translation process used to convert procedural code into equivalent constraints, explaining the changes to this process required to use the new global constraint. Later sections discuss the necessary solver functionality, including our implementation of the new global constraint.

7.2 Model Generation

Previously the translation from code to constraints was separated into two parts. First, we flattened the code into an ordered list of state changes (assignments) and queries (variable references), with each change having an associated condition defining when that change will be executed. Then, we added constraints and variables to ensure that each state query corresponded correctly to the
preceding state changes and their execution conditions. Figure 7.1 shows the
flattened list and the resulting constraints for a small code snippet.

In the new approach, we first construct a graph representing the possible
execution paths, then collect the other required information about changes and
queries in order to make one or more calls to the global constraint, which will
replace the previously used query constraints.

**Execution graph**

Our new translation uses a directed graph instead of a simple list to represent
the possible execution paths. This is necessary when using loop untangling (as
discussed in Chapter 5), but is equally applicable when not untangling. The
only difference in the non-untangling case is that the graph will be acyclic.
The global constraint uses the execution graph to reason about how execution
can flow between changes and queries. Rather than associating an execution
condition with each change, we now have a condition associated with each edge
in the execution graph, as illustrated in Figure 7.1. For more details about how
this graph is constructed, refer to the loop untangling chapter (Section 5.3.4).

We model the execution graph using a successor variable for each node (or
more efficiently for each contiguous block with no branches), and constrain it
to form a path using the *subpath* constraint introduced in Chapter 6.1 The
successor variables are further constrained to respect the edge conditions. Each
solution of these constraints represents a valid execution path, where a subset
of the changes and queries (those occurring in blocks included on the chosen
path) will be executed, while the remainder are not executed.

**Match arguments**

In addition to the execution graph, the reaching definitions propagator also
needs matchingness information in order to enforce the required constraint; that
each query result equals the value associated with the most recently executed
matching change. As discussed in previous chapters, what is meant by *matching*
depends on the type of change and query. For example, a field assignment
matches a variable reference if it uses the same field name and refers to the
same object. For all of our change and query types, there are at most two
arguments which must be equal in order for a pair to match. We therefore
take the approach of simply providing the global constraint with two variables
for each change and query representing their match arguments. If only one
argument is required, the unneeded variable is fixed to 0. For example, for a
local variable assignment/reference, both match arguments will be fixed, one
representing nothing (with value 0), and the other giving the variable ID. For a
field assignment/reference, one gives the field ID while the other represents the
owning object. For a *Map put/get*, one match argument represents the map and
the other the key.

---

1We are using the *check+prevent* version of *subpath* as *scc* does not appear to be worthwhile
for our relatively simple graphs.
Values

The final required arguments for the global constraint are the variables representing change and query values. For an assignment, this is the variable giving the assigned value, and for a variable reference it is the result variable (whose value should be determined by the global constraint).

MiniZinc predicate

The reaching definitions constraint has the MiniZinc \[86\] type signature below.

```plaintext
predicate reaching_defs(int: numchanges, int: numqueries, int: numnodes,
    array[int] of var int: match1,
    array[int] of var int: match2,
    array[int] of var int: value,
    array[int] of int: fromindex,
    array[int] of int: toindex,
    array[int] of var bool: edge,
    array[int] of var bool: happens);
```

The arguments for this predicate are interpreted as follows. For an example illustrating its use see Figure 7.1.

- The numchanges and numqueries parameters give the number of relevant changes and queries. The remainder of the arguments assume changes take indices 1..numchanges and queries use numchanges+1..numchanges+numqueries.
- The match1 and match2 arrays hold the first and second match argument for each change and query.
- The value array holds the value variable for each change and query.
- The fromindex, toindex and edge arrays define the execution graph. Each triple defines an edge, with fromindex giving the index of the origin, toindex giving the index of the destination, and edge giving the edge condition (true if the edge is required, false if it has been eliminated).
  - The edge conditions are assumed to be appropriately linked to any other representation of the execution path (e.g. successor variables).
  - Index 0 is reserved for a special start-program node, which must be the source of the graph, while numnodes-1 represents the end-program node. If numnodes > numchanges+numqueries+2, the extra indices refer to nodes in the execution graph other than relevant changes/queries.
- The happens array gives a Boolean variable for each node which is true if that node occurs on the execution path and false otherwise. For this purpose we use the subpath representation (a node is excluded from the path iff the associated successor variable takes its own index as a value). That is, for the ith node \(n_i\), \(\text{happens}[i] = (\text{succ}[n_i] \neq n_i)\).
```plaintext
Code snippet:
x.f = 5;
if (z.f > 0)
y.f = y.f + 1;
return z.f;

Flattened list:

<table>
<thead>
<tr>
<th>name</th>
<th>cond</th>
<th>change/query</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>A.f</td>
<td>:= 0</td>
</tr>
<tr>
<td>c2</td>
<td>B.f</td>
<td>:= 0</td>
</tr>
<tr>
<td>c3</td>
<td>x.f</td>
<td>:= 5</td>
</tr>
<tr>
<td>q1</td>
<td>z.f</td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td>y.f</td>
<td></td>
</tr>
<tr>
<td>c4</td>
<td>(q1 &gt; 0)</td>
<td>y.f := q2 + 1</td>
</tr>
<tr>
<td>q3</td>
<td>z.f</td>
<td></td>
</tr>
</tbody>
</table>

Constraints (decomposition):
q1 = [0, 0, 5][index1]
[A, B, x][index1] = z
(x = z) → index1 ≥ 3

q2 = [0, 0, 5][index2]
[A, B, x][index2] = y
(x = y) → index2 ≥ 3

q3 = [0, 0, 5, q2 + 1][index3]
[A, B, x, y][index3] = z
(true, true, true, q1 > 0)[index3]
(x = z) → index3 ≥ 3
(y = z ∧ q1 > 0) → index3 ≥ 4

Execution graph:

Constraints (global):
succ[1] ∈ {2, 3}, succ[2] ∈ {2, 3}
subpath(succ, 1, 3)
(succ[1] = 2) → (q1 > 0)
(succ[1] = 3) → ¬(q1 > 0)
reaching_defs(4, 3, 11,
[A, B, x, y, z, y, z],
[f, f, f, f, f, f, f, f],
[0, 0, 5, q2 + 1, q1, q2, q3],
[0, 1, 2, 3, 5, 8, 6, 4, 9, 7],
[1, 2, 3, 5, 8, 6, 4, 9, 7, 10],
[T, T, T, T, T, succ[1] = 2],
[T, T, T, T, T, succ[2] ≠ 2, T, T],

Figure 7.1: Example showing for a simple code snippet the model produced using the decomposition approach (flattened list of changes/queries and then constraints for each query), and the model produced using the new global constraint (execution graph plus further node details passed to reaching_defs constraint). We assume variables x, y and z may all refer to either of two concrete objects A and B, both of which initially have value 0 for field f.
```
Reducing graph complexity

Although it would be possible for a single reaching definitions constraint to handle all changes and queries in the program under consideration, the propagator would need to work with essentially the entire execution graph. Since dealing with large graphs is very expensive, it is better to partition the changes and queries into disjoint subsets such that elements of different subsets do not affect each other. We can then apply a separate reaching definitions constraint to each subset using a compressed version of the execution graph specific to that subset.

In order to compute an appropriate partition of the changes and queries, we make use of an overestimation of the set of changes which may on some execution path reach each query. For each query $q$ we call this set of changes the relevant changes for $q$. The computation of relevant changes was discussed earlier in Chapter 4. For correctness we require that all relevant changes for a query $q$ will be managed by the same reaching definitions constraint as $q$. Since a change can be relevant to more than one query, each constraint may handle multiple queries.

Given the relevant changes for each query, we use a union-find data structure [107] to efficiently compute the required partition. The algorithm is very simple. We begin with all changes and queries in separate groups, and then for each change $c$ relevant to query $q$, we merge the current groups of $c$ and $q$.

The next section explains how we construct a compressed execution graph for use by each reaching definitions constraint, based on its subset of the changes and queries. The remainder of the chapter will discuss solver functionality including the implementation of the global constraint.

### 7.3 Graph Compression

In this section we discuss how the execution graph can be compressed to reflect only the structure relevant to a given subset of its nodes (the selected changes and queries for a single constraint as discussed above).

For the propagator to work correctly, the compressed graph must accurately reflect the possible paths through the nodes of interest in the original graph. We call this selected set of nodes of interest, plus the start and end program nodes, the important nodes. The compressed graph must retain all important nodes. Furthermore:

- Any path $P_o$ in the condensed graph must be supported by at least one path $P_c$ through the original graph, where $P_o$ visits the same subset of the important nodes as visited by $P_c$, in the same order.

- Any path $P_o$ in the original graph must be represented by a path $P_c$ through the condensed graph, where $P_c$ visits the same subset of the important nodes as visited by $P_o$, in the same order.

These conditions must hold initially, and must also be maintained during
propagation when edges are deleted from the original graph through search and deduction.

To this end, the compressed graph will contain a subset of the nodes in the original graph, and edges between pairs of nodes for which there exists a path in the original graph which does not go through any other retained node. That is, if our original graph is $G_o = (V_o, E_o)$ and our compressed graph $G_c = (V_c, E_c)$, then $V_c \subseteq V_o$, and $E_c = \{(a, b) : \exists$ an $a - b$ path $P$ in $G_o$, such that $\forall v \in P, v \notin (V_c \setminus \{a, b\})\}$.

Choosing nodes to retain

It is possible to construct a ‘compressed’ graph containing only the important nodes. However, this new graph may easily contain more edges than the original graph (consider the example in Figure 7.2). Also, each of these new edges represents a set of possibly disjoint paths in the original graph, which makes it expensive to maintain consistency between the two graphs during propagation. We therefore prefer to retain some additional structural nodes.

Definition Node $a$ directly reaches node $b$ if there exists in the original graph a path from $a$ to $b$ which does not pass through any retained nodes.

Beyond the important nodes, we wish to retain only nodes which directly reach more than one other retained node. The intention is to delete irrelevant structure from the graph, without destroying the relevant structure. The resulting compressed graph will never contain more nodes or edges than the original graph, and can be straightforwardly maintained during propagation.

Clearly our requirement for retained nodes above is cyclic. However, for acyclic execution graphs (the execution graph is only cyclic if we are using loop untangling) this is not a problem as we can simply process the nodes using a postorder depth first traversal. Assuming the graph is acyclic, by processing nodes in this order we ensure that when we consider a particular node $n$, we will have already processed all nodes reachable by $n$ and can therefore compute whether multiple retained nodes are directly reachable.

For cyclic graphs, we relax the requirement for retained nodes so that we can still compute them efficiently. We still perform a postorder traversal, and simply assume the worst case, that any edge leading from $n$ to a node which will be processed after $n$ leads directly to multiple retained nodes, and therefore $n$ should be retained. This results in the inclusion of some extra nodes and edges representing irrelevant structure. We discuss a simplification techniques used to mitigate this problem in Section 7.10.1 where we revisit loop untangling.

Computing edges

It is also necessary to compute the edges of the compressed graph, and to link these to edges of the original graph. To facilitate this we record for each non-retained node $n$ a retained representative. This will be the single retained node
directly reachable from \( n \). Note that such a node must exist for every non-retained node.

- If more than one retained node is directly reachable from \( n \) then \( n \) would be retained also.
- The end-of-program node is always retained. Any node from which there is no path to the end of the program is irrelevant to our problem (and is deleted in a preprocessing step). Therefore all non-retained nodes must reach at least one retained node.

Figure 7.2: (a) An execution graph with 9 important nodes (bold) and 17 edges. This graph shape is common as it is produced by an if-then statement inside a loop. (b) A 'compressed' version of this graph using only the important nodes, with 18 edges (more than the original graph). Our compression algorithm instead retains nodes \( e \), \( g \) and \( i \), producing graph (c).

141
Once we have chosen which nodes to retain and recorded the representative for other nodes, the edges for the compressed graph can be computed trivially. We simply take all edges from the original graph originating from a retained node, and if the destination node has not been retained, we replace it with its representative. That is, \( E_c = \{(a, \text{representative}(b)) : a \in V_c \land (a, b) \in E_o\} \).

In this way, we ensure that all relevant paths from the original graph are represented in the compressed graph. It is not possible to lose a relevant path, because we only compress paths which do not pass through any retained nodes. Furthermore, it is straightforward to maintain consistency between the two graphs. Each compressed edge corresponds to a subset of the original edges with that origin. When all original edges in the subset have been eliminated, the compressed edge is also eliminated.

Pseudo-code for the algorithm used to select retained nodes and compute a retained representative for each node is given in Figure 7.3. This algorithm is based on Tarjan's strongly connected components algorithm [106], and uses post-order computation for the representatives.

### 7.4 Optional Variables

As discussed earlier, the goal of our global constraint is to ensure that query results reflect the state changes occurring before that query on the chosen execution path. At a minimum, propagation should exclude from the query result any value which cannot correspond to a preceding matching change. While this is necessary and desirable, when combined with our graph representation of the possible execution paths it presents a problem for standard constraint solvers. The problem is that any query which does not fall on the chosen execution path will become an isolated node in the execution graph. The propagation algorithm will therefore find no preceding changes, and we will conclude that no value is acceptable for the query result. For standard decision variables, an empty domain triggers failure. Therefore if we try to implement even the most basic desired propagation, we will incorrectly force all queries to be included on the execution path.

Previously, when assuming a fixed order for changes and queries, we avoided this problem by assuming that all queries happen. Execution conditions were associated with changes only. Where necessary, a dummy change inserted immediately before the query provided a default value (in case no real matching change is executed). The execution condition associated with this dummy change ensured that it could only be chosen as the reaching change if the query did not fall on the execution path (and therefore its value was irrelevant).

Although it would be possible to approximate this previous solution by adjusting our propagation algorithms to retain a default value until the query is known to happen, we have chosen instead to pursue a more elegant solution based on optional variables.
computeReps() {
    numseen = 0;
    visit(startnode);
}

visit(v) {
    // assign visit number, visit unvisited children
    v.visitOrder = numseen;
    numseen = numseen + 1;
    v.lowpoint = v.visitOrder;
    SCCstack.push(v);
    v.onStack = true;
    if (v.important) {
        v.rep = v; // v should be retained
    }
}

// compute lowpoint and representative
for each edge (v,w) in G_o {
    if (w.visitOrder is undefined) {
        visit(w);
        v.lowpoint = min(v.lowpoint, w.lowpoint);
    } else if (w.onStack) { // haven't finished processing w
        v.lowpoint = min(v.lowpoint, w.visitOrder);
        v.rep = v; // assume worst and retain v
    }
    if (v.rep is undefined) {
        v.rep = w.rep;
    } else if (v.rep != w.rep) {
        v.rep = v; // retain v as it reaches multiple retained nodes
    }
}

// if v is the root of an SCC take the nodes in that SCC off the stack
if (v.lowpoint == v.visitOrder) {
    do {
        last = SCCstack.pop();
        last.onStack = false
    } while (last != v)
}

Figure 7.3: The algorithm used to select retained nodes and compute a retained representative for each non-retained node, when compressing the execution graph based on a chosen subset of important nodes. A retained node has itself as its representative.
7.4.1 Option types - background

Option types are one of the more recent additions to MiniZinc [80]. A new type modifier allows integer or Boolean variables to be declared *optional*. The basic idea is that unlike ordinary variables, an optional variable may take no value, in which case it is said to be *absent*. The occurrence or absence of an optional variable can be tied to various conditions through the use of an associated Boolean *occurs* variable.

**Modelling with option types**

Option types are very useful for modelling in cases where certain parts of the model are only relevant under conditions which depend on other decision variables. A common example of this is scheduling optional tasks. In a scheduling problem where some tasks may not be performed at all, the start time for a task is only relevant if that task is scheduled. If the task is not scheduled, its start time should have no effect on the satisfiability of the model. Using an optional variable to represent the start time is an elegant way of achieving that objective. Another example is configuration problems, where choices about features of a component are only relevant if that component is to be included.

Having modelled a problem using option types in MiniZinc, one has two choices for solving the model. Optional variables and the constraints on them can be automatically translated into a representation using only ordinary variables. Alternatively a solver which has an implementation for optional variables can receive the original variables and constraints directly.

For the automatic conversion to ordinary variables, MiniZinc replaces each optional variable with an ordinary variable paired with a Boolean *occurs* variable, and uses a library of constraint redefinitions to transform the constraints. This library of redefinitions follows the basic principle that absent variables should not constrain the rest of the model. While meeting this general requirement there are still multiple possibilities for the way constraints over optional variables can be interpreted. The MiniZinc implementors have chosen to consistently use the following strategies. See [80] for more details.

- For predicates, the *projection interpretation* is used. A constraint with an absent argument is satisfied if there exists a value for that argument which would satisfy the constraint. The value does not have to belong to the argument’s original domain.

- For functions, identity lifting is used when this makes sense, otherwise absorption lifting is used.
  - Identity lifting means that an absent value is treated as the identity for that operation (e.g. 0 for addition or *false* for disjunction). If there is no identity in the position of the absent variable then the result is absent.
  - Absorption lifting means that a function with an absent argument has an absent result.
This automatic translation allows modellers to use option types even if the underlying solver has no support for optional variables. However, if the solver does have support for optional variables then stronger propagation can be achieved.

**Solver implementations of optional variables**

Solver implementations for various forms of optional variables have existed for some time. For example, *time-interval variables* [74] can be used in ILOG CP Optimizer to represent the execution interval of an optional task. If the contained *start*, *end* and *duration* domains become inconsistent, this does not immediately trigger failure. Instead the time-interval *execution status* is set to false. As another example, in a *dynamic constraint satisfaction* solver [84] used for configuration problems, variables are classified at all times as either *active* or *inactive*. Special constraints called *activity constraints* determine when variables become active based on the values and status of other variables. Only active variables are ever assigned values, and the ordinary constraints on the values of variables, called *compatibility constraints* here, are not considered until all involved variables are known to be active.

Clearly time-interval variables cannot be used for our purpose as they are specific to intervals. The dynamic constraint satisfaction approach is not helpful either, as we do not wish to delay reasoning about the possible values of expressions until they are guaranteed to be active (that is, to occur on the execution path). Furthermore, since we wish to continue using a lazy clause generation solver, we need an implementation of optional variables which is compatible with this technology.

**Optional variables in a lazy clause generation solver**

In a non-explaining solver, it is relatively straight-forward to introduce a new type of variable which does not trigger failure when all ordinary values are removed from its domain, and instead sets a flag to indicate that it does not occur. In a lazy clause generation solver we also need to ensure that we do not learn incompatible literals which when combined will incorrectly cause failure.

**Example** Imagine through one chain of reasoning we learn the literal $[x \geq 5]$. Through another chain of reasoning we might learn $x < 5$. Usually this would be represented with the literal $\neg[x \geq 5]$. If we learn this literal it will cause immediate failure in the SAT engine. If $x$ is optional, this is not the desired behaviour. Instead we wish to deduce that $x$ is absent.

We are only aware of one implementation of optional variables in a lazy clause generation solver, which is that described in [100]. Here difficult scheduling problems with optional tasks were solved using optional integer variables to represent task start times. The implementation for optional integer variables was essentially a decomposition using two ordinary integer variables and a Boolean variable. The two integer variables are interpreted as representing the
upper and lower bound of the optional variable, while the Boolean variable indicates whether or not the variable occurs. As soon as the lower bound is known to be greater than the upper bound, the optional variable must be absent.

For an optional variable \( \text{optvar} \) with domain \( \{x..y\} \), the following ordinary variables are created:

- \( \hat{\text{ub}} \) with domain \( \{x-1..y\} \)
- \( \hat{\text{lb}} \) with domain \( \{x..y+1\} \)
- \( \text{occurs} \) with domain \( \{\text{true}, \text{false}\} \)

Channeling constraints between these variables ensure that in a solution either \( \hat{\text{ub}} = \hat{\text{lb}} \) and \( \text{occurs} \) is true, or \( \hat{\text{ub}} = x-1, \hat{\text{lb}} = y+1 \) and \( \text{occurs} \) is false.

Propagators interact with the \( \text{optvar} \) variable in the following ways.

- Upper bounds are imposed only on the \( \hat{\text{ub}} \) variable, and propagators must never impose a bound lower than \( x-1 \) as this would cause failure. If the true bound is lower, imposing a bound of \( x-1 \) is sufficient as this already forces \( \text{optvar} \) to be absent because \( \hat{\text{ub}} < \hat{\text{lb}} \).
- Similarly, lower bounds must be posted using the \( \hat{\text{lb}} \) variable, and never with a bound greater than \( y+1 \).
- Any explanations using the bounds of \( \text{optvar} \) also use literals from the corresponding bound variable. This ensures that conflicting upper and lower bounds do not immediately cause failure in the SAT engine as they use a different set of literals. Instead, if incompatible bounds are deduced, the channeling constraints set \( \text{occurs} \) false.

Due to the nature of the application, the work in [100] focused on bounds propagation. However, value propagation was discussed briefly and is possible using this representation.

- If \( v \) is not a valid value for \( \text{optvar} \), we can set \( \neg [\hat{\text{ub}} = v] \) and \( \neg [\hat{\text{lb}} = v] \). If an explanation uses the fact that \( \text{optvar} \neq v \), either of these literals will be sufficient to include in the explanation as they are linked through the channeling constraints.
- Conversely, if \( v \) is the only valid value for \( \text{optvar} \), we cannot set \( [\hat{\text{ub}} = v] \) or \( [\hat{\text{lb}} = v] \) as this will force \( \text{optvar} \) to occur. Instead we simply set the upper and lower bound of \( \text{optvar} \) to \( v \) using the method for bounds propagation described above. Unfortunately when we later use the fact that \( v \) is the only valid value for \( \text{optvar} \) in an explanation we must include in the clause the literals corresponding to both bounds, which introduces an extra disjunction.
7.4.2 Applying option types to our problem

Optional variables have an obvious application to our problem. The chosen execution path determines which state queries and expressions will be computed. If an expression is never computed then its value is clearly irrelevant. So all state query results, and all expressions built from these (which includes all match values and the values associated with changes) are represented using optional variables instead of regular decision variables.

The occurrence of each optional variable is tied to the successor variable associated with its block in the execution graph. These constraints ensure that the variable belonging to an expression in block \( b \) occurs in a solution iff \( b \) is included on the chosen execution path (that is, \( \text{succ}[b] \neq b \)).

Benefits of optional variables

By using optional variables, we not only solve the problem of queries having no valid value, but also achieve stronger propagation in comparison to the decomposition, by removing the need for a default value.

As an example, consider the variables and constraints we would use to represent a call to the `example` method shown below.

```java
int example(boolean b, int y) {
    if (b)
        return not3(y) + 1;
    else
        return y;
}
```

```java
int not3(int y) {
    if (y == 3)
        throw new Exception();
    else
        return y;
}
```

Looking at the `not3` method, we can see that if the chosen value for \( y \) is 3, then there is no valid return value for the call `not3(y)`. However, even though in our application an exception being thrown does invalidate an execution path, it is not correct to enforce \( y \neq 3 \). It is perfectly valid for \( y \) to take the value 3 if \( b \) is false, as in this case the call to `not3` will not occur. So using our previous approach the variable representing the result of this call would be assigned a default value. Let that variable be called `ret_not3`, and assume we choose 0 as the default value. Also let `ret_example` be the variable representing the return value for the `example` method.

Until \( b \) is known, the call `not3(y)` may not happen, and so the default value remains. That is, \( 0 \in D(\text{ret_not3}) \). Also until \( b \) is known we don’t know which return statement will be executed, so \( D(\text{ret_example}) = D(\text{ret_not3}+1) \cup D(y) \), and therefore \( 1 \in D(\text{ret_example}) \). This means that until we know the value of \( b \) we will think that `example` might return 1, even if we know \( y > 1 \). The link between lower bounds of \( y \) and lower bounds of `ret_example` has been completely broken, which could be especially problematic if `ret_example` is involved in our objective function.

If we instead use an optional variable for `ret_not3`, then although we won’t know whether or not this variable occurs until \( b \) is known, we will know that
Contrary to the decision made for MiniZinc, we do not wish to use identity lifting for the addition function. If we did so, then $D(\text{ret\_not3+1})$ would include 1 until \text{ret\_not3} was known to occur. This would reproduce exactly the problem we encountered before. Instead, we expect that $D(\text{ret\_not3+1}) = \{v + 1\mid v \in D(\text{ret\_not3})\}$, and that if \text{ret\_not3} is absent then so is \text{ret\_not3+1}. This implies $D(\text{ret\_not3+1}) \subseteq \{v + 1\mid v \in D(\text{y})\}$ and $D(\text{ret\_example}) \subseteq \{v + 1\mid v \in D(\text{y})\} \cup D(\text{y})$, thus preserving the link between the lower bound of \text{y} and that of \text{ret\_example}.

### 7.4.3 Implementing optional variables in Chuffed

In order to achieve the goals discussed above, we have implemented optional variables natively in the lazy clause constraint solver Chuffed [32].

The decomposition approach taken in [100] and described above has two disadvantages. Firstly, every propagator using optional variables must be aware of that fact in order to avoid imposing bounds which are too strong, and to behave appropriately when one or more of its arguments are absent. Secondly, there is a small loss of explanation strength when reasoning about values rather than bounds, because there is no single literal representing the fact that \text{v} is the only valid value for optional variable \text{optvar}. A pair of bounds must be used instead.

With a native implementation we have been able to address both of these problems without any extra overhead. We define an optional integer variable with exactly the same interface as an ordinary integer variable.

### Internal representation

Internally, an optional variable \text{x} with initial domain $\{l..u\}$ maintains the following literals:

- $[\text{occurs}(x)]$ (\text{x} occurs)
- $[x \leq v]$, \quad $l \leq v \leq u$ (\text{x} is less than or equal to \text{v} or absent)
- $[x \geq v]$, \quad $l \leq v \leq u$ (\text{x} is greater than or equal to \text{v} or absent)
- $[x = v]$, \quad $l \leq v \leq u$ (\text{x} is equal to \text{v} or absent)
- $[x \neq v]$, \quad $l \leq v \leq u$ (\text{x} is not equal to \text{v})

In a solution, if \text{x} is absent then all literals except $[\text{occurs}(x)]$ will be true. Otherwise, $[\text{occurs}(x)]$ is true and \text{x} must be assigned a value \text{u}. Of the equality literals only $[x = u]$ will be true, and of the disequality literals only $[x \neq u]$ will be false. $[x \leq v]$ will be true for $v \geq u$, while $[x \geq v]$ will be true for $v \leq u$. The variable itself is responsible for ensuring this by enforcing the channeling constraints shown in Figure 7.4.
\begin{align*}
\lceil x \leq v \rceil & \rightarrow \lceil x \leq v + 1 \rceil, \\
\lceil x \geq v \rceil & \rightarrow \lceil x \geq v - 1 \rceil, \\
\lceil x = v \rceil & = \lceil x \leq v \rceil \land \lceil x \geq v \rceil, \\
\lceil x \neq v \rceil & = \lceil x \leq v - 1 \rceil \lor \lceil x \geq v + 1 \rceil, \\
\lceil x \neq l \rceil & = \lceil x \geq l + 1 \rceil \\
\lceil x \neq u \rceil & = \lceil x \leq u - 1 \rceil
\end{align*}
\begin{align*}
\neg \lceil \text{occurs}(x) \rceil & \rightarrow (\lceil x \leq v \rceil \land \lceil x \geq v \rceil \land \lceil x \neq v \rceil \land \lceil x = v \rceil), \\
\lceil \text{occurs}(x) \rceil & \rightarrow (\lceil x = v \rceil = \neg \lceil x \neq v \rceil), \\
\lceil \text{occurs}(x) \rceil & \rightarrow (\lceil x \geq v \rceil = \neg \lceil x \leq v - 1 \rceil),
\end{align*}

Figure 7.4: Channeling constraints enforced by optional integer variables.

**External interface**

There are no changes required to the way propagators interact with variables. The method of posting domain changes, querying the current domain, and requesting explanation literals is identical to that for ordinary integer variables.

The variable itself takes care of directing domain changes to the correct set of literals, and interprets extreme bound changes (or any other domain-emptying change) as equivalent to setting the variable absent.

For explanation, fortunately a distinction already existed between requests for literals meaning e.g. $x > 5$ and $x \leq 4$, and between requests for $x = 5$ and $x \neq 5$, even though previously these would return the same literal (just with opposite sign). The optional variable is therefore able to respond to requests using the correct set of literals.

The use of separate equality and disequality literals allows value changes to be handled correctly in the same way as bound changes, and eliminates the loss of explanation strength observed in the decomposition approach. Note that the number of literals used is essentially the same as the decomposition. An optional integer variable has twice the number of literals used by an ordinary integer variable.

**Correct handling of absent variables**

We have stated that optional integer variables present the same interface to propagators as ordinary integer variables, but it is still necessary to ensure that absent values are handled correctly. For this we are able to take advantage of a unique property of our particular application. Recall that the occurrence of each optional variable is tied to its position in the execution graph. If an expression belongs to a code block which falls on the chosen execution path, then its variable must occur, otherwise it is absent. In our models, the only constraint with arguments originating from different blocks is the reaching definitions constraint.
which links state changes and state queries. All other constraints operating on optional variables simply build terms from one or more state queries originating from the same block of code. Since these expressions all belong to the same block, either they all exist in a solution or they are all absent, and this will be enforced by the constraints linking their occurrence to the execution graph.

This allows us to avoid changing the propagation algorithms for existing propagators. While all of its arguments may occur the propagator operates as normal. If it or another propagator removes the last value from one of its arguments, the changed internal behaviour of the variable will simply set that variable to be absent rather than triggering failure. This in turn will cause all other arguments for this constraint to be absent. After this point the propagator can continue to run and no propagation it produces will have any effect (as the domain of all of its variables is already empty). Actually since there will be no further domain changes for any of its arguments the propagator should not be woken to run at all.

Note that this bears a resemblance to conditional constraint satisfaction in that a constraint with an absent argument is irrelevant. However, here propagators are executed until an argument is known to be absent, which is the complement of the conditional constraint satisfaction approach where a constraint is not considered until all of its variables are known to occur.

The reaching definitions constraint does not behave this way as its arguments originate from different blocks in the execution graph. The reaching definitions propagators are designed specifically to operate on optional variables, and they are the only propagators in Chuffed which are aware that they do so.

Optional Booleans

We also introduce an optional Boolean variable, which is essentially a three valued variable. Obviously in normal circumstances two binary variables are sufficient to represent a choice between three values. However, our optional Boolean variable is comprised of three literals, not two. This is because effective explanations require all three of the following literals (none of which is the negation of another):

- true or absent (variable cannot be false)
- false or absent (variable cannot be true)
- occurs (value unknown but variable must occur)

In a solution, exactly one of these three literals will be false. Internal channeling ensures that their values are kept consistent.

Implementing optional Boolean variables required some architectural changes in Chuffed to remove the assumption that a Boolean variable is equivalent to a literal. It was also necessary to adjust the way explanation literals were requested to force propagators to differentiate between ‘true’ and ‘not false’. Unlike integer variables propagators commonly requested a literal and then negated
it instead of requesting the negated literal. Unfortunately solving this problem required carefully checking each request to determine which literal was actually desired and then modifying the code to specifically request that literal. It was however still not necessary to change any of the propagation or explanation reasoning.

**Declaring optional variables**

Option types were not yet supported by MiniZinc when we began our implementation, so rather than using the type system, we specify that a variable is optional using an annotation on the variable declaration. We also define a predicate to specify the occurrence condition for an optional variable, and we use this to link the occurrence of each optional variable to the inclusion of its block on the execution path.

Special care must be taken to ensure that any intermediate variables introduced during the conversion to FlatZinc are declared and constrained correctly. If a constraint on optional variables introduces a new intermediate variable then that variable must also be declared optional and constrained to occur iff the original arguments to the constraint occur.

In some cases intermediate variables are also introduced in Chuffed while parsing constraints. For this reason we perform a check during parsing to see if a constraint has optional arguments. If so we use optional variables for any intermediate variables and constrain their occurrence appropriately.

### 7.5 Revisiting Special Case Translations

The following sections will introduce our implementation for the **reaching definitions** global constraint, which replaces the original constraints used for state queries in the general case. Here we first briefly revisit the **sum** and **max/min** special cases applicable when the assignments and queries match a specific pattern. These special cases can still be applied when using the global constraint, but the constraints require some adjustment due to the use of optional variables. We have not reimplemented the **bool** special case, as this reduced overhead but did not change the propagation strength. For a fair comparison, in all experiments in this chapter we have also disabled the **bool** special case when using the decomposition.

#### 7.5.1 Constraints for **sum** calculations

We first consider the **sum** special case, for which we have previously used a linear sum constraint as shown below. This special case is applicable when the relevant changes for the given query consist of a change giving an initial value followed by some number of changes whose values are composed of a query to the same field with an expression added or subtracted.
**field reference:** qobj.field

**assignments:**

\[
\text{cond}_i : \text{obj}.f \text{iel}d := \text{obj}.f \text{iel}d + \text{expr}_i, \quad i \in 1..n
\]

**constraint:**

\[
\text{resultq} = \sum([\text{init}] + [\text{expr}_i \times \text{bool2int}(\text{cond}_i \land \text{obj}_i = \text{qobj}) \mid i \in 1..n])
\]

The \textit{sum} special case is still applicable when using the global constraint. However, unless all relevant changes are guaranteed to occur, we cannot simply use the same constraint, because the arguments to \textit{sum} do not satisfy the property that if any argument is absent then all are absent. Therefore the linear sum propagator may produce invalid propagation. In fact, not only do the arguments for the \textit{sum} constraint fail to satisfy this condition, but the reified equality constraints used to test matchingness have the same problem. It would be possible to implement new \textit{sum} and reified equality propagators capable of working correctly with arbitrarily optional variables, but we have not done this. Instead we simply use identity lifting, as shown below.

**constraint:**

\[
\text{resultq} = \sum([\text{init}] + [\text{lift}(\text{expr}_i, 0) \times \text{bool2int}(\text{lift}(\text{obj}_i, v_i) = \text{lift}(\text{qobj}, v_q)) \mid i \in 1..n])
\]

The \text{lift}(\text{var},\text{val}) function introduces a new variable constrained to equal \text{var} unless \text{var} is absent, in which case it equals \text{val}. For the match variable for assignment \(i\) we use an arbitrary identity \(v_i\) taken from the variable domain (currently the minimum value). Note that we no longer need to include the execution condition directly, as the lifted value expression will be 0 if the change does not occur.

This approach is simple, but it does weaken the propagation (and explanation) power of the \textit{sum} constraint compared with a purpose-written constraint. An optional \textit{sum} constraint would have a wide range of applications, so in future it would probably be worthwhile implementing this constraint.

### 7.5.2 Constraints for \textit{max/min} calculations

The \textit{max/min} special case was previously applied when every non-initialisation assignment was conditional on a greater or less than comparison between the new and previous value. When there were no other assignment conditions, and all changes were known to match the query, we used a \textit{max/min} constraint as shown below.

**assignments:**

\[
\text{max} := \text{init} \\
(\text{value1} > \text{max}) : \quad \text{max} := \text{value1} \\
(\text{value2} > \text{max}) : \quad \text{max} := \text{value2}
\]

**constraint:**

\[
\text{finalmax} = \max([\text{init}, \text{value1}, \text{value2}])
\]

When extra execution conditions were present, or if matchingness was not certain, we used an individual greater or less than constraint for each change, to ensure that if that change did occur and match the query, then the final query result would be greater/less than the assigned value. A final disjunction ensured that the result did indeed equal one of the assigned values.
When applying this special case in our new scheme, the first important observation is that rather than using the value queries from the assignments, we should be using the equivalent expression from the comparison. Consider the code snippet below.

```plaintext
1 int max = init;
2 if(v1 > max)
3    max = v1;
4 if(v2 > max)
5    max = v2;
6 return max;
```

Assuming this code block is guaranteed to be executed, it can still be translated using the simple version of the max special case, with the following constraint.

```
constraint: retval = max([init, v1, v2])
```

However, for the values v1 and v2, we must not use the variables representing the results of the v1 and v2 queries in lines 3 and 5 of the code. These variables are absent if the corresponding assignment does not occur. Instead we should use the variables representing the v1 and v2 queries on lines 2 and 4 (inside the comparison expression). Then we can use an ordinary max/min constraint since all arguments are known to occur.

If the entire code block is only conditionally executed (e.g. if it defines a method which may or may not be called), then the v1 and v2 queries and the return value will all be optional. Previously we would still have used the simple version of the max/min special case, because the extra execution conditions would be eliminated due to the scope of the max variable (see Section 3.2.4). Now we instead apply the weaker translation, adjusted for optional variables as discussed below.

**Adjusting weak max/min for optional variables**

When applying the weaker translation, we now always include the ordinary (non special case) constraints as well. This replaces the final disjunction used previously and means we can consider separately whether or not to add a constraint for each assigned value. To add a constraint, we require that the assignment cannot occur after the query, so that if both the comparison node and the final query happen, then the query result must be greater/less than or equal to the assigned value. Assuming the max case, the basic form of constraint we use is the following. Note that as for the simple case, the value$_i$ variable is taken from the comparison not the assignment.
constraint:  
\[(\text{lift}(\text{finalmax}, \text{lb}(\text{finalmax})) \geq \text{lift}(\text{value}_i, \text{ub}(\text{value}_i))) \lor \\
(\text{lift}(\text{obj}_i, v_i) \neq \text{lift}(\text{qobj}, v_q)) \lor \\
\text{absent}(\text{value}_i) \lor \text{absent}(\text{finalmax})\]

Obviously, only optional variables are lifted. The match variables are lifted as for the sum special case, the value variable uses its upper bound as a default value, and the result its lower bound (these are swapped for the min special case). If we know that \(\text{absent}(\text{finalmax}) \rightarrow \text{absent}(\text{value}_i)\), then we can drop the \(\text{absent}(\text{finalmax})\) disjunct. This is true if the final query anti-dominates the \(\text{value}_i\) query in the execution graph. Similarly, if the \(\text{value}_i\) query dominates the result query we can drop the \(\text{absent}(\text{value}_i)\) disjunct, since in this case \(\text{absent}(\text{value}_i) \rightarrow \text{absent}(\text{finalmax})\).

As with the sum constraint, it would be possible to implement a special-purpose optional max/min constraint which would propagate more strongly than our approach. Ideally it should incorporate the matchingness check as well, to make use of the known relationship between the occurrence of match variables and of the changes and query (match variables must occur if the associated query/change does).

### 7.6 Implementing the Global Constraint

In this section we describe our initial implementation of the reaching definitions constraint. Our goal for this version is to preserve all reasoning achieved by the previously used decomposition, while removing the requirement for a known fixed order for changes and queries, with minimal overhead. Sections 7.8 and 7.9 explain how we can extend the propagators described here, achieving stronger propagation.

#### 7.6.1 Reminder: previously used decomposition

For convenience we list here the constraints used previously (in Chapter 4) for a field reference. Our initial implementation of the reaching definitions constraint aims to achieve the same propagation strength as this decomposition.

Note that the \(\text{index}\) variable in the decomposition defines the possibly reaching changes.

\[
\begin{align*}
\text{field reference:} & \quad \text{objq.field} \\
\text{relevant assignments:} & \quad \text{cond1 : obj1.field := expr1} \\
& \quad \quad \quad \ldots \\
& \quad \quad \quad \text{condn : objn.field := exprn} \\
\text{constraints:} & \quad \text{element(index, [obj1, ..., objn], objq)} \\
& \quad \text{element(index, [cond1, ..., condn], true)} \\
& \quad \text{element(index, [expr1, ..., exprn], result)} \\
& \quad (\text{cond2 \&\& objq = obj2} \rightarrow \text{index} \geq 2) \\
& \quad \quad \ldots \\
& \quad (\text{condn \&\& objq = objn} \rightarrow \text{index} \geq n)
\end{align*}
\]
7.6.2 Architecture

Our implementation of the reaching definitions constraint consists of a collection of propagators with a shared data structure storing all of the relevant variables and parameters, including the condensed execution graph. In the remainder of this chapter, we use the following names for these shared variables and data.

Assuming a call

\[
\text{reachingdefs}(\text{numchanges}, \text{numqueries}, \text{numnodes},
\text{value}, \text{match1}, \text{match2}, \text{from}, \text{to}, \text{cond}, \text{happens})
\]

we store the following variables:

\[
\begin{align*}
val_i &= \text{value}[i], & i &\in 1..\text{numchanges+numqueries} \\
m_{1i} &= \text{match1}[i], & i &\in 1..\text{numchanges+numqueries} \\
m_{2i} &= \text{match2}[i], & i &\in 1..\text{numchanges+numqueries} \\
hap_i &= \text{happens}[i], & i &\in 0..\text{numnodes-1} \\
E &= \{(\text{from}[i],\text{to}[i],\text{cond}[i]) | i \in 1..\text{size}(\text{cond})\} \\
rdef_q, & q \in \text{numchanges+1..numchanges+numqueries}
\end{align*}
\]

The \textit{rdef} variables are created within the solver to keep track of the possibly reaching changes for each query. In a solution, \textit{rdef} \textit{q} gives the index of the reaching change for \textit{q} (note this is equivalent to the \textit{index} variable in the decomposition). Unless \textit{q} is known to happen, \textit{rdef} \textit{q} is optional, and constrained to occur iff \textit{hap} \textit{q} is true.

We continue to use element constraints to ensure that the reaching change matches the query and happens, and that the query result equals the value associated with this change. These constraints are still valid as they do not depend on the assumed fixed execution order. However, we do need to adjust the element propagator slightly to correctly handle optional variables. Given that the index and result have the same occurrence conditions, the only required change is that we delay backwards propagation from the result to the chosen element until the index is not only fixed but also known to occur.

**Example** Consider a query \textit{q} with three potentially reaching changes. We use the constraint \textit{element}(\textit{rdef} \textit{q}, \textit{[val} \textit{1, val} \textit{2, val} \textit{3, val} \textit{q}) to ensure that the query result corresponds to the reaching change value. If at some point \textit{D(\textit{rdef} \textit{q}) is reduced to \{1\}, then the standard element propagator would set \textit{D(\textit{val} 1) = D(\textit{val} 1) \cap D(\textit{val} q)}. Essentially this says that the change cannot set a value other than that retrieved by the query. However, this is only correct if the query is actually executed. So our modified element propagator waits until \textit{rdef} \textit{q} is known to occur (and therefore the query is known to be executed) before performing this propagation.

To replace the implication constraints, which do depend on the assumed execution order, we introduce several new propagators. A query propagator for each query implements the forward reasoning, while a change propagator
for each change, in cooperation with a single reachOrNotMatch propagator, implement the backwards reasoning.

7.6.3 Basic query propagation

As stated above, the query propagator is responsible for achieving the forward reasoning previously provided by implication constraints in the decomposition. Each implication constraint:

\[(\text{cond}, \land \text{obj}_q = \text{obj}_i) \rightarrow \text{index} \geq i\]

provided the following (forwards) reasoning:

**Rule:** If change \(c\) with index \(i\) must happen and must match query \(q\), then no change \(c_2\) with index \(j < i\) can reach \(q\).

We need to generalise this rule to eliminate its dependence on the known execution order for changes and queries. Firstly, we need to remove the reference to the index values, which are no longer meaningful. The statement \(j < i\) means that \(c_2\) cannot be executed after \(c\). There is also an implicit assumption that \(c\) cannot occur after \(q\) (with a known execution order no change ever occurring later than \(q\) is relevant for \(q\)). Combining these two conditions (\(c_2\) before \(c\) and \(c\) before \(q\)), we conclude that if they are all executed \(c\) falls between \(c_2\) and \(q\). Since we also have that \(c\) must happen, this is equivalent to saying that all execution paths leading from \(c_2\) to \(q\) pass through \(c\). Intuitively, \(c_2\) cannot be the reaching change for \(q\) because it will always be overwritten by \(c\).

We therefore generalise the rule as follows:

**Rule:** If change \(c\) must match query \(q\), and all execution paths leading from change \(c_2\) to \(q\) pass through \(c\), then \(c_2\) cannot reach \(q\).

The basicquery algorithm

The query propagator for query \(q\) enforces this rule using the basicquery algorithm, which performs a simple search of the execution graph, starting at the node for \(q\) and traversing backwards following all non-excluded edges, but stopping at any definitely matching change. Any change \(c\) not seen during this traversal is eliminated as a reaching change for \(q\), as no valid execution path leads from \(c\) to \(q\) without passing through an overwriting change.

Note that theoretically this is more general than the rule stated above, as the paths from change \(c\) to \(q\) may be blocked by a set of other changes rather than any single change, but in practice (due to the typical shape of execution graphs) this rarely makes a difference. The basicquery algorithm also eliminates changes which given the current graph can only occur after \(q\). Previously, any change occurring after \(q\) could be removed during pre-processing.
**Example** Consider the execution graph and match variable values shown below. Nodes are labelled by type: \( c \) for changes, \( q \) for queries, and \( n \) for structure nodes. The dashed edge has been excluded.

![Execution Graph Example](image)

<table>
<thead>
<tr>
<th>( D(m_1) )</th>
<th>( D(m_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 )</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>{1}</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>{1}</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>{1}</td>
</tr>
</tbody>
</table>

The basic query algorithm for \( q_2 \) visits nodes \( q_2, n_3, c_3 \) (stopping because \( c_3 \) matches \( q_2 \)), \( n_1 \) and \( c_1 \). Since node \( c_2 \) was not seen, the propagator removes \( c_2 \) from \( D(rdef_{q_2}) \). The propagator for \( q_1 \) visits \( q_1, n_2, c_2, n_1 \), and \( c_1 \). Node \( c_3 \) is not visited so we remove \( c_3 \) from \( D(rdef_{q_1}) \).

### 7.6.4 Basic change propagation

The *change* propagators (in cooperation with the *reachOrNoMatch* propagator) are responsible for replicating the backward reasoning provided by the implication constraints in the decomposition. An implication constraint:

\[
(\text{cond}_i \land \text{obj}_q = \text{obj}_i) \rightarrow \text{index} \geq i
\]

provides the following backwards reasoning:

**Rule:** If query \( q \) cannot be reached by any change with index \( j \geq i \), then change \( c \) with index \( i \) either does not match \( q \), or does not happen.

which is equivalent to:

**Rule:** If query \( q \) cannot be reached by any change with index \( j > i \),
then either change \( c \) with index \( i \) is the reaching change for \( q \),
c does not match \( q \), or \( c \) does not happen.

As with the forwards rule, we need to remove the reference to the change indices, and also make explicit the assumption that \( c \) cannot occur after \( q \), and that \( q \) must happen (recall that previously we assumed all queries happened, and if a default value was necessary this was provided by a dummy change with index greater than all real changes).

We generalise the backwards rule as follows:

**Rule:** If change \( c \) cannot occur after \( q \), and no change \( c_2 \) which may occur after \( c \) can be the reaching change for \( q \), then either \( c \) is the reaching change for \( q \), \( c \) does not match \( q \), \( c \) does not happen, or \( q \) does not happen.
The change propagator for change \( c \) is responsible for finding queries meeting the test conditions of the above rule. That is, any query \( q \) which cannot occur before \( c \), and for which no change \( c_2 \) possibly occurring after \( c \) can be the reaching change. In that case we know that either \( c \) does not match \( q \), \( c \) is the reaching change for \( q \), or one of \( c \) or \( q \) does not happen. Since this disjunction will not necessarily produce any propagation immediately, when such a situation is discovered the change propagator registers the pair with the \textit{reachOrNotMatch} propagator which is responsible for listening to the relevant events and setting a disjunct true once all others are known to be false.

The basicchange algorithm

The basicchange algorithm used by the propagator for change \( c \) works as follows.

1. First we find the set of queries \( Q \) for which we may be able to produce propagation. Of the queries known to this propagator, we keep only queries \( q \) satisfying all of the following conditions.
   - We have not already registered the pair \( \{c,q\} \) with the \textit{reachOrNotMatch} propagator.
   - The query \( q \) was not originally (at the root node) reachable backwards from \( c \) in the execution graph. This ensures that our change \( c \) cannot occur after \( q \). Note that if the graph contains cycles we could achieve stronger propagation by checking for queries which are not \textit{currently} reachable backwards from \( c \), but this would be much more expensive.
   - The query \( q \) may be executed. Otherwise the disjunction we’re aiming to discover is already satisfied. Note that similarly, we do not need to execute this algorithm at all if \( c \) is known not to occur.
   - The pair \( c \) and \( q \) may match. As above, if this is not the case the disjunction is satisfied.

2. If the set \( Q \) is empty we can stop, as no new disjunctions can be discovered. In fact, since reducing domains cannot change any of these conditions from true to false, we know no new disjunctions will be discovered until the solver backtracks.

3. Otherwise we reduce \( Q \) to \( Q' \), removing any query \( q \) for which a possible overwriter \( c_o \) can be found, satisfying the following conditions.
   - The change \( c_o \) may be the reaching change for \( q \) (i.e. \( c_o \in D(rdef_q) \)).
   - The change \( c_o \) was originally reachable from \( c \). Note that once again we could strengthen propagation by checking if \( c_o \) is \textit{currently} reachable from \( c \), but this is unlikely to pay off as we would need to perform many more graph searches.

4. Then, for each query \( q \) in \( Q' \), we register a new pair \( \{c,q\} \) with the \textit{reachOrNotMatch} propagator.
Example Consider the execution graph and variable domains shown below.

![Execution Graph](image)

<table>
<thead>
<tr>
<th></th>
<th>$c_0$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$q_5$</th>
<th>$q_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D(m_1)$</td>
<td>${1}$</td>
<td>${1}$</td>
<td>${1,2}$</td>
<td>${1}$</td>
<td>${2}$</td>
<td>${1}$</td>
<td>${}$</td>
<td>${}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>$D(m_2)$</td>
<td>${1,2}$</td>
<td>${1,2}$</td>
<td>${1,2}$</td>
<td>${1,2}$</td>
<td>${1,2}$</td>
<td>${1,2}$</td>
<td>${}$</td>
<td>${}$</td>
<td>${1,2}$</td>
</tr>
<tr>
<td>$D(hap)$</td>
<td>${t}$</td>
<td>${t, f}$</td>
<td>${t, f}$</td>
<td>${t, f}$</td>
<td>${t}$</td>
<td>${t}$</td>
<td>${f}$</td>
<td>${t}$</td>
<td></td>
</tr>
<tr>
<td>$D(rdef)$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>${c_0}$</td>
<td>${c_0}$</td>
<td>${c_0, c_2}$</td>
<td>${c_1, c_2}$</td>
<td>${}$</td>
<td>${c_0, c_1}$</td>
</tr>
</tbody>
</table>

The basic change algorithm for $c_1$ computes $Q = \{q_4, q_6\}$ ($q_1$ and $q_2$ were originally reachable backwards from $c_1$, $q_3$ cannot match $c_1$, $q_5$ does not happen). It then removes $q_4$ because $c_2$ is an overwriter. Query $q_6$ remains ($c_0$ was never reachable from $c_1$, $c_2 \not\in D(rdef_{q_6})$). So $Q' = \{q_6\}$, and we register $(c_1, q_6)$ with reachOrNotMatch. We do not discover the pair $(c_1, q_2)$.

7.6.5 The change and query propagators

Having introduced the basic propagation algorithms, here we discuss the query and change propagators in more detail, including efficiency considerations. The basic query propagation algorithm in particular can be expensive as it requires a graph search to replace the assumption of a known execution order. To minimize the overhead incurred it is important to carefully consider when this algorithm should be executed.

In Chuffed, as in most propagation-based solvers, the execution of propagators is managed using events. Each propagator registers itself as interested in certain domain changes for particular variables. When the domain of a variable changes, this raises an event, which prompts the solver to wake propagators interested in that domain change. It is also possible to flag a propagator as satisfied, in which case it will not be executed even if relevant events occur. We use both of these mechanisms, as well as some trailed state (trailed meaning changes are undone upon backtracking) to reduce the number of calls to our propagation algorithms.

The query propagator

The basic query propagation algorithm performs a backwards search of the execution graph. Before executing this algorithm, the query propagator first attempts to follow fixed edges backwards from the query node $q$. It then records the node reached and the reason for stopping in a trailed state variable, to avoid repeating this part of the traversal in future executions. After the fixed edge traversal,
the full basicquery algorithm is executed (starting at the last node reached along
fixed edges). To avoid missing propagation, the fixed edge traversal must not
move past a change node which could block other changes from reaching our
query node \( q \). Therefore, if a change \( c \in D(rdef_q) \) is discovered on this fixed
path, we do not traverse further.

Note that we may traverse past a possibly matching change if it is known for
some other reason (perhaps due to the assigned value) not to reach \( q \). This ac-
tually gives us a further opportunity for propagation. Any change \( c \) encountered
during fixed path traversal must either reach \( q \) or not match it, unless \( q \) does
not occur. It cannot be overwritten as we never move past a possibly reaching
change. Therefore we can register the pair \( \{ c, q \} \) with the reachOrNotMatch
propagator, as is done in the basicchange algorithm. Actually we have slightly
stronger information in this case. We know all paths to \( q \) must arrive through \( c \)
(because there is a fixed path back from \( q \) to \( c \)), so it is not possible for \( q \) to oc-
cur unless \( c \) occurs. Therefore the reachOrNotMatch propagator does not need
to separately consider \( c \) not occurring, so we specify when registering the pair
that only the query not occurring should be allowed to satisfy the disjunction.

It is also possible to detect two cases where the query propagator can provide
no further propagation. Firstly, if \( q \) does not happen, then \( rdef_q \) will be absent,
which means \( D(rdef_q) = \{ \} \), so obviously no further changes can be removed.
In this case we set the propagator satisfied. Similarly, if the fixed edge traversal
stops at a possibly reaching change, and this is the only possibly reaching change
for \( q \), then there is no need to execute the basicchange algorithm as there are no
further changes to exclude. Instead we again simply set the propagator satisfied.
Figure 7.5 gives pseudo-code for the query propagation function.

**Example** Consider the execution graph shown below. The first time the query
propagator for \( q \) is woken, fixed edge traversal will follow the edge back to \( n_1 \)
and stop there (as it is a branch), and then the basicquery algorithm will be
executed starting at this node. Subsequent executions of the propagator will
continue to simply call basicquery starting at \( n_1 \), until one of the edges leading
in to \( n_1 \) is eliminated.

Let us assume at some point the edge \( c_4 \rightarrow n_1 \) is eliminated, and that at this
time \( D(rdef_q) = \{ c_1, c_2, c_4 \} \). The query propagator will then move its fixed
dge traversal past \( n_1 \) and \( c_3 \), stopping at \( c_2 \) (because \( c_2 \in D(rdef_q) \)), and will
register both \( (c_3, q) \) and \( (c_2, q) \) with the reachOrNotMatch propagator, before
calling basicquery starting at \( c_2 \). Subsequent executions of the propagator will
continue to call basicquery starting at \( c_2 \), until \( c_2 \notin D(rdef_q) \), in which case
fixed edge traversal will move to \( c_1 \), or \( |D(rdef_q)| \leq 1 \), in which case we will set
the propagator satisfied.

160
propagateQuery() {
    if(!mayHappen(q)) {
        satisfied = true;
        return;
    }
    current = currentNode(); // look up from trailed state
    while(true) {
        // if current is a possible overwriter (according to trailed state)
        if(atPossReach()) {
            if(rdef[q].isFixedTo(current)) {
                satisfied = true; // nothing left to exclude
                return;
            }
            // if current cannot reach q, move past it
            if(!rdef[q].indomain(current)) {
                setAtBranch(current);
            }
            // if current may still reach q, stay here
            else {
                basicquery(current); // start search at current
                return;
            }
        }
        // if current is not an overwriter
        else {
            // if there’s exactly one predecessor step backwards
            if(preds(current).size() == 1) {
                current = preds(current).first();
                if(isChangeNode(current)) {
                    // current reaches q, does not match q, or q does not happen
                    reachOrMatchQ.registerQ(current, q);
                    // update state: stay here until current is not an overwriter
                    setAtPossReach(current);
                }
                else {
                    // update state: attempt to keep following fixed edges
                    setAtBranch(current);
                }
            }
            // if the path back is not fixed, stay here
            else {
                basicquery(current); // start search at current
                return;
            }
        }
    }
}

Figure 7.5: Propagation function for the query propagator (for query q).
We have also carefully considered which events the query propagator should listen to. Clearly graph changes are relevant, but we only need to listen to the deletion of edges traversed in an initial execution of basicquery. Match arguments are also relevant, as when a change becomes known to match our query the search will stop at that node, so we listen to events from \( m_{1q} \) and \( m_{2q} \), as well as match variables for changes, but again only those seen in an initial execution of the algorithm. For the fixed path exploration, we need to listen to domain changes for \( rdef_q \), as these allow us to move past changes.

The change propagator

Although the basicchange algorithm does not perform a graph search, the change propagator performs a fixed path traversal very similar to the query propagator. This time we follow fixed edges forwards from our change \( c \), stopping if a possibly matching change is discovered. Any query encountered along this fixed path must either be reached by \( c \) or not match it, unless \( c \) does not occur. As is done in the query propagator, we immediately register this information with the reachOrNotMatch propagator. In this case we know that the query must occur if \( c \) does, so this time we note when registering the pair that only the change not occurring should satisfy the disjunction.

If the fixed path traversal stops at a possibly matching change \( c_m \), then for efficiency we do not execute the basicchange algorithm. Any queries earlier on the fixed path have already been registered, and \( c_m \) will usually be a possible overwriter for later queries. In theory this may lead to missed propagation (because \( c_m \) may be known not to reach some later query and therefore not be an overwriter after all), but our experimental results (see Section 7.6.7) show the effect on propagation strength is negligible.

If the fixed path traversal reaches a definitely matching change, then no further propagation will be made until the solver backtracks, so we set the propagator satisfied. We also set the propagator satisfied within the basicchange algorithm if the set \( Q \) is empty (as discussed when describing that algorithm, if \( Q \) is empty no further propagation is possible). Pseudo-code for the change propagation algorithm is shown in Figure 7.6.

Example

Assuming the execution graph shown below, initially the change propagator for \( c_1 \) will follow the fixed edge to \( q_1 \), register the pair \((c_1,q_1)\) with the reachOrNotMatch propagator, and then call basicchange starting at \( q_1 \).

If subsequently the edge \( q_1 \rightarrow c_3 \) is eliminated, the change propagator will follow the now fixed edge to \( c_2 \). Let us assume that at this point \( c_2 \) may match \( c_1 \). In this case we will stop here and not call basicchange. If at some later point \( c_2 \) is known not to match \( c_1 \), then the fixed edge traversal will continue on to \( c_3 \). Alternatively, if \( c_2 \) must match \( c_1 \), then this propagator is satisfied.
propagateChange() {
    if(!mayHappen(c)) {
        satisfied = true;
        return;
    }
    current = currentNode(); // look up from trailed state
    while(true) {
        // if current is a possible overwriter (according to trailed state)
        if(atPossMatch()) {
            if(mustMatch(current, c)) {
                satisfied = true; // no further propagation possible
                return;
            }
        }
        // if current cannot match c, move past it
        else if(!mayMatch(current, c)) {
            // update state: attempt to follow fixed edges
            setAtBranch(current);
        }
        // if current may still match c, stay here
        else {
            // stopped at a possibly matching change, do not call basicchange
            return;
        }
    }
    // if current is not an overwriter
    else {
        // if there’s exactly one successor, step forwards
        if(succs(current).size() == 1) {
            current = succs(current).first();
            if(isChangeNode(current)) {
                // update state: stay here until current is not an overwriter
                setAtPossMatch(current);
            }
            else {
                if(isQueryNode(current)) {
                    // c reaches current, doesn’t match current, or doesn’t happen
                    reachOrNotMatch.registerC(c, current);
                }
            }
        }
    }
    // if path forward is not fixed, stay here
    else {
        basicchange(current);
        return;
    }
}

Figure 7.6: Propagation function for the change propagator (for change c).
The final efficiency consideration for the change propagator is the waking conditions. For the basic change algorithm to discover a new pair to register with the reachOrNotMatch propagator, we need a change which was a possible overwriter for a given query to cease to satisfy those requirements. The fixed path traversal relies on edge removal and other changes becoming known not to match c to move on. We therefore listen to the following events.

- For all pairs of changes and queries $(c_o, q)$ where both $c_o$ and $q$ were originally reachable from $c$, the removal of $c_o$ from $D(rdef_q)$.
- Upper and lower bound events for $m_{1c}$ and $m_{2c}$, and the match arguments for all other changes which were originally reachable from $c$.
- Edge removal, but only for edges seen during an initial forwards traversal from $c$.

As a further optimisation, upon receiving a match variable event, we wake only if the match variable belongs to $c$ or the current node (in the fixed path traversal). Similarly, edge deletion only actually wakes the propagator if the edge originates from this current node.

7.6.6 The reachOrNotMatch propagator

As stated earlier, along with the query and change propagators we also create a single reachOrNotMatch propagator for each reaching definitions constraint. This propagator receives notifications from the change and query propagators when it is discovered that a change $c$ cannot occur after or be overwritten for a query $q$. In this case we know that either $c$ is the reaching change for $q$, $c$ does not match $q$, or one of $c$ or $q$ does not happen. In some cases we have extra information about the relationship between the occurrence of $c$ and $q$, so we can drop one of these from the disjunction.

The reachOrNotMatch propagator stores a state variable for each change-query pair, indicating whether the pair has been registered with the complete disjunction, with one of the stronger disjunctions, or any combination of these three (including the case where it has yet to be registered at all). A final value is reserved to indicate that the complete disjunction is satisfied. These state variables are trailed, so that if the solver backtracks or backjumps past the registration of a change-query pair, the state will revert to unregistered and the reachOrNotMatch propagator will cease to perform propagation for that pair. In the remaining discussion we assume the following functions are used to access these state variables.

**isSat(c, q):**

$$((rdef_q = c) \lor (m_{1c} \neq m_{1q}) \lor (m_{2c} \neq m_{2q}) \lor \neg hap_c \lor \neg hap_q) \text{ is satisfied}$$

**isRegCQ(c, q):**

$$((rdef_q = c) \lor (m_{1c} \neq m_{1q}) \lor (m_{2c} \neq m_{2q}) \lor \neg hap_c \lor \neg hap_q) \text{ has been registered}$$
isRegQ(c, q):

\[(rdef_q = c) \lor (m_{1c} \neq m_{1q}) \lor (m_{2c} \neq m_{2q}) \lor \neg hap_q)\] has been registered

isRegC(c, q):

\[(rdef_q = c) \lor (m_{1c} \neq m_{1q}) \lor (m_{2c} \neq m_{2q}) \lor \neg hap_c)\] has been registered

Propagation

The propagation algorithm for the reachOrNotMatch propagator is very simple. Its task is to check for cases where all but one disjunct for a registered disjunction are known to be false, and then set the remaining disjunct true. The pseudo-code is shown in Figure 7.7. Note that disequalities between two variables cannot be set directly, so to enforce non-matchingness we need one pair of match variables to be known equal and one of the other pair to be fixed to some value $v$. We can then remove $v$ from the domain of the remaining match variable. There is also some subtlety in the use of setReaches(c, q). This function removes all values except $c$ from $D(rdef_q)$. Recall that $rdef_q$ will be absent (with no value) if $hap_q$ is false. Therefore calling setReaches(c, q) is equivalent to setting \((rdef_q = c) \lor \neg hap_q\). The function notReaches(c, q) tests if $c$ is excluded as a reaching change for $q$ (so is equivalent to $rdef_q \neq c$).

After attempting propagation, if any of the disjuncts are known to be true then we set the state of this pair to satisfied. This includes if $c$ is the only possible reaching change for $q$, as in this case we have \((rdef_q = c) \lor \neg hap_q\).

Waking

The reachOrNotMatch propagator must listen for changes to match arguments, happens variables, and index (rdef) variables. To avoid excessive executions of the propagateReachOrNotMatch algorithm, the reachOrNotMatch propagator maintains a Boolean array keeping track of which pairs may produce propagation. When an event is reported, this array is updated for the affected pairs, as detailed in the table below. Then when the propagator is executed, propagateReachOrNotMatch is only applied to pairs for whom a relevant event has occurred.

<table>
<thead>
<tr>
<th>Event</th>
<th>Affected pairs</th>
</tr>
</thead>
</table>
| $m_{1c}$ or $m_{2c}$ fixed | \{c, q\} where \!
|                        | \!
|                        | \!
|                        | \!
| $m_{1q}$ or $m_{2q}$ fixed | \{c, q\} where \!
|                        | \!
|                        | \!
| $hap_c$ set true       | \{c, q\} where \!
|                        | \!
|                        | \!
| $hap_q$ set true       | \{c, q\} where \!
|                        | \!
|                        | \!
| $c \notin D(rdef_q)$   | \{c, q\} if \!
|                        | \!
|                        | \!

165
propagateReachOrNotMatch(c, q) {
    if (mustMatch(c, q)) {
        if (isRegQ(c, q))
            setReaches(c, q);
        else if (isRegC(c, q) && mustHappen(c))
            setReaches(c, q);
        else if (isRegCQ(c, q) && mustHappen(c))
            setReaches(c, q);
        else if (!mayReach(c, q)) {
            if (isRegC(c, q))
                setNotHappens(c);
            else if (isRegCQ(c, q)) {
                if (mustHappen(c))
                    setNotHappens(q);
                else if (mustHappen(q))
                    setNotHappens(c);
            }
        }
    } else if (!mayReach(c, q)) {
        if (isRegQ(c, q) && mustHappen(q))
            attemptSetNotMatch(c, q);
        else if (isRegC(c, q) && occurs(c))
            attemptSetNotMatch(c, q);
        else if (isRegCQ(c, q) && occurs(c) && occurs(q))
            attemptSetNotMatch(c, q);
    }
}

Figure 7.7: Propagation function used by the reachOrNotMatch propagator for a change-query pair \{c, q\}.

### 7.6.7 Evaluation

We compared this initial implementation of the reaching definitions global constraint with the decomposition experimentally, to verify that it successfully maintains the previously achieved propagation strength. All experiments in this chapter were performed on a 3.2 GHz Intel i5-6500 with 8GB RAM, and used a five minute timeout. In the following discussion we use decom and decom+ to denote the decomposition without and with special cases respectively, and glob and glob+ for the global constraint (again without and with special cases).

Note that the decom and decom+ models are not identical to the equivalent models in earlier chapters (e.g. new and new+ in Chapter 4). We have reimplemented the model generation component of our system to allow both the decomposition and global translations to be performed with minimal differences. This ensures that the experiments we perform in this chapter are not affected by extraneous differences, but does mean there are minor changes to the models.
produced using the decomposition technique compared with previous chapters. These changes are however limited to details not discussed in this thesis, such as the exact set of expression simplifications performed.

**Comparison without special cases**

As can be seen in Table 7.1, the propagation strength of **glob** is almost identical to that of **decom**. Note that the smaller number of failures for **glob** for larger sizes of some benchmarks is due to timeouts. For **proj3**, **glob** is slightly stronger, probably due to the use of optional variables. However, the execution time is in most cases slightly and in some cases very much worse. We should expect some overhead when removing a useful assumption (i.e. that the changes and queries have a fixed order), but upon investigation we discovered that in the worst cases, the majority of the extra time is actually spent propagating element constraints.

This difference in the performance of element is due to an optimisation during model generation which was more widely applicable using the previous method. Previously, when two queries were known to match, we used the result of the earlier query as a initial value for the later query, inserting a dummy change and removing all earlier changes. This worked because the queries did not have execution conditions, just a location amongst the changes. It also greatly improved efficiency, because the element constraints for the later query did not need to wake and propagate on events associated with the excluded changes. Now queries are tied to a node in the execution graph, and are absent (take no value) if that node does not fall on the execution path. This means we can only use an earlier query as an initialising change if it dominates the current query, as in the graph below. Here, assuming $q_1$ matches $q_2$, $q_2$ can use $q_1$ as an initialising value and ignore $c_1$ and $c_2$.

In cases where this dominance relationship is not present we can no longer use the optimisation. For example, given the graph below, $q_2$ cannot use $q_1$ as a initialising change in case $q_1$ does not occur (and therefore has no value). Previously $q_1$ would have had a value regardless of whether it actually fell on the execution path or not.

Possibly this problem could be mitigated by using a specialised constraint rather than element, and adding more general reasoning about relationships between queries. We discuss this further in Section 7.11.1.
<table>
<thead>
<tr>
<th></th>
<th>Failures (000s)</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decom</td>
<td>glob</td>
</tr>
<tr>
<td>bins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>11</td>
<td>168</td>
<td>168</td>
</tr>
<tr>
<td>12</td>
<td>2874</td>
<td>2377</td>
</tr>
<tr>
<td>golomb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>337</td>
<td>337</td>
</tr>
<tr>
<td>10</td>
<td>2720</td>
<td>2720</td>
</tr>
<tr>
<td>knap1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>45</td>
<td>110</td>
<td>109</td>
</tr>
<tr>
<td>55</td>
<td>387</td>
<td>191</td>
</tr>
<tr>
<td>knap2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>651</td>
<td>651</td>
</tr>
<tr>
<td>70</td>
<td>10358</td>
<td>10353</td>
</tr>
<tr>
<td>85</td>
<td>10859</td>
<td>10882</td>
</tr>
<tr>
<td>knap3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>45</td>
<td>110</td>
<td>109</td>
</tr>
<tr>
<td>55</td>
<td>388</td>
<td>191</td>
</tr>
<tr>
<td>pizza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1885</td>
<td>1046</td>
</tr>
<tr>
<td>4</td>
<td>3527</td>
<td>1724</td>
</tr>
<tr>
<td>proj1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>10</td>
<td>3350</td>
<td>3344</td>
</tr>
<tr>
<td>15</td>
<td>33727</td>
<td>16342</td>
</tr>
<tr>
<td>proj2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>10</td>
<td>2481</td>
<td>1251</td>
</tr>
<tr>
<td>15</td>
<td>8953</td>
<td>3511</td>
</tr>
<tr>
<td>proj3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>354</td>
<td>349</td>
</tr>
<tr>
<td>9</td>
<td>1807</td>
<td>1190</td>
</tr>
<tr>
<td>route</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>7</td>
<td>587</td>
<td>587</td>
</tr>
<tr>
<td>talent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5.9</td>
<td>5.9</td>
</tr>
<tr>
<td>8</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>9</td>
<td>405</td>
<td>352</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison between the previously used decomposition and an initial implementation of the reaching definitions global constraint, both with and without special cases. Each figure is the average for 10 instances of the given size. The number of instances that timed out (if any) is shown beside the time.
Comparison with special cases

Comparing `decom+` and `glob+` (also in Table 7.1), we can see that usually once again the propagation strength is the same. For `proj3` and `talent glob+` is weaker, probably due to the weaknesses in our adjusted special case translations discussed in Section 7.5.

For some benchmarks (`knap1`, `knap3` and `pp1`) adding special cases combats the problem of excessive element propagation, because the long chain(s) of matching queries causing the performance problem originate from `sum` or `max/min` calculations, and so the problematic element constraints are no longer required when using special cases.

However, there is a separate problem when using the `max/min` special case, which is especially visible for the `talent` benchmark (it also affects `pizza`, but the impact is hard to see because of the large number of timeouts). Previously when using `max/min` we no longer needed the intermediate queries inside the comparisons used to decide whether or not to update the result variable. However, we now need these queries to constrain the execution graph, as the comparisons define the conditions of `if` statements.

Actually the path through those statements has no effect on the rest of our model, but we cannot currently detect that. This means the new model contains extra essentially useless queries, slowing the solver down considerably. This problem should be solvable using graph simplification techniques, possibly using an algorithm similar to our graph compression algorithm (7.3), but we have not investigated this possibility in detail.

In summary, the global constraint successfully achieves the desired propagation strength, but has some performance problems unrelated to the new propagators which would benefit from further investigation.

7.7 Explanations

In order to use our new propagators in a lazy clause solver, we need to provide an explanation, in the form of a clause, for each propagation. We describe here the explanations used by the `query` propagator and the `reachOrNotMatch` propagator. Note that the `change` propagator performs no propagation directly; its task is to register disjunctions with the `reachOrNotMatch` propagator.

7.7.1 Explaining `query` propagation

The `basicquery` algorithm discovers cases where a change `c` cannot reach the query `q` because no path exists from `c` to `q` without passing through an overwriting change. To explain this propagation, we need to select a set of overwriting change nodes `B`, and explain both the lack of a path from `c` to `q` using this set as blocking nodes, and the fact that each change in `B` must overwrite `c` for `q`.

For effective explanations, it is important that the set `B` is not larger than necessary. That is, if a change matches `q` but plays no part in blocking `c` from reaching `q`, then it should not be included in `B`. To achieve this we begin with
a set $B'$ giving all possible blockers (changes known to match $q$), and while computing the lack of path explanation, we also reduce $B'$ to only the used blockers $B$.

To explain the lack of a path, we first find all nodes which could occur between $c$ and $q$ in the initial execution graph. Call this set $R'$. We then find the subset $R \subset R'$ of nodes which are still reachable backwards from $q$ given edge deletions and without passing through any node in $B'$. We can then explain the lack of a path by stating that all edges leading backwards from currently reachable non-blocking nodes ($R \setminus B'$) to nodes which originally may have lead to $q$ but are no longer reachable ($R' \setminus R$), are excluded. That is,

$$\text{nopath}(c,q) = \bigwedge_{(f,t,\text{cond}) \in E,\ t \in R \setminus B',\ f \in R' \setminus R} \lnot \text{cond}$$

Our set of blockers $B$ should include only changes which prevented nodes which were originally on a path between $c$ and $q$ from being currently reachable. So we compute $B = \{b \in B' \cap R \mid \exists (f,b,\text{cond}) \in E,\ f \in R' \setminus R\}$. The explanation for query propagation is then:

$$\left(\text{nopath}(c,q) \land \bigwedge_{b \in B,\ n \in \{1,2\}} (\llbracket m_{nq} = \text{val}_n \rrbracket \land \llbracket m_{nb} = \text{val}_n \rrbracket)\right) \rightarrow \llbracket \text{rdef}_q \neq c \rrbracket$$

The second part of the explanation states that all changes in $B$ match $q$. Note that we can only detect that a change and query must match if all relevant match variables are fixed. Unfortunately it is necessary to include the specific values taken by these variables ($\text{val}_1$ and $\text{val}_2$) in the explanation.

### 7.7.2 Explaining reachOrNotMatch propagation

When explaining reachOrNotMatch propagation, we first need to explain the reason behind a pair's registration, and then the reason for a specific propagation (basically that all other disjuncts are false). As there is a lot of overlap between different reachOrNotMatch propagation explanations, we begin by defining the potential components. We then show how the explanations are composed from these components, before detailing the specific literals used for the more complex components.

**Components of reachOrNotMatch explanations**

The following table lists the components of reachOrNotMatch explanations. Note that if $\text{rdef}_q$ is optional, $\llbracket \text{rdef}_q \neq c \rrbracket$ is not the negation of $\llbracket \text{rdef}_q = c \rrbracket$.  

---

170
Assuming if both pairs of relevant match variables are fixed. We can only detect (and explain) matchingness if they both occur).

Similarly, we can only enforce non-matchingness if three of the relevant match variables are fixed (one pair to the same value). We enforce non-matchingness based directly on the conditions used to detect the opportunity for propagation.

### Explanations by pseudo-code line number

The following table lists the explanation for each `reachOrNotMatch` propagation by line number in the pseudo-code in Figure 7.7. These explanations are all based directly on the conditions used to detect the opportunity for propagation.

<table>
<thead>
<tr>
<th>#</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td><code>regQ(c,q) ∧ match(c,q) → [rdef_q = c]</code></td>
</tr>
<tr>
<td>6</td>
<td><code>regC(c,q) ∧ match(c,q) ∧ [hap_c] → [rdef_q = c]</code></td>
</tr>
<tr>
<td>8</td>
<td><code>regCQ(c,q) ∧ match(c,q) ∧ [hap_c] → [rdef_q = c]</code></td>
</tr>
<tr>
<td>11</td>
<td><code>regC(c,q) ∧ match(c,q) ∧ [rdef_q ≠ c] → ¬[hap_c]</code></td>
</tr>
<tr>
<td>14</td>
<td><code>regCQ(c,q) ∧ match(c,q) ∧ [rdef_q ≠ c] ∧ [hap_c] → ¬[hap_q]</code></td>
</tr>
<tr>
<td>16</td>
<td><code>regC(c,q) ∧ match(c,q) ∧ [rdef_q ≠ c] ∧ [hap_q] → ¬[hap_c]</code></td>
</tr>
<tr>
<td>22</td>
<td><code>regQ(c,q) ∧ [hap_q] ∧ [rdef_q ≠ c] ∧ partmatch(c,q) → nomatch(c,q)</code></td>
</tr>
<tr>
<td>24</td>
<td><code>regC(c,q) ∧ [hap_c] ∧ [rdef_q ≠ c] ∧ partmatch(c,q) → nomatch(c,q)</code></td>
</tr>
<tr>
<td>26</td>
<td><code>regCQ(c,q) ∧ [hap_c] ∧ [hap_q] ∧ [rdef_q ≠ c] ∧ partmatch(c,q) → nomatch(c,q)</code></td>
</tr>
</tbody>
</table>

### Matchingness explanations

We now define the `match(c,q)`, `partmatch(c,q)` and `nomatch(c,q)` components used in the above explanations. We can only detect (and explain) matchingness if both pairs of relevant match variables are fixed.

Assuming $D(m_{1c}) = D(m_{1q}) = \{v_1\}$ and $D(m_{2c}) = D(m_{2q}) = \{v_2\}$:

$\text{match}(c,q) = [m_{1c} = v_1] ∧ [m_{1q} = v_1] ∧ [m_{2c} = v_2] ∧ [m_{2q} = v_2]$  

Similarly, we can only enforce non-matchingness if three of the relevant match variables are fixed (one pair to the same value). We enforce non-matchingness by removing a value from the domain of the remaining match variable.

Assuming $D(m_{1c}) = D(m_{1q}) = \{v_1\}$ and $D(m_{2c}) = \{v_2\}$:

$\text{partmatch}(c,q) = ([m_{1c} = v_1] ∧ [m_{1q} = v_1] ∧ [m_{2c} = v_2])$  

$\text{nomatch}(c,q) = ¬[m_{2q} = v_2]$
Registration explanations

reg_{CQ}(c, q)

When the full disjunction is deduced (by basicchange) for a pair \((c, q)\), the reason is that no change \(c'\) which could originally occur between \(c\) and \(q\) is now allowed to reach \(q\). Let \(R\) be the set of other changes originally reachable from \(c\). The explanation is then:

\[
reg_{CQ}(c, q) = \bigwedge_{c' \in R} [rdef_q \neq c']
\]

Note that for some changes in \(R\) the literal \([rdef_q \neq c]\) will be known true at the root node (for example changes which could only ever occur after \(q\)).

reg_{C}(c, q) and reg_{Q}(c, q)

The stronger disjunctions are discovered during fixed path traversal by the change and query propagators. In each case we simply explain that the path is fixed (all other original edges are eliminated), and that any potentially over-writing changes along this path do not in fact overwrite the change.

For a pair \((c, q)\) discovered by the change propagator, let \(P\) be the set of nodes included on the fixed path from \(c\) to \(q\), and let the function \(\text{next}(n)\) give the successor on this path for \(n \in P\). Then,

\[
reg_{C}(c, q) = \bigwedge_{(f, t, \text{cond}) \in E, \, f \in P \setminus \{q\}, \, t \neq \text{next}(f), \, \text{isChange}(n)} \neg [\text{cond}] \land \bigwedge_{n \in P \setminus \{c\}, \, \text{isChange}(n)} \text{notmatch}(c, n)
\]

where \(\text{notmatch}(c, n)\) is one of the following, letting \(l_{ij}\) and \(u_{ij}\) be the current lower and upper bound of match variable \(m_{ij}\):

- if \(l_{1c} > u_{1n}\): \([m_{1c} \geq l_{1c}] \land [m_{1n} \leq u_{1n}]\)
- if \(l_{1n} > u_{1c}\): \([m_{1n} \geq l_{1n}] \land [m_{1c} \leq u_{1c}]\)
- if \(l_{2c} > u_{2n}\): \([m_{2c} \geq l_{2c}] \land [m_{2n} \leq u_{2n}]\)
- if \(l_{2n} > u_{2c}\): \([m_{2n} \geq l_{2n}] \land [m_{2c} \leq u_{2c}]\)

For a pair \((c, q)\) discovered by the query propagator, let \(P\) be the set of nodes included on the fixed path back from \(q\) to \(c\), and let the function \(\text{prev}(n)\) give the successor on this path for \(n \in P\). Then,

\[
reg_{Q}(c, q) = \bigwedge_{(f, t, \text{cond}) \in E, \, t \in P \setminus \{c\}, \, f \neq \text{prev}(t), \, \text{isChange}(n)} \neg [\text{cond}] \land \bigwedge_{n \in P \setminus \{c\}, \, \text{isChange}(n)} [rdef_q \neq n]
\]
7.7.3 Evaluation of explanations

The experimental results in Table 7.2 suggest that our explanations are often weaker than those used in the decomposition. It is extremely difficult to pinpoint exactly why this is, as the learned clauses depend not only on the strength of explanations provided by propagators, but the order in which these propagators are run, and the order of literals within each explanation. Our propagators necessarily run in a different order from the decomposition because the implications are translated directly to SAT clauses which are always propagated before standard propagators.

However, the most likely cause is the fact that the decomposition makes direct use of the bounds literals for index variables. If the order of values is meaningful (as it is for index variables in the decomposition), these bounds literals greatly improve the strength of explanation as they summarise a group of related value literals. For example, if change $c_3$ happens and matches the query, the decomposition will exclude changes $c_1$ and $c_2$ by setting a single literal $[\text{index} \geq 3]$ (with a single explanation). Our query propagator will separately exclude these two changes, each with its own explanation. While we could adjust the basic query algorithm to exclude a set of changes with a single explanation, there is no guarantee that this set will have contiguous indices, let alone that they will fall at an extreme end of the index domain, so a single literal summarising the excluded values may not exist.

In future it would be very interesting to investigate the possibility of implementing an unordered values variable which could dynamically create summarising literals rather than using bounds literals (which are not meaningful if the values are not ordered). This would require changes to the variable interface to allow propagators to exclude arbitrary sets of values with a single explanation, and the selection of useful summarising literals would be far from trivial. However, there is the potential to greatly improve the effectiveness of learning when using this type of variable (which is very common in CP-style problems).

It is also possible that the fixed edge explanations are overly specific (including more edges than required), but an investigatory attempt to replace these with more general explanations did not yield an obvious improvement.

When using special cases, in general the explanations no longer seem significantly weaker (with the exception of talent). Possibly this is because setting bounds on index variables is most useful when there are many relevant changes for a query which may overwrite each other. The most common cause of such a pattern in our benchmarks is a sum or max/min calculation.

Interestingly, glob+ seems to produce especially good explanations for proj3, even though it was weak for this benchmark without explanation.

The execution time results do not suggest significant overhead for computing explanations for the global constraint propagations. Benchmarks for whom glob is weak are unsurprisingly also slow. In most cases where the propagation strength is the same, the execution time is also reasonably close. The exceptions are proj3 and talent, but for these benchmarks glob already exhibited especially poor time performance when not using explanation.
<table>
<thead>
<tr>
<th></th>
<th>Failures (000s)</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decom glob decom+ glob+</td>
<td>decom glob decom+ glob+</td>
</tr>
<tr>
<td>bins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>282 284 8.2 8.1</td>
<td>2.5 3.5 0.04 0.05</td>
</tr>
<tr>
<td>11</td>
<td>685 755 8.9 9.0</td>
<td>7.4 10.5 0.04 0.06</td>
</tr>
<tr>
<td>12</td>
<td>11099 9403 198 198</td>
<td>120.6 148.8 0.52 0.93</td>
</tr>
<tr>
<td>golomb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>99 99 99 99</td>
<td>0.5 0.5 0.49 0.48</td>
</tr>
<tr>
<td>9</td>
<td>620 629 620 629</td>
<td>4.4 4.3 4.41 4.30</td>
</tr>
<tr>
<td>10</td>
<td>3836 3836 3836 3836</td>
<td>40.7 36.7 40.64 36.54</td>
</tr>
<tr>
<td>knap1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>80 192 25 25</td>
<td>11.8 65.0 0.11 0.12</td>
</tr>
<tr>
<td>55</td>
<td>175 396 68 68</td>
<td>43.8 216.1 0.28 0.33</td>
</tr>
<tr>
<td>65</td>
<td>310 534 194 194</td>
<td>134.6 300.0 0.85 0.97</td>
</tr>
<tr>
<td>knap2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>103 103 103 103</td>
<td>15.8 15.7 15.76 15.78</td>
</tr>
<tr>
<td>115</td>
<td>145 145 145 145</td>
<td>29.2 29.3 29.28 29.21</td>
</tr>
<tr>
<td>130</td>
<td>201 201 201 201</td>
<td>52.6 52.6 52.53 52.58</td>
</tr>
<tr>
<td>knap3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>25 51 6.4 6.4</td>
<td>2.4 8.4 0.05 0.05</td>
</tr>
<tr>
<td>45</td>
<td>80 192 25 25</td>
<td>11.8 65.0 0.11 0.12</td>
</tr>
<tr>
<td>55</td>
<td>175 397 68 68</td>
<td>43.9 216.1 0.28 0.33</td>
</tr>
<tr>
<td>pizza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3454 2013 4177 3491</td>
<td>70.9 83.3 72.22 115.06</td>
</tr>
<tr>
<td>4</td>
<td>8930 5628 11039 7174</td>
<td>208.4 233.5 208.46 245.03</td>
</tr>
<tr>
<td>proj1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>52 514 0.0 0.0</td>
<td>3.1 77.3 0.06 0.28</td>
</tr>
<tr>
<td>150</td>
<td>118 1059 0.0 0.0</td>
<td>27.4 300.0 0.09 0.69</td>
</tr>
<tr>
<td>200</td>
<td>156 882 0.0 0.0</td>
<td>40.9 300.0 0.13 1.48</td>
</tr>
<tr>
<td>proj2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6.4 32 0.1 0.1</td>
<td>0.10 0.6 0.03 0.04</td>
</tr>
<tr>
<td>15</td>
<td>760 7452 0.5 0.5</td>
<td>15.7 229.5 0.04 0.05</td>
</tr>
<tr>
<td>20</td>
<td>8396 9095 58 60</td>
<td>272.8 300.0 0.54 1.04</td>
</tr>
<tr>
<td>proj3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>92 63 27 2.8</td>
<td>1.8 2.3 0.18 0.09</td>
</tr>
<tr>
<td>11</td>
<td>358 260 82 9.5</td>
<td>7.7 12.6 0.52 0.22</td>
</tr>
<tr>
<td>12</td>
<td>1735 802 105 9.2</td>
<td>42.8 49.2 0.67 0.22</td>
</tr>
<tr>
<td>route</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>34 34 34 34</td>
<td>0.3 0.4 0.32 0.37</td>
</tr>
<tr>
<td>6</td>
<td>207 207 207 207</td>
<td>2.9 3.3 2.91 3.34</td>
</tr>
<tr>
<td>7</td>
<td>1014 1014 1014 1014</td>
<td>36.3 40.5 36.30 40.49</td>
</tr>
<tr>
<td>talent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>49 49 7.4 17</td>
<td>1.0 3.4 0.09 0.83</td>
</tr>
<tr>
<td>8</td>
<td>387 394 24 74</td>
<td>8.6 31.8 0.26 4.04</td>
</tr>
<tr>
<td>9</td>
<td>3262 2795 67 293</td>
<td>88.7 262.4 0.80 21.99</td>
</tr>
</tbody>
</table>

Table 7.2: Comparison between the decomposition and initial global constraint, using explanation. Each figure is the average for 10 instances of the given size. The number of timeouts (if any) is shown beside each time figure.
7.8 Extension: Filtering Change Values

The change propagator can be extended to reason not only about single queries the change must reach or not match, but about sets of matching queries which impose a constraint on the assigned value. Assume from change node $c$ it is not possible to reach an overwriting change or the end of the program without passing through one from a set of query nodes $Q$. If all queries in $Q$ definitely match $c$, then we can deduce that if $c$ happens it must be the reaching change for at least one query in $Q$. This allows us to reduce the domain of the value associated with $c$ so that $D(val_c) \subseteq \bigcup_{q \in Q} D(val_q)$.

7.8.1 Using single queries

In the first, simple case, any query $q$ encountered during fixed path traversal which is known to match our change $c$ can be used to update the domain of $val_c$ directly. As discussed previously, if $c$ occurs it must reach or not match $q$, so if $c$ and $q$ match then $c$ cannot possibly use a value inconsistent with $q$. We therefore insert at lines 22 and 46 of the change propagation function (Figure 7.6) a call to update the bounds of $val_c$ according to $val_q$. The explanation for the lower bound (using components defined previously and letting $lb = \min(D(val_q))$) is:

$$(reg_C(c, q) \land match(c, q) \land [val_q \geq lb] \rightarrow [val_c \geq lb]$$

Actually, queries not on the fixed path but which we have previously registered with reachOrNotMatch can be used to update the domain of $val_c$ as well, as long as the query is known to happen. We perform bounds propagation using these queries as well, with the following explanation (for a lower bound change).

$$(reg_Q(c, q) \land hap_q \land match(c, q) \land [val_q \geq lb] \rightarrow [val_c \geq lb]$$

7.8.2 Using sets of queries

As mentioned above, it is also possible to reason about the possible values for our change using a set of matching queries. Essentially, we are looking for a vertex cut which separates the change node $c$ from other possibly matching (and therefore overwriting) changes and the end-of-program node. To propagate, the cut must consist entirely of query nodes known to match $c$ which all agree on at least one value to exclude from $D(val_c)$. Note that there may be many different cut sets consisting of matching queries. These sets commonly overlap, and not all of them will produce propagation.

Example Consider the graph shown in Figure 7.8, and assume that all queries are known to match $c$. The cut set closest in the graph to $c$ is $A$, but this set does not support any propagation, despite the fact that individually, both $q_1$ and $q_2$ would exclude a value (5 and 0 respectively). If we stopped searching after discovering this set we would miss potential propagation provided by sets $B$, $C$, and $D$.

175
To simplify the problem of selecting a good cut set, we consider bounds propagation only, and search separately for a set of queries providing a new lower and upper bound. We perform this search immediately after the simple bounds propagation inserted at line 46 of the change propagation function (Figure 7.6). This is when the change propagator has stopped fixed edge traversal at a branch. We do not need to bother searching for sets of queries if fixed edge traversal has stopped at a possibly matching change (line 22), as any cut set would be a singleton dealt with already.

**Searching for a cut set**

Before searching for cut sets, we first compute the following two sets of queries.

\[
Q_u = \{ q | \text{match}(c, q) \land (c \notin D(\text{rdef}_q) \lor (\text{max}(D(\text{val}_q)) < \text{max}(D(\text{val}_c)))) \} \\
Q_l = \{ q | \text{match}(c, q) \land (c \notin D(\text{rdef}_q) \lor (\text{min}(D(\text{val}_q)) > \text{min}(D(\text{val}_c)))) \}
\]

These are the maximum sets of queries which could be used to adjust the upper and lower bound of \(\text{val}_c\). Note that if a query is known not to have \(c\) as its reaching change, then its value bound is irrelevant. Such a query may form part of our cut set, in which case it indicates a path which is actually invalid.
Also, any query we have already used for single query bounds propagation as described above will not be included in either of these sets as the change bound must now be as strong as that query bound. For efficiency we can also exclude any query which could never occur after $c$ in the execution graph.

We then attempt to update the bounds of $val_c$. For the upper bound, if $Q_u$ is not empty we perform a graph search to discover if a subset of $Q_u$ in fact blocks $c$ from reaching any possibly matching change or the end-of-program node.

- The search begins at $c$ and traverses forwards in a depth first manner.
- Upon reaching a node $n$, we check the following.
  - If $n$ is the end-of-program node or a change which may match $c$, then we stop immediately as no propagation is possible
  - If $n \in Q_u$, we add $q$ to an initially empty set $Q$.
  - Otherwise we continue beyond $n$ exploring its successors.

If the search terminates without encountering a forbidden node, then $Q$ is a cut set as required.

**Setting the new bound**

Given a cut set $Q$, we can update $val_c$ using the weakest bound from a query in this set, ignoring any query known not to have $c$ as its reaching change.

If any query $q$ in $Q$ is not allowed to take $c$ as its reaching change, we know that the path we explored from $c$ to $q$ is actually invalid, because if that path were taken $c$ would have to be the reaching change for $q$. Therefore when calculating the new bound we use for these queries a value which would wipe the domain of $val_c$ completely. If $Q$ is composed entirely of queries known not to have $c$ as the reaching change, we will set this strong bound, which will force $val_c$ to be absent and therefore $hap_c$ to be false. This is the desired behaviour, as we have determined that if $c$ happens then it must reach a query from $Q$, but we already know it does not reach any of them.

Let $S$ be the set of all nodes seen in the search, $Q_r = \{ q \in Q \mid c \in D(rdef_q) \}$, and $U = \{ \min(D(val_c)) - 1 \} \cup \{ \max(D(val_q)) \mid q \in Q_r \}$. Then the new upper bound for $c$ is $ub = \max(U)$.

We set this new bound with the following explanation (using $match$ and $nomatch$ defined earlier).

\[
\left( \text{boundingSet}(Q, c) \land \bigwedge_{(f,t,cond) \in E, f \in S \setminus Q, t \notin S} \neg \text{cond} \land \bigwedge_{c' \in S \setminus \{c\}, \text{isChange}(c')} \text{nomatch}(c, c') \right) \rightarrow \llbracket val_c \leq ub \rrbracket
\]
where

$$\text{boundingSet}(Q, c) = \bigwedge_{q \in Q} \text{match}(c, q) \land \bigwedge_{q \in Q_r} [\text{val}_q \leq ub] \land \bigwedge_{q \in Q \setminus Q_r} [rdef_q \neq c]$$

The explanation says that all queries in $Q$ match $c$ and either cannot be reached by $c$ or have the required bound, all edges leading from a seen node not in $Q$ to an unseen node are excluded, and no seen change matches $c$.

**Idempotence**

Our simple search algorithm is not guaranteed to find the best new bound, and is not idempotent. It would of course be possible to use a max flow / min cut algorithm to find the best bound, or to run the presented algorithm repeatedly until a fix-point is reached. However, the extra overhead involved is unlikely to pay off.

**Example** Consider again the graph in Figure 7.8. Our search algorithm will find set $B$ when searching for a lower bound (finding new bound 1), and set $C$ when searching for an upper bound (finding new bound 4). We do not find the set $D$ (which would provide a stronger lower bound of 2), but would do so if the algorithm were repeated (and will do if nothing has changed the next time the propagator is run).

**7.8.3 Experimental evaluation**

Here we compare the original implementation of the global constraint ($\text{glob}$) with a version extended to reason about change values ($\text{glob1}$), to judge the usefulness of this extension. As can be seen in Table 7.3, without special cases the new reasoning is beneficial for several benchmarks, both with and without explanations. However, once we add special cases the added reasoning seems to have practically no effect on propagation strength. This is presumably because special case calculations account for almost all opportunities to apply the added propagation.

Given the origin of our benchmarks (code defining optimisation problems, and mainly classical CP problems), it is not surprising that beating the special case translations is difficult. Sum and maximum/minimum calculations account for almost all complicated calculations in our benchmarks. For simple calculations, it is likely that when change $c$ is known not to be overwritten for query $q$ it is also known to be the reaching change for $q$, in which case the values element constraint will already update the change value domain.

In future it would be interesting to evaluate the usefulness of this extension using benchmarks originating from software engineering applications (e.g. test-case generation or bounded model checking), as the code may then include different types of calculations (for which we do not have a special case translation).

Since the extension appears to be useful when not using special cases, we have decided to keep it.
<table>
<thead>
<tr>
<th></th>
<th>No explanation</th>
<th></th>
<th>With explanation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (secs)</td>
<td>Fails</td>
<td></td>
<td>Time (secs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>glob</td>
<td>glob1</td>
<td>glob</td>
</tr>
<tr>
<td></td>
<td></td>
<td>glob1</td>
<td>glob1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>glob1+</td>
<td>glob1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>glob1+</td>
<td>glob1</td>
<td></td>
</tr>
<tr>
<td>bins</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2.9</td>
<td>3.3</td>
<td>100% 100%</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>8.7</td>
<td>9.9</td>
<td>100% 100%</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>135.9</td>
<td>146.2</td>
<td>94% 100%</td>
<td>12</td>
</tr>
<tr>
<td>golomb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.5</td>
<td>0.5</td>
<td>100% 100%</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>5.6</td>
<td>5.5</td>
<td>100% 100%</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>63.4</td>
<td>62.6</td>
<td>100% 100%</td>
<td>10</td>
</tr>
<tr>
<td>knap1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>10.7</td>
<td>2.9</td>
<td>13% 100%</td>
<td>45</td>
</tr>
<tr>
<td>45</td>
<td>87.1</td>
<td>16.8</td>
<td>9% 100%</td>
<td>55</td>
</tr>
<tr>
<td>55</td>
<td>202.0</td>
<td>80.5</td>
<td>16% 100%</td>
<td>65</td>
</tr>
<tr>
<td>knap2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>8.0</td>
<td>7.8</td>
<td>100% 100%</td>
<td>100</td>
</tr>
<tr>
<td>70</td>
<td>94.7</td>
<td>93.9</td>
<td>102% 101%</td>
<td>115</td>
</tr>
<tr>
<td>85</td>
<td>137.2</td>
<td>136.1</td>
<td>102% 101%</td>
<td>130</td>
</tr>
<tr>
<td>knap3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>10.6</td>
<td>2.9</td>
<td>13% 100%</td>
<td>35</td>
</tr>
<tr>
<td>45</td>
<td>87.2</td>
<td>16.8</td>
<td>9% 100%</td>
<td>45</td>
</tr>
<tr>
<td>55</td>
<td>202.0</td>
<td>80.2</td>
<td>16% 100%</td>
<td>55</td>
</tr>
<tr>
<td>pizza</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>122.7</td>
<td>123.0</td>
<td>90% 90%</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>270.8</td>
<td>275.7</td>
<td>91% 92%</td>
<td>4</td>
</tr>
<tr>
<td>proj1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.0</td>
<td>0.0</td>
<td>42% 100%</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>54.6</td>
<td>12.6</td>
<td>19% 100%</td>
<td>150</td>
</tr>
<tr>
<td>15</td>
<td>300.0</td>
<td>210.0</td>
<td>68% 100%</td>
<td>200</td>
</tr>
<tr>
<td>proj2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.0</td>
<td>0.0</td>
<td>87% 100%</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>102.2</td>
<td>53.8</td>
<td>49% 100%</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>300.0</td>
<td>300.0</td>
<td>87% 103%</td>
<td>20</td>
</tr>
<tr>
<td>proj3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.7</td>
<td>1.9</td>
<td>100% 100%</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>46.5</td>
<td>50.3</td>
<td>100% 100%</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>166.0</td>
<td>171.9</td>
<td>94% 101%</td>
<td>12</td>
</tr>
<tr>
<td>route</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.7</td>
<td>0.8</td>
<td>100% 100%</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7.4</td>
<td>7.7</td>
<td>100% 100%</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>53.5</td>
<td>55.7</td>
<td>100% 100%</td>
<td>7</td>
</tr>
<tr>
<td>talent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2.9</td>
<td>3.1</td>
<td>100% 100%</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
<td>27.1</td>
<td>100% 100%</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>241.5</td>
<td>247.9</td>
<td>98% 100%</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 7.3: Comparing the initial global implementation (glob) with an extended version (glob1). We show (both without and with explanation) the average execution time and number of timeouts (if any) for glob and glob1, then fails for glob1 as a percentage of fails for glob, and fails for glob1+ as a percentage of fails for glob+. Note instance sizes are different with explanation.
7.9 Extension: Eliminating Edges

With a minor adjustment to the basicquery algorithm, we can occasionally detect infeasible edges in the constraint graph. Recall that basicquery performs a backwards search of the execution graph, looking for changes which may reach q without being overwritten. The query propagator calls basicquery after finishing its fixed edge traversal, whether it stopped because of a possibly reaching change (line 23 of Figure 7.5), or because of a branch (line 46).

Our extension of basicquery is called branchquery, and we call this extended version only when fixed edge traversal has stopped at a branch node b. Furthermore, we require that either q is known to happen, or the path between q and b is fixed in both directions. In this case we know the following:

- If b happens then q must happen (because we know q happens or because all paths forward from b lead to q).
- The reaching change for q must occur before b (because fixed path traversal does not go past a possibly reaching change).
- There are multiple edges leading in to b (or fixed path traversal would not have stopped).

The branchquery algorithm performs a search backwards from b as usual, but after visiting each predecessor of b it checks if the exploration of that branch discovered any possibly reaching changes (or joined a branch already explored via another edge). If not, then this edge can be removed. If the edge were used, then b would be on the execution path, which means q must happen, and yet there would be no reaching change for q.

7.9.1 Explanation

There are two parts to the explanation used when deleting edge (p, b, cond).

Firstly, we need to explain that if execution reaches b, then it will continue to q without passing any change which could reach q. Let P be the set of nodes included on the fixed path back from q to b. If q is known to happen, then (letting prev(n) give the successor on this path for n ∈ P) we define:

\[
\text{nothingAfter}(b) = [\text{hap}_q] \land \bigwedge_{(f,t,cond) \in E, t \in P \setminus \{b\}, f \neq \text{prev}(t)} \lnot [\text{cond}] \land \bigwedge_{n \in P, \text{isChange}(n)} [\text{rdef}_q \neq n]
\]

Otherwise, letting succ(n) give the successor for n on the fixed path from b to q, we use:

\[
\text{nothingAfter}(b) = \bigwedge_{(f,t,cond) \in E, f \in P \setminus \{q\}, t \neq \text{succ}(f)} \lnot [\text{cond}] \land \bigwedge_{n \in P, \text{isChange}(n)} [\text{rdef}_q \neq n]
\]
Secondly, we explain that no change in $D(rdef_q)$ can reach $p$ without being overwritten. For this we use a very similar technique as for explaining basicquery propagation.

We first find three sets of changes. Let $C$ be the changes in $D(rdef_q)$ (the reaching changes), $N$ be the changes not in $D(rdef_q)$ (the non-reaching changes), and $B'$ be the changes which definitely match $q$ but are not allowed to be the reaching change (the blocking changes). Also let $R'$ bet the set of all nodes which could occur between a node in $C$ and $p$ in the initial execution graph.

We next find the subset $R \subset R'$ of nodes which are still reachable backwards from $p$ given edge deletions and without passing through any node in $B'$. We can then explain the lack of a path from any node in $C$ to $p$ by stating that all edges leading backwards from currently reachable non-blocking nodes to nodes originally between $C$ and $p$ but no longer reachable, are excluded. That is,

$$\text{nopath}(C, p) = \bigwedge_{(f, t, \text{cond}) \in E, t \in R \setminus B, f \in R' \setminus R} \neg \llbracket \text{cond} \rrbracket$$

Next we reduce our set of blockers to only those which actually limited the search, letting $B = \{n \in B' \cap R \mid \exists (f, n, \text{cond}) \in E, f \in R' \setminus R\}$. We can then define:

$$\text{nothingBefore}(p) = \bigwedge_{c \in N} \llbracket rdef_q \neq c \rrbracket \land \text{nopath}(C, p) \land \bigwedge_{c \in B} \text{match}(c, q)$$

That is, only changes in $C$ (not in $N$) are allowed to reach $q$, and none of those can reach $p$ without going through a change in $B$, all of which definitely match $q$ (note also $B \subseteq N$). The complete explanation for deleting edge $(p, b, \text{cond})$ is then:

$$(\text{nothingAfter}(b) \land \text{nothingBefore}(p)) \rightarrow \neg \llbracket \text{cond} \rrbracket$$

### 7.9.2 Evaluation

We show experimental results for this extension to query propagation in the next section where we revisit loop untangling. Due to the typical shape of the execution graph, the extra propagation performed by branchquery is very rarely applicable when not using loop untangling. When using loop untangling the execution graph is more complex and the branchquery algorithm can be beneficial. It would be straight forward and possibly advisable to use branchquery only if the execution graph contains a cycle, but since there is only a very small amount of extra overhead involved in running branchquery rather than simplequery we have not investigated this in detail.
7.10 Revisiting Loop Untangling

In this section we revisit loop untangling, first introducing an adjustment to the graph compression algorithm, and then discussing experimental results using our new reaching definitions global constraint.

7.10.1 Graph compression

The previously presented graph compression algorithm is correct when applied to an execution graph with an untangled loop, but the resulting compressed graph can be overly complicated. Recall that the purpose of this algorithm is to remove structure unrelated to the important nodes - a set of changes and queries to be handled by a single reaching definitions constraint. The algorithm is based on Tarjan’s strongly connected components algorithm, which performs a depth first traversal of the graph. For simplicity, our compression algorithm retains all nodes with an edge leading to an ancestor in this traversal. This means that even if there are no important nodes within a given SCC, we will still retain the basic structure of that SCC.

Example Consider the execution graph in Figure 7.9, which includes a simple untangled loop with three copies of the body, followed by a single important node $c$. Assuming the visit order shown in this figure, node 9 will be retained because it is important, nodes 5 and 7 because they have edges to ancestors (2 is an ancestor for 5, and 2 and 4 are both ancestors for 7), node 3 because it has paths to both 5 and 7, and node 1 because it has paths to nodes 3, 5 and 7. This results in a compressed graph with irrelevant structure.

Figure 7.9: An execution graph including a small untangled loop (left), a possible visit order for the compression algorithm with retained nodes shown as squares (middle), and the resulting compressed graph (right).
Obviously we would like to avoid this problem so that our propagators do not perform traversals of cyclic graphs unnecessarily. We would also like to take advantage of the imposed order for untangled iterations having the same label value. Recall that when more than one copy of the loop body is created for the same value of the label query, we eliminate symmetry by imposing a fixed order on these copies. When performing graph compression, if all important nodes in an SCC belong to iterations having the same label value, then we can greatly simplify the structure of the compressed graph as we know that all valid execution paths will visit the important nodes in this order.

We achieve both of these goals using a simple check for each SCC discovered by the compression algorithm. Upon discovering an SCC (on line 35 of Figure 7.3) we collect the important nodes within that SCC, and check if they have a known order (including the case where there are no or only a single important node). That is, we attempt to find an ordering for the important nodes so that each node \( i \) is known to dominate node \( i + 1 \). The dominance calculation takes into account imposed dominances for copies of loop bodies having the same label value. If we can find such an ordering, we replace the SCC with the structure shown in Figure 7.10.

![Figure 7.10: An execution graph (left) including an untangled loop with two iterations for label value \( a \) and one iteration for label value \( b \), and its simplified compressed graph (right). The two important nodes within the loop (\( c_1 \) and \( c_2 \)) both belong to iterations with label value \( a \). Therefore we can compute that \( c_1 \) dominates \( c_2 \), and simplify the compressed graph.](image-url)
Essentially we insert a new start and end node for the SCC, then add the important nodes in order, each having an edge to the next important node and to the end loop node. This simplification allows us to only retain cycles in our compressed execution graph if there are actually two or more important nodes which can reach each other.

### 7.10.2 Experimental results

Here we evaluate the performance of our new global constraint on untangled benchmarks. Using the global constraint we have been able to apply loop un-tangling to one extra benchmark from our suite (talent), as well as the two benchmarks used in Chapter 5 (pizza and route). The remaining benchmarks do not include any loops which make obvious targets for untangling.

We also investigate the impact of the second extension to our implementation, which adjusted the query propagator to detect and delete infeasible edges in the execution graph.

Table 7.4 shows the experimental results without special cases. For the pizza benchmark, untangling is clearly beneficial, both in terms of number of failures and execution time. For route however, untangling actually produces much worse performance. This is most likely due to the simplifications we made in the basic change algorithm which prevent propagation if a change and query are involved in a cycle in the execution graph. We can see that adding edge deletion does provide some extra propagation, but this has no significant effect on the execution time. Finally, for talent untangling is (highly) beneficial, and as with route edge deletion does appear to slightly improve propagation strength, but not execution time.

<table>
<thead>
<tr>
<th></th>
<th>Failures (000s)</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decom</td>
<td>glob1</td>
</tr>
<tr>
<td>pizza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>200.1</td>
</tr>
<tr>
<td>4</td>
<td>893.0</td>
<td>511.7</td>
</tr>
<tr>
<td>route</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.4</td>
<td>3.4</td>
</tr>
<tr>
<td>6</td>
<td>20.7</td>
<td>20.7</td>
</tr>
<tr>
<td>7</td>
<td>101.4</td>
<td>101.4</td>
</tr>
<tr>
<td>talent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4.9</td>
<td>5.0</td>
</tr>
<tr>
<td>8</td>
<td>38.7</td>
<td>39.4</td>
</tr>
<tr>
<td>9</td>
<td>326.2</td>
<td>273.6</td>
</tr>
</tbody>
</table>

Table 7.4: Experimental results comparing the decomposition with loops unwound (decom), the global constraint (including the first extension) with loops unwound (glob1), this same global constraint with loops untangled (~glob1), and the global constraint with both extensions enabled and loops untangled (~glob2). Each figure is the average over 10 instances of the given size, with explanation enabled.
With special cases (Table 7.5) the situation is somewhat different. Untangling is still beneficial for pizza, and it is now also clearly worthwhile for route; while untangling apparently introduces substantial overhead, the improvement in number of failures offsets this as the instance size increases. For talent, untangling continues to improve performance compared with using the global without untangling, but can no longer match the decomposition (with special cases). This is probably due to the same special case weaknesses which cause glob+ to perform worse than decom+ for this benchmark.

Also in contrast to the results without special cases, when using special cases edge deletion reduces the number of failures for all three benchmarks, and produces a corresponding reduction in execution time for pizza and route.

Given these results, the edge deletion extension seems useful, and untangling continues to appear promising as a technique for modelling code involving complex loops. For further evaluation, while it would be fairly straightforward to construct more native interface examples which would benefit from loop untangling, it would be more interesting to explore whether or not the technique can be usefully applied in other applications, such as bounded verification. For any further investigation, automatic detection of promising loops and label queries would be highly advantageous.

<table>
<thead>
<tr>
<th></th>
<th>Failures (000s)</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>decom+</td>
<td>glob1+</td>
</tr>
<tr>
<td>pizza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>417.7</td>
<td>337.4</td>
</tr>
<tr>
<td>4</td>
<td>1103.9</td>
<td>690.1</td>
</tr>
<tr>
<td>route</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3.4</td>
<td>3.4</td>
</tr>
<tr>
<td>6</td>
<td>20.7</td>
<td>20.7</td>
</tr>
<tr>
<td>7</td>
<td>101.4</td>
<td>101.4</td>
</tr>
<tr>
<td>talent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.7</td>
<td>1.7</td>
</tr>
<tr>
<td>8</td>
<td>2.4</td>
<td>7.4</td>
</tr>
<tr>
<td>9</td>
<td>6.7</td>
<td>29.3</td>
</tr>
</tbody>
</table>

Table 7.5: Experimental results with special cases enabled, comparing the decomposition with loops unwound (decom+), the global constraint (including the first extension) with loops unwound (glob1+), this same global constraint with loops untangled (~glob1+), and the global constraint with both extensions enabled and loops untangled (~glob2+). Each figure is the average over 10 instances of the given size, with explanation enabled.

### 7.11 Conclusion

This chapter introduces the reaching definitions global constraint, and describes a first attempt at implementing this constraint in the lazy clause generation solver Chuffed. The implementation involved adding optional variables as well as several new propagators to the solver. The new propagators successfully
retain the propagation strength desired, and we also discuss two extensions able
to improve on this in some cases. However, there are various performance and
explanation problems which should be investigated further.

We conclude with a brief discussion of a few promising avenues we could
explore for a more powerful reaching definitions implementation.

7.11.1 Query-query reasoning

We have not yet investigated reasoning about the relationships between queries.
If two queries match and no matching changes can occur between them, their
value domains can be intersected. It may be possible to extend the change value
filtering algorithm to achieve this.

Also, in theory our basic query algorithm could stop upon reaching a match-
ing query, and use the index variable associated with that query to determine
which changes would be seen beyond that point. However in this case propaga-
tion order for query propagators would become very important.

7.11.2 Equality reasoning

Much of our reasoning involves equalities (and disequalities), but currently we
cannot detect equality between two variables until both are fixed, and we can-
ot enforce disequality until one is fixed. Since we both deduce and use equali-
ties/disequalities, we could be losing valuable information which could be shared
to produce further propagation. For example, if \( \text{val}_q \) and \( \text{val}_c \) are used as match
variables, we may be able to deduce that \( \text{val}_q \neq \text{val}_c \) and conclude that \( \text{rdef}_q \neq c \).

We also have no literals to represent equalities, so the solver cannot learn
about these directly. It is very weak to explain equality in terms of exact values,
because as soon as one variable takes a different value the clause is irrelevant.

It should be possible to implement a global equalities propagator, which
would perform a function similar to the theory of equality in SMT, and use a
similar union-find algorithm to keep track of known equalities and reason tranc-
sitively. Basically, other propagators could register an equality or disequality
for a pair of variables, or query a pair of variables to see what is known about
their equality status. The equality propagator should probably create literals
lazily as required, and must ensure that these are consistent.

7.11.3 Dominance reasoning

It is also possible to achieve stronger propagation by reasoning about dominance.
When only one change \( c \) can reach query \( q \), or when \( c \) dominates all other
possibly reaching changes for \( q \), we can deduce a new dominance in the execution
graph (\( c \) dominates \( q \)). Then, having discovered a dominance relationship \( a \)
dominate \( b \), we can enforce the property \( \text{hap}_b \rightarrow \text{hap}_a \).

Ideally, the maintenance of dominance information should be integrated into
the path constraint applied to the execution graph, possibly using DomReachability
rather than a circuit-based constraint, or some hybrid of the two.
Chapter 8

Conclusion

This thesis primarily concerns the usability of constraint solving technology for software developers attempting to incorporate optimisation functionality in a software application. However, due to the nature of the proposed approach for improving usability, much of the included research is also relevant to software engineering applications of constraint solving. This final section summarises the contributions made in the thesis, noting explicitly where wider applications are possible.

The first contribution is a proposal for a new style of interface to a constraint solver, which we call a native programming language interface. The idea is to allow optimisation (or constraint satisfaction) problems to be defined using native code in a general purpose programming language. The programmer defines the problem through code which combines individual decisions to create a candidate solution (using application data types), and code which evaluates a candidate solution. A library method can then be used to retrieve an optimal solution in the host representation. This is a general approach applicable to many different combinations of host language (whether declarative or procedural) and constraint solving technology (e.g. CP, MIP, SMT). The various advantages to a native programming language interface are discussed in Chapter 2.

The next contribution is a proof-of-concept implementation of a native Java interface to a CP solver. We show in Chapter 3 how the desired functionality can be achieved using existing techniques to convert the code provided by the programmer into an equivalent constraint model.

The remainder of the thesis investigates techniques to improve the translation of procedural code to constraints. These improvements are also applicable for software engineering applications of symbolic execution, such as automated testing or bounded program verification. Three main ideas are explored, as outlined below.

Firstly, a new query-based technique is presented to improve the handling of destructive assignments with aliasing between variables. Rather than creating a solver representation of the state at each execution step and constraining variable references to be consistent with the current state, we directly constrain
the variable references to be consistent with the relevant assignments. For our application this technique can produce much better models, in some cases as good as a simple hand-written model for the same problem.

Secondly we consider (for and while) loops, and introduce a novel approach for creating copies of the loop body. In contrast with the standard technique of unwinding loops, loop untangling involves labelling copies of the body based on the value of a key expression within the loop. This means that in each copy of the body the value of this key expression is known. Although we may need more copies of the loop body, and we can no longer assume a known order of execution for these copies, with a good choice of label query we can greatly simplify the constraints required for each copy. In some cases this can significantly improve the resulting model.

In order to support loop untangling in general, we need a path constraint to ensure that execution follows a proper path through the program. Chapter 6 investigates how the circuit propagator (which can also be used for a path) can be extended to handle the case where not all nodes are visited (as is the case for our execution paths), and explores how both versions of the propagator are best extended to explain themselves for use in a lazy clause generation constraint solver. This work is not specific to loop untangling, but relevant for any use of the path/circuit constraint.

The final contribution in the area of symbolic execution is a new global constraint reaching definitions, to be used to ensure that the value retrieved from a variable corresponds correctly to the reaching assignment. Chapter 7 describes a first attempt at implementing this constraint, which is required for general use of loop untangling, but also applicable when unwinding loops. The presented implementation successfully replicates the reasoning provided by the previously used decomposition without assuming a known order for assignments, and in some cases achieves stronger propagation. Several avenues for further research to improve this initial implementation are suggested in Section 7.11.

As part of the implementation of the reaching definitions constraint we also discuss optional solver variables and explain how a native implementation can be achieved in a lazy clause solver. This discussion has relevance beyond our specific application, as optional variables are useful for modelling a wide range of different constraint problems.
Bibliography


193


Author/s:
Francis, Kathryn Glenn

Title:
Natural optimisation modelling for software developers

Date:
2016

Persistent Link:
http://hdl.handle.net/11343/128081

File Description:
Natural Optimisation Modelling for Software Developers

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.