MODEL SELECTION AND TESTING FOR AN AUTOMATED CONSTRAINT MODELLING TOOLCHAIN

Bilal Syed Hussain

**A Thesis Submitted for the Degree of PhD
at the
University of St Andrews**

2017

Model Selection and Testing for an Automated Constraint Modelling Toolchain

THESIS

submitted to the

UNIVERSITY OF ST ANDREWS

for the degree of

DOCTOR OF PHILOSOPHY

by

Bilal Syed Hussain
bilalshussain@gmail.com

University of
St Andrews

August 2016

Abstract

Constraint Programming (CP) is a powerful technique for solving a variety of combinatorial problems. Automated modelling using a refinement based approach abstracts over modelling decisions in CP by allowing users to specify their problem in a high level specification language such as Essence. This refinement process produces many models resulting from different choices that can be selected, each with their own strengths. A parameterised specification represents a problem class where the parameters of the class define the instance of the class we wish to solve. Since each model has different performance characteristics the model chosen is crucial to be able to solve the instance effectively. This thesis presents a method to generate instances automatically for the purpose of choosing a subset of the available models that have superior performance across the instance space.

The second contribution of this thesis is a framework to automate the testing of a toolchain for automated modelling. This process includes a generator of test cases that covers all aspects of the Essence specification language. This process utilises our first contribution namely instance generation to generate parameterised specifications. This framework can detect errors such as inconsistencies in the model produced during the refinement process.

Once we have identified a specification that causes an error, this thesis presents our third contribution; a method for reducing the specification to a much simpler form, which still exhibits a similar error. Additionally this process can generate a set of complementary specifications including specifications that do *not* cause the error to help pinpoint the root cause.

Candidate's declaration

I, Bilal Syed Hussain, hereby certify that this thesis, which is approximately 25500 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student and as a candidate for the degree of Doctor of Philosophy in September, 2012; the higher study for which this is a record was carried out in the University of St Andrews between 2012 and 2016.

date ______________________ *signature of candidate* ______________________

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date ______________________ *signature of supervisor* ______________________

Permission for publication

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by the candidate and the supervisor regarding the publication of this thesis: No embargo on any electronic nor print copy.

date ______________________ *signature of candidate* ______________________

date ______________________ *signature of supervisor* ______________________

Contents

CHAPTER 1

Introduction

Constraint Programming (CP) is a powerful technique for solving a variety of combinatorial problems such as scheduling [BLN01], enumeration [Dis+12] and parsing [KPS10]. CP is declarative programming paradigm where *only* the relationships between a set of decision variables, which represent a choice that needs to be made in order to solve the problem, are stated. Each decision variable has a domain of allowed values associated with it, the relationships are then expressed in the form of constraint expressions, which place restrictions on the allowed combinations of assignments to the decision variables. While CP has the potential to solve these problems efficiently, users need significant experience to harness its power effectively. CP can be split into two separate stages, namely *modelling* which formalises the problem and *solving* which uses inference and search based on the constraints on the decision variables to solve the problem.

The modelling stage formalises the problem in question, specifying what decisions need to be made (the decision variables) and the restrictions upon the values these variables can take (the constraints). Optionally an objective can be specified, such as minimising the number of rooms needed in a timetable scheduling problem. Additionally the specification may represent a problem *class*, since it may depend on external input data such as the prerequisite modules in our example.

Solving a model involves finding an assignment of the decision variables such that they satisfy the constraints in the model. CP solvers such as Gecode [Gecode] and Minion [GJM06] support a limited number of different types of decision variables, including booleans, integers and matrixes of these types. For other constructs such as multi-sets or partitions, decisions need to be made as to how these constructs should be represented in terms of the elementary decision variables that a solver supports. Since the most efficient representation depends on the constraints utilised and may even depend on how external data is structured, it is non-trivial for a user to choose the most efficient representation to use. Additionally a user might not even know of the existence of a particular representation. This is known as the "modelling bottleneck" [Pug04] and impedes the wide adaptation of constraint programming.

There are many proposed solutions to this problem, ranging from machine learning to automatically discover which constraints are applicable [BS11] to refinement based approaches which starts with an abstract specification and refines it into a concrete model that a solver can support natively [Akg14] which is the focus of this thesis.

Resulting from abstract specification languages such as Essence [Fri+08] and Zinc [Mar+08] supporting many high level types such as partitions and multi-sets there are many possible refinements to consider each with their own trade offs. To further complicate the refinement process, it can be beneficial to refine the *same* decision variable with *different* representations in a single model [RVW06], which raises the number of refinement choices dramatically. Another complication is that a specification may be parameterised, hence the need to apply the refinement process to parameters as well.

To compare the performance of the refined models, we need a wide variety of instance data of varying difficultly. To show why the creation of these instances is non-trivial consider the Essence specification language, where decision variables can be arbitrarily nested (e.g. `set of mset of partition`) and can have any number of constraints to restrict the values of the parameters (such as constraining an integer parameter to be even). A feature that makes Essence more expressive is that the domain of allowed values that a parameter can take can *depend* on the value chosen for another parameter. This thesis automates the process of generating instance data for the purpose of automated model selection, with further applications to automatic test case generation.

Once we have generated individual instances we then focus on generating complete specifications with full coverage of the Essence language. The purpose of this is to automatically generate test cases for validating the toolchain described in Chapter 3. This testing process is able to detect, among other errors, inconsistencies in the refined models and invalid solutions. Notably this process can generate parameterised specification by utilising our instance generation methods. We then *reduce* a specification that causes an error to a much simpler specification that still exhibits the same error, which helps pinpoint the error. The reduction process also utilises the instance generation process to generate new instances for the reduced specification, which is used among other things to test if the error is caused by a particular value of the parameter.

1.1 Thesis Statements

The thesis defended in this dissertation is: Through the automatic generation of instance data from a high level specification we can select a subset of the refined constraint models that perform well across the instance space.

Through the automatic generation of high level specifications we can test a toolchain for automatic constraint modelling. Additionally we can reduce a specification, which may be parameterised, that causes an error to much simpler form that still exhibits the error.

1.2 Contributions

The main contributions of this thesis are: a way of generating instance data for high level specifications automatically for the purpose of model selection; a way of automatically generating test cases to test and validate a toolchain for constraint modelling and solving; and a way of *reducing* a test case to a much simpler form while still exhibiting the same error. In chronological order:

Solution Translation: The first contribution, that the rest of the contributions depend on, is a generic method to translate a low-level solution of the refined model back to the Essence level. This involves enhancing the refinement process to store the necessary information to convert back.

Instance Generation: The instance generation process works for any Essence specification, including specifications that place restrictions on the values that the parameters may take.

Generation of Dependent Parameters: A feature that makes Essence more expressive is that a parameter's domain's allowed values can *depend* on the value chosen for another parameter. We automate this by finding an ordering such that the dependencies are generated first.

Test Case Generation: A notable feature is that our generator can generate any Essence specification, including features such as matrix comprehensions and quantification over variables. Additionally the generation process can reconfigure itself automatically, guiding itself to *different* errors.

Generation of Parameterised Specifications: By utilising the instance generation method the test case generator can produce *parameterised* specifications. This allows testing of more parts of the toolchain automatically and allows the generator to finds bugs that depend on a particular *value* of a parameter of the instance.

Reducing Test Cases: This process reduces a test case to a much simpler form by applying a set of transformations to pinpoint the error. Additionally this process transforms the rules applied to produce the original refinement in such a way that they are still applicable to a candidate reduction.

Generalising Test Cases: To provide more insight into an error we can *generalise* a test case, to produce new specifications including specifications that show when the error does not occur.

1.3 Publications

In chronological order the publications that relate to this thesis are shown below:

[HMG13] is a preliminary paper showing a method that was developed to generate instances automatically for a small subset of the Essence specification language.

[Akg+13b] describes the toolchain for modelling and solving CP problems, developed at the University of St Andrews School of Computer Science including my solution translation contribution.

[Gen+14] forms the basis of Chapter 5 which extends the paper significantly. The paper showed various methods to generate discriminating instances automatically. I contributed the experiments and the implementation which I later completely redesigned for this thesis so that it could be repurposed for test case generation and test case reduction.

1.4 Thesis Structure

The thesis is structured in the following way:

We begin with Chapter 2 which starts by discussing constraint modelling, and the various proposals to automate it. It then expands the discussion to the process of generating instances and concludes with a discussion of test case generation and test case reduction.

Chapter 3 describes the *toolchain*, which is used for the automated modelling approach that is utilised throughout the thesis. Additionally this chapter presents our first contribution namely solution translation. This process transforms a solution to a refined model, written in the low level ESSENCE′ language back to ESSENCE hence abstracting over modelling decisions and refinements that were used on behalf of the user.

In the presence of a highly parameterised specification having many possible models resulting from the various combinations of possible refinements, Chapter 5 shows how to generate instance data that can be used to choose a subset of the models that have good performance over the given instance space. This generation method also has applications to model generation and testing.

Model generation is the focus of Chapter 6. It presents a way of testing and validating the toolchain without relying on any particular piece being correct. A notable feature is that the generation process covers the complete ESSENCE language including parameterised specifications. Additionally we discuss how combining the generation process with our other contributions, we guide the generator to more interesting specifications automatically.

Once a test case has been generated (or even given by a user) we a need a way of *reducing* it to a simpler form. Chapter 7 shows how this is achieved using a series of transformations. In addition it presents a process which enhances the refinement process so the that refinement of each reduced specification candidate will result in a similar error, if it exists, without having to refine all models. It also deals with the idea of generalisation which takes a reduced specification and provides a selection of complementary test cases to give insight into the root cause.

We conclude in Chapter 8, giving a summary of what was achieved in addition to possible future work.

Chapter 2

Related Work

In this chapter we describe the surrounding context as well as related work. We start with a discussion of the various approaches to automated modelling and model selection. We follow with a discussion of how instance data can be generated. We then expand this discussion to the generation of test cases. Finally we focus on how to *reduce* a test case, starting from the most general case, to more specific techniques that focus on a particular domain.

2.1 Constraint Modelling

Constraint modelling is the process of formalising a problem in question in a way that a constraint solver can understand. A comprises of a set of decision variables each with an associated domain of values and a set of constraints which place restrictions on the assignments to each decision variable. A solution to the model is an set of assignments to the decision variables such that all the constraints are satisfied. Optionally an expression to optimise can be specified converting a model into a optimisation problem.

CP Solvers such as Gecode [Gecode] and Minion [GJM06] only provide a limited number of decisions variables including boolean, integer and matrixes of these types. Other types such as relations have to be *modelled* in term of these elementary types. For non-expert users in particular, the most efficient representation may not be obvious since it will depend on the constraints utilised and the size of the instance data given. Additionally since there are many specialised constraints for particular problems, it can be difficult to know when a particular constraint is applicable. This problem, namely the formulation of an effective constraint model in a form that a constraint solver can understand is the 'modelling bottleneck' [Pug04], which hinders the adoption of constraint programming. There are many approaches to overcome this problem, ranging from refinement based to example driven, as described in the chapter.

2.2 Automating Modelling

We will start with a discussion of refinement based approaches. Our approach starts with a high level abstract specification (in Essence) which is then refined into a concrete model which a constraint solver can solve. This approach to automated modelling which we be will be utilising throughout the thesis will be described in Chapter 3.

2.2.1 Refinement Based Approaches

2.2.1.1 OPL

OPL [Van+99] is one of the earliest examples of abstracting over the internals of a constraint solver. It supports as the decision variables only integer and enumerated types, in addition to an aggregate type to group related data. It provides various constraints which act upon these variables and additionally allows the user to specify an objective expression to maximise/minimise. Unlike some later languages it does not provide any abstract domains.

ESRA [FPÅ04] is a language that can be compiled into OPL by utilising a series of rewrite rules. It introduced relation decision variables, increasing the expressiveness of the language. A single model is produced after refinement since there is only a single representation available for a relation.

2.2.1.2 F and Fiona

Fiona [Hni03] takes a specification written in the F language and refines it into a target language, one of which is OPL. The most notable feature of F is that it supports mathematical functions as decision variables, but does not allow nesting, that is placing functions inside another function decision variable. It supports multiple representations for functions depending on the restrictions placed upon the decision variables. As output it produces a single resulting model, applying heuristics to decide which representation to use during refinement.

2.2.1.3 Zinc and MiniZinc

MiniZinc [Net+07] is a widely supported "medium-level" modelling language. In addition to common features such as booleans and integers it also supports set of integers. MiniZinc interacts with constraint solvers by converting the MiniZinc model to a solver specific FlatZinc representation. Like OPL, MiniZinc models can be parametrised, during the conversion to FlatZinc the model is instantiated.

Zinc [Mar+08] is a high level modelling language. It provides more abstract domains then MiniZinc such as user defined records and provides user defined functions[1]. Since there are many different refinement choices at the level Zinc operates at, it could be used to compare the effectiveness of different representations. However the Zinc compiler currently only produces a single model. This is achieved using 'Type reductions' [KBS10] which convert high level types into a form that a solver can understand.

2.2.2 Machine Learning

Beldiceanu and Simonis present Model Seeker, a system that automatically constructs models from positive solutions [BS12]. This system utilises their Constraint Seeker system [BS11] that can return a ranked list of the most appropriate constraints given training inputs. The limitation of this approaches is that it currently does not use negative examples which could be used as a tiebreaker between similar models. Bessiere et al. similarly uses positive and negative examples as input, and uses a SAT Solver for the learning process [Bes+05], but has the limitation that it does not work effectively for global constraints over a large number of variables.

2.2.3 Case Based Reasoning

O'CASEY [Lit+03] in contrast to the refinement based approaches described previously uses case based reasoning. O'CASEY works by building knowledge in the form of a 'case' of when certain decisions were beneficial. When a new input is given, it tries to adapt the most similar previously used case to the new input. The experience gained stored in the cases contains the

[1] User defined functions were added to MiniZinc in a later version.

choice of search heuristics and propagators. Unlike some of the reformulation approaches it does not change the representation of the variables.

2.2.4 Reformulation

In a similar vein to refinement, reformulation aims to enhance the user's model and abstract away some of the difficulties of modelling. Examples include: common sub-expression elimination [Nig+14], automatic removal of symmetry [Mea+11] and finding global constraints automatically [BS11].

It can be beneficial to reformulate a CSP into alternative methodology such as propositional satisfiability (SAT) [SM14] and SMT[2] [Ans+13]. Similar to the refinement of an abstract specification where there are many possible refinements, there are many possible encoding to SAT, each with their own strengths. Ways of automating the selection process range from the use of heuristics to select a single model [Akg+13a] to model portfolios, which are described in the next section.

2.2.5 Algorithm Selection

Model selection is an example of the algorithm selection problem first described by Rice [Ric76]. This is the problem, given a set of instances and a set of algorithms, find a mapping which determines which algorithm to use on a particular instance. Since there is unlikely to be a single best algorithm for a problem [WM97], this led to the use of a *portfolio* of algorithms which can overcome this difficulty. Algorithm portfolios [GS97; HLH97] are comprised of a subset of the available algorithms. In the simplest case, a single algorithm is selected (by some metric) when a new instance is provided to be solved. In other variants multiple algorithms are executed on the same instance, in the hope that one of the algorithms would be suited for that particular instance.

There are two main kinds of portfolios: In the static variant once the algorithms for the portfolio have been selected, they are not changed, which is in contrast to dynamic portfolios where the algorithms selected can change making them possibly more adaptable to

[2] Satisfiability modulo theories (SMT) is an extension to SAT that is more expressive, where the goal is to determine if a formula is satisfiable with respect to some logical theories.

new instances. Since the algorithms do not change in a static portfolio, it is vital that the algorithms selected for the portfolio are diverse. For some problem classes diversity is more impor tant than performance when building a static portfolio. Hong et al. show that selecting randomly from a diverse set of algorithms outperforms selecting the algorithms with the best performances on their training data [HPB04].

There are many approaches to creating a diverse (static) portfolio [XHL10; SM07]. SATzilla [Xu+08] (a portfolio of SAT solvers) selects solvers which have won previous SAT competitions as the basis for their portfolio. Dynamic portfolios have the flexibility to refine the portfolio by adding and removing algorithms as necessary. The CLASS system generates new 'composite' heuristics for SAT problems which are competitive with current search heuristics [Fuk02].

Hurley et al.'s approach is similar to our refinement based approach in that it starts with a single model [Hur+14]. The main difference is that they use machine learning techniques such as linear regression to decide between different SAT encodings, as well as using CP directly. Similarly Stojadinović and Marić selects the most appropriate SAT encoding for a CSP based on the given instance [SM14].

A survey of techniques to solve the algorithm selection problem is given by Kotthoff with a focus on combinatorial search problems [Kot14].

2.3 Instance Generation

There have been many approaches to generating input data, usually with specific characteristics with use cases ranging from benchmarking to algorithm configuration. Many of the instance generation methods that we will discuss generate SAT formulae, which comprise variables, parentheses and operators (conjunction, disjunction, negation).

Ansótegui et al. show how to generate instances that have similar properties to those found in industrial SAT instances [ABL09; Ans+12]. The motivation being that while the generation of purely random SAT formulae [SML96] allows the difficulty to be controlled (by varying the ratio of clauses to the total number of variables in the formula), they do not have the characteristics of industrial SAT instances meaning that training on these instances does not generalise to the real instances that people want to solve. Ansótegui et al. overcomes this by utilising a geometric and power-law distribution to control the selection of variables in the formulae. A notable feature of their generation process is that they ensure there is always a problem hardness phase transition with regards to the parameters of their process. This is the point where the instances become much harder to solve, for SAT this is the point just before the formulae is over constrained hence trivially unsolvable. This means the range of SAT formulae produced range from very easy to very hard to solve. 'Morphing' has a similar goal: namely to add structure to random generated instances, making those instances closer to those that people want to solve [Gen+99].

In comparison to complete search methods such as CP or SAT, local search methods are incomplete. That is they may not find the optimal solution, even when there is one. Resulting from this it can be hard to differentiate between a problem that has no solution and a failure in the local search procedure. Achlioptas et al. overcome this by *only* generating satisfiable instances. This differs from previous generation methods, which generate instances and use complete search methods to discard the unsolvable instances [Ach+00].

Instance generation has also been used in model driven engineering where a 'model' represents the system in a form such as UML diagram and conforms to a 'metamodel'[3]. Since metamodels can have arbitrarily complex constraints placed on them, generating an instance 'model' manually is difficult and approaches to automate this have utilised SMT Solvers [WMP13].

Our instance generation method (Chapter 5) can generate instances for any Essence specification. In contrast to other generation approaches the instances that we generate are usually highly structured. For example, having the `given` statement `set (size 3) of function (injective) of int(1..5) --> int(1..10)` would mean that our instance generator would produces sets with three injective functions. Additionally Essence specifications can place arbitrary restrictions on the instances we generate including constraints between the individual parameters of the Essence specifications.

[3] For a metamodel of a programming language, a valid instance would be a program that can be complied.

2.4 Test Case Generation

There are two main forms of automated testing: black-box and white-box. Their difference being in whether we utilise internal information about the program under test. Our approach (Chapter 6) which utilises random sampling is most similar to black box testing which we will discuss in detail.

White-box testing approaches such as Symbolic Execution [Kin75; Ana+13] analyse the program under test, with use-cases such as a security analysis at the source level [CF10] and improving test coverage by generating new test cases [G+08].

In black-box testing, a generator produces inputs for a program without knowing the internal details. The majority of these methods utilise random sampling in some form. Property based testing approaches such as QuickCheck [CH11] requires the user to specify the invariants of the program that should hold. These methods then try to generate an input such that the invariant is falsified. If this happens then either the invariant is badly specified or an error has been found. Property based testing has various applications, Li et al. use it to test web based services by synthesising the test cases from a description in WSDL[4] detailing the input and output of a service [Li+14]. Our approach uses random sampling to generate Essence specifications but differs in that we dynamically reconfigure the process to find *different* errors, in addition to automatic test reduction.

In model-based testing, test cases can be generated from the specification (usually a variant of UML) [Cav+02]. Related approaches include using model checking [ABM98], utilising static analysis [BFG03] and converting a restricted form of natural language into UML then using the UML to generate test cases [SV11].

Fuzz testing tries generating a large number of random inputs to find crashes and flaws that can be exploited. An example would be programs such as a jpeg decoder in a web browser. Even giving invalid data should not cause a crash in a way that can be exploited maliciously. Fuzz testing works particularly well when an individual input can be tested efficiently. These approaches usually focus on a specific domain such as JavaScript [Ruda; Rudb] and C [Yan+11] both of which require no modification of the program under test.

[4] Web Services Description Language (WSDL) is a XML base description language specified by the W3C at `http://www.w3.org/TR/wsdl20/`.

Generating test cases can also be modelled as a CSP. Mossige et al. use it for stress testing robot deadlines [MGM14]. Lazaar et al. take a different approach to testing by presenting a framework that can verify reformulations and the addition of auxiliary constraints (e.g. for symmetry breaking), using the original specification as the oracle [LGL12]. Finally a survey of test case generation methods is given by Anand et al. [Ana+13].

Our test case generator (Chapter 6) covers all aspects of the Essence specification language. Additionally the chapter discusses how to validate the solutions produced by the toolchain described in Chapter 3. Our test case generator differs from the generators discussed above since Essence is a specification language for formalising combinatorial problems rather then a general purpose programming language. Resulting from this, our generator has to be able to generate unusual features that are allowed in Essence such as out of bounding indexing of matrixes and functions, non-contiguous matrixes indexed by arbitrary expressions and parameters whose domains depend on other parameters. Additionally our generator can produce parametrised specification along with instance data using our instance generation method.

2.5 Test Case Reduction

Automated testing is likely to produce huge test cases which may also contain multiple errors, thus making it harder to find the root cause. Ideally we would like a minimal test case that contains only a single error.

Different approaches to this problem have been proposed, ranging from very general algorithms to domain specific ones. Cleve and Zeller's 'delta debugging algorithm' can reduce a failing test case in the general case, but this generality comes at the price of efficiency since no domain knowledge is utilised [CZ00]. For programming languages, in particular, reduction is likely to lead to syntax errors which are unlikely to be useful. Examples of this include unbalanced brackets when reducing nested expressions, the removal of the semi-colon at the end of a line in C or the removal of the dot at the start of the body of a quantification in Essence.

To overcome this inefficiency more specialised versions of the reduction process have been developed. For the C programming language this was shown by Regehr et al. which uses its domain knowledge of the C programming language to avoid undefined behaviour during the reduction process [Reg+12] since if *any* undefined behaviour was performed that program is not well formed. In a similar vein Mozilla [Rudc] has a test case reducer that has an understanding of Javascript, which means the reducer does not generate invalid syntax during the reduction process.

Test case reduction has also been applied to other fields such as symbolic execution. Zhang et al. showed how reducing test cases (by removing parts that are redundant for the purpose of program coverage) allows more test cases to be run, hence leading to increased overall coverage of the program for a given total time [ZGA14].

Our test case reducer (Chapter 6) can be applied to any Essence specification that causes an error. Similar to other test case reducers, the reductions produced by our reducer are always a valid Essence specification. Our reduction process also works on the instance data for parametrised specification leading to specification with simpler dependencies. Additionally our test case reducer can be applied to Essence′ models when errors are found in Savile Row.

2.6 Solution Translation

Once a specification has been refined into a concrete constraint model, and solved using a constraint solver we have a solution in terms of that model. We ideally would like to present the solution in terms of the original specification to abstract over the modelling decisions that were made on behalf of the user. Our first contribution, solution translation achieves this by providing a generic method to translate a low-level solution of the refined model back to the original level of abstraction. This is of particular importance since the refined models can represent the same abstract domain in numerous ways. Our solution translation method converts these solutions into a uniform format in terms of the original specification.

Similar ideas have been applied in web browsers using a *source map*[5] which is usually used

[5] The specification is located at [LF11], Mozilla's implementation is located at `https://github.com/mozilla/source-map`

when a language is compiled to Javascript, or when Javascript is minimised[6]. The source map provide supplementary information such that a line in the complied code can be mapped back to the line of the original code that it was derived from. This is particularly useful if an error occurs and a stack trace is produced since it can help pinpoint where the error occurred in the original code. The idea of generating supplementary information to help support debugging has being used in many compilers such as LLVM and has similar benefits to source maps. More generally solution translation is an example of a program transformation that increases the level of abstraction. Some of these transformations *decompile* the resulting program (e.g. Java bytecode) back to the original language (Java in our example). Decompilation attempts to reconstruct the structure of the original program without access to the source code of the original program [War04]. Decompilation can be particularly useful if the source code of a program has been lost [EW04].

2.7 Summary

In this chapter we have described the various approaches to automating the modelling process. Following this we discuss approaches for generating instances ranging from purely random instances, to instances that must have a certain characteristic such as always being solvable. We then discussed methods for generating test data using two approaches namely white-box testing and black-box testing. We concluded with a discussion on how to *reduce* these test cases ranging from the most general techniques to domain specific applications.

[6] By removing whitespace, comments and by extracting common expressions to make the resulting Javascript file smaller. This usually results in the webpage loading faster.

Chapter 3

Architecture

This chapter discusses the *toolchain* that we will use throughout. We will start by firstly giving an overview of the Essence specification language. Following this we will work our way down the toolchain describing each tool in turn and its to the rest of the toolchain. Figure 3.1 shows how an Essence specification is refined, solved and finally translated back to a solution at the Essence level.

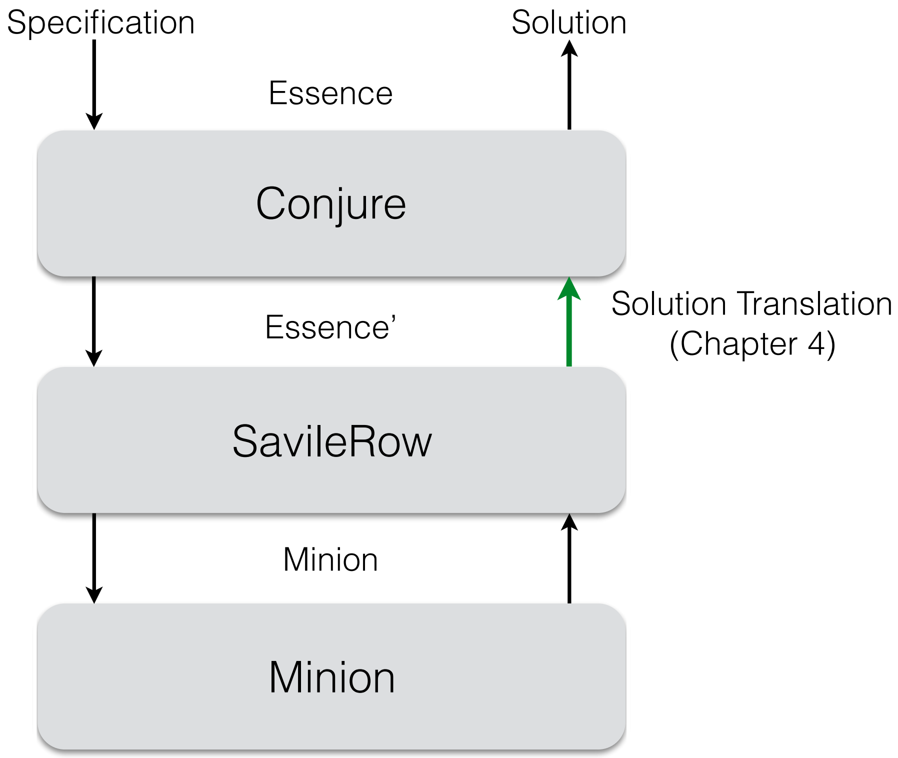


Figure 3.1: An overview of the toolchain.

Many parts of the toolchain already exist: Conjure by Akgün [Akg14], Savile Row by Nightingale et al. [Nig+14] and Minion by Gent et al. [GJM06].

Of our contributions, Instance Generation (Chapter 5) utilises the toolchain for two main purposes: solving a generated Essence specification derived from a given specification to generate Essence parameters; and evaluating the performance of these parameters on Essence′ models refined from the given specification. Test Generation (Chapter 6) and Reduction (Chapter 7) tests and validates the toolchain though the automatic generation of Essence specifications.

3.1 Essence

Essence is a high level specification language [Fri+08]; Its distinguishing feature is that it does not require the user to make modelling decisions for how variables should be represented. To achieve this Essence provides *abstract* domains for decision variables which include familiar mathematical constructs: multi-sets, relations, partitions and functions, all of which can be nested to an arbitrary depth.

Figure 3.2: An Essence specification

```
language Essence 1.3
given n : int(1..100)
given totalWeight : int(1..1000)
given weights, values : function (total) int(1..n) --> int(1..100)

find picked: set (maxSize n, minSize 1) of int(1..n)

maximising (sum i in picked . values(i))
such that
    (sum i in picked . weights(i)) <= totalWeight
```

An Essence specification comprises: a sequence of `find` statements which represent the combinatorial objects we wish to find; the constraints these objects must satisfy (`such that`); and optionally a sequence of `given` statements which parameterise the problem class and whose values define the instance. In addition an objective function (`minimising/maximising`) may be imposed and identifiers defined (`letting`).

Table 3.1 shows all the available domains and domain constructors in the Essence language. The Conjure system as described in the following section, as well as our solution translation contribution in Chapter 4, can handle all of these domains.

Domain	Kind
`bool`	concrete
`int`	concrete
`tuple` $(\tau_1, \ldots, \tau_n)$	abstract
`set of` τ	abstract
`mset of` τ	abstract
`function of` $\tau_1 \rightarrow \tau_2$	abstract
`relation of` $(\tau_1, \ldots, \tau_n)$	abstract
`partition from` τ	abstract
`enumerated`	abstract
`unnamed`	abstract

Table 3.1: The domains of Essence

A domain is *abstract* if modelling decisions need to be made to refine it into a form which is supported natively by the target language/tool. As an example the Essence′ language only supports integers, booleans and matrixes of these types. For other domains, modelling decisions need to be made and the domains refined. In addition, Essence provides operators including set union and function application which act on these abstract domains. Each of these operations themselves have to be refined in the same manner by making modelling decisions. Instance data at the Essence level needs to be refined specifically for *each* Essence′ model produced, since each of the models may use different representations for the `given` statements. The Conjure automated modelling system is one way of achieving these refinements automatically.

3.1.1 Unnamed Types

Solutions to a CSP have symmetries if we can apply a mapping to any of the solutions to produce a new solution to the CSP. The main benefit of removing symmetry is that we can reduce the amount of search needed to solve the CSP by imposing additional constraints [RVW06]. Unnamed types, by definition, are interchangeable and can only be tested for (in-)equality by the user. The advantage of an unnamed type as compared to an enumeration is that they do not impose additional symmetry which leads to improved model performance. We will use the social golfer's problem [Har] to illustrate how unnamed types could be employed. The goal of the problem is to find a schedule for g golfers for w weeks who play in groups of size s with the restriction that no two golfers play together more then once. Figure 3.3 presents one way of specifying the problem in Essence.

Figure 3.3: An Essence specification of the Social Golfers Problem

```
1  language Essence 1.3
2  given w, g, s : int(1..)
3  letting Golfers be new type of size g * s
4  find sched : set (size w) of
5                  partition (regular, numParts g, partSize s) from Golfers
6
7  such that
8      forAll week1 in sched . forAll week2 in sched, week1 != week2 .
9          forAll group1 in parts(week1) .
10             forAll group2 in parts(week2) .
11                 |group1 intersect group2| < 2
```

Since the golfers are interchangeable we use an unnamed type (line 3) to prevent symmetry from being introduced. A set of partitions is used to represent the groups of golfers that will play together, the attributes on the partition ensure that each group is of the required size. For the socialisation constraint, that is to ensure that each golfer plays another exactly once, we check that the size of the intersection of any two groups is less then two in any pair of weeks.

3.2 Conjure

The Conjure [Akg14] automated modelling system uses a refinement based approach akin to compilation. Conjure operates at the class level, meaning that even though the specification may be parameterised the refinement process only needs to be done *once* per problem class regardless of the number instances that need to be solved.

From a given specification Conjure can produce many models, resulting from the different rules that can be applied. The next step is to select the models to use, there are two main strategies for model selection: Heuristics such as *compact* [Akg+13a] work *during* the refinement process and try to give a reasonable model quickly, which is useful when writing an Essence specification. Alternatively methods such as racing [Bir+02] can determine the most appropriate model(s) to use, but require *discriminating* instance data to be successful on parameterised problem classes. Chapter 5 demonstrates how to generate this data automatically.

3.2.1 Process

Conjure works by utilising a set of rules to refine an Essence specification. The rules can be separated into two broad categories namely *representation selection* and *expression refinement*.

Representation selection rules refine a domain to a more concrete domain. A rule (Figure 3.4) comprises: an identifier, the resulting expression, the preconditions that must be satisfied, and a list of special cases that require extra constraints to be added. An & introduces a variable that be can be referenced, `refn` is the resulting expression a matrix in our example. Figure 3.4 illustrates how a (total) function mapping integers could be refined. Since the domain of the function is integers and we are mapping all of them, we can use these integers as indices of a one-dimensional matrix. In addition refinement rules can impose supplementary constraints. In our example the function being injective and total implies that all the elements of the matrix will be unique, hence the `allDiff` constraint is introduced.

Figure 3.4: Rule for the Function1D representation

```
~~> Function~1D
~~> matrix indexed by [&fr] of &to
    where &fr hasType `int`

*** function (total) &fr --> &to

*** function (total, injective) &fr --> &to
    ~~> allDiff(refn)
$ Other attributes cases omitted.
```

Expression refinement rules transform expressions. There are two kinds, vertical rules that decrease the level of abstraction when transforming an expression, and horizontal rules that provide transformations without changing the level of abstraction. Vertical rules decide how an expression should be represented. As an example, equality over the explicit representation for sets (where the set is converted to a matrix and additional constraints are imposed to ensure the elements are unique) would be refined by quantifying over both matrixes and comparing the elements at each index for equality. Horizontal rules provide reasonable defaults for operations independent of the representation. This allows a new representation to be created with optimised representations for specific operations without needing to provide an implementation for every other operator.

The resulting models from Conjure are in the Essence$'$ language, suitable for Savile

Row. While Conjure can refine a specification it lacked a way of translating the refined Essence′ model back to the Essence level. Solution translation in Chapter 4 describes such a mechanism.

One of the models produced from refining Figure 3.2 is shown in Figure 3.5. The refinement rule from Figure 3.4 was used to convert the total functions `weights` and `values` to a 1d matrix.

Figure 3.5: Refinement of Figure 3.2.

```
language ESSENCE' 1.0
given n: int(1..100)
given totalWeight: int(1..1000)
given weights_Function1D: matrix indexed by [int(1..n)] of int(1..100)
given values_Function1D: matrix indexed by [int(1..n)] of int(1..100)

find picked_Occurrence: matrix indexed by [int(1..n)] of bool

maximising
  sum([toInt(picked_Occurrence[i]) * values_Function1D[i] | i : int(1..n)])

such that
  sum([toInt(picked_Occurrence[i]) * weights_Function1D[i] | i : int(1..n)])
      <= totalWeight,
  1 <= sum([toInt(picked_Occurrence[q1]) | q1 : int(1..n)]),
  sum([toInt(picked_Occurrence[q1]) | q1 : int(1..n)]) <= n
```

3.3 SavileRow

Savile Row [Nig+14] is a constraint modelling assistant that translates Essence′ to many different output formats. In addition to Minion which the toolchain focuses on, it can also produce MiniZinc for interoperability with other constraint solvers and can output SAT files for use with SAT solvers [NSM15]. Savile Row does not just translate the model to the target solver's input format it can improve the model drastically by applying a series of reformulations including common subexpression elimination and domain reduction (which prunes domains values from the upper and lower bound of each variable). While some of the reformulations individually only yield modest gains, the combination of the reformulations can result in significant gains solving previously unsolvable instances instantly [Nig+14]. In addition to using Savile Row as part of the process of solving an Essence specification, we utilise it when generating test cases in Chapter 6 where we can generate both Essence specifications and Essence′ models automatically.

3.4 Minion

Minion [GJM06] is a highly efficient constraint solver. Unlike other tools in the toolchain Minion takes as input a specific problem instance. This means each instance of a problem class has to be encoded specifically, which makes it harder to use manually, but means that the input format is much closer to the internals of Minion. Minion has many constraints with different performance characteristics, which the user would have to understand to produce an efficient model manually. Automated modelling tools can decide when each constraint is appropriate to use thus abstracting over the details for the user.

3.5 Summary

This chapter gave an overview of the Essence specification language and the associated toolchain. We described the toolchain starting with Conjure which takes an Essence specification at the class level and refines it into an Essence′ model for use with Savile Row. Following this we discussed how Savile Row improves Essence′ models by automatic reformulation as well as it's many output formats. This was followed by a brief overview of the Minion constraint solver.

Chapter 4

Solution Translation

Our first contribution is a way to transform a low level Essence$'$ solution into Essence. The process can handle all aspects of the Essence language, including each abstract domain described in Table 3.1. The benefit to the user is that the resulting solution is shown in terms of the original specification, thus abstracting over the different ways the specification can be modelled. Solution translation is needed for the rest of the contributions:

- For instance generation (Chapter 5) we utilise it when using a constraint solver to generate instances.

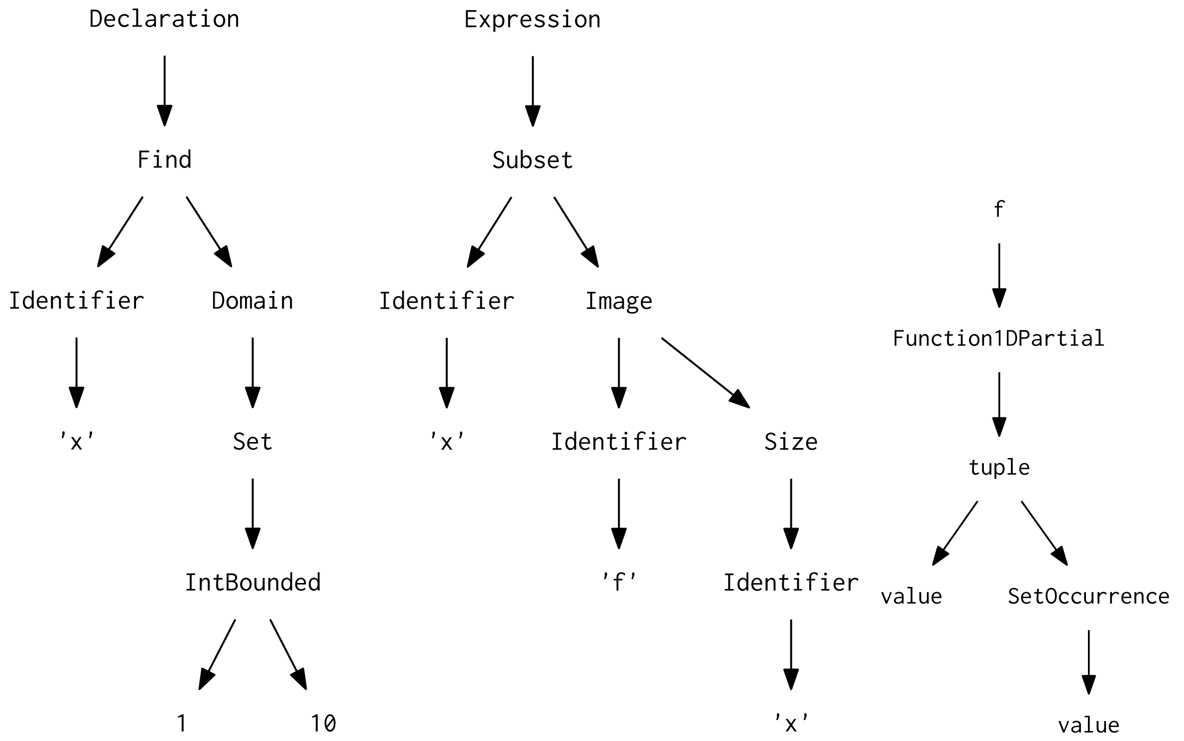
- For automated testing, we use it to perform validation of the solution at the Essence level.

4.1 Data Structures

Our solution translation does not act on the source of an Essence specification. It instead uses a parsed form, namely an abstract syntax tree (AST). The advantage is that it allows us to easily perform transversals and transformations upon the tree as compared to the source directly. We use Figure 4.2 to illustrate how a specification (Figure 4.1) is represented.

Figure 4.1: A small Essence specification

```
find x: set of int(1..5)
find f : function int(1..3) --> set of int(0..9)
such that x subset f(|x|)
```



(a) Find declaration AST: line 1

(b) Constraint AST: line 3

(c) Representation tree of the refinement of f

Figure 4.2: The structures used for Figure 4.1

In addition, solution translation uses a *representation tree* for each decision variable which stores representations that were chosen to model the variable. The representation tree in Figure 4.2(c) shows how a partial function mapping integers to sets of integers would be stored. This will be described in more detail as part of an worked example in Section 4.5.

4.2 Overview

Given an Essence′ solution, an Essence specification from which the Essence′ model was refined, a set of instances that were used, and a list of rules applied, a high level overview of solution translation is the following:

1. Instantiate all the Essence parameters into the Essence specification.

2. Read the domains from the Essence specification.

3. Resolve any domain references; this means that domains will not contain references to other declarations.

4. Create a representation tree for each variable assignment in the solution. Then associate the original domains with each assignment's tree.

5. Traverse the representation tree to create the Essence version of each variable assignment.

6. Introduce enums and unnamed types which require special handling.

Traversing the representation tree proceeds in the following fashion with pseudocode shown in Figure 4.4:

1. An accumulator of inverse rule functions is built up while traversing the tree.

2. When a leaf of the representation tree is reached these functions are applied in reverse.

4.3 Inverse Rule Functions

An inverse rule function transforms an Essence literal. Each rule takes two arguments, the Essence value to transform and the domain of the *resulting* value. Figure 4.3 gives a concrete example of an inverse rule function; The rule translates the occurrence representation of a set, a matrix of booleans back to the set itself. In this representation a element of the matrix is assigned the value true if it is in the original set. To reconstruct the set we pair elements of the matrix with the values of the domain. We then filter the pairs keeping only pairs whose first element are set to true.

```
Algorithm 1: occurrence($V, D$)
  Input: Matrix of booleans $V$, domain of the resulting set $D$
  Output: Set literal of type $D$, $D$ itself
  $ix \leftarrow$ all-values $D$
  $M \leftarrow$ zip $V$ $ix$
  $picked \leftarrow \{ v \mid (chosen, v) \in M,\ chosen \}$
  return ($picked$, $D$)
```

Figure 4.3: Occurrence representation transformation function

4.4 Process

Figure 4.4 shows the pseudocode for evaluating the representation tree. The representation tree is a multi-way tree with three kinds of nodes. At a `leaf` node the chain of rule functions are applied to the value associated with the leaf node. At a `branch` node, the inverse rule application is added to the chain (i.e. the variable `fs`). If the branch node has multiple child nodes, which occurs when a variable is channelled for example, the child node with the smallest tree depth is chosen. A `tuple` node is handled separately since it combines multiple variables.

4.5 Worked Example

As an example of how solution translation is performed we will consider the representation of a function mapping integer pairs to sets of integers (Figure 4.5) with one of the possible solutions in Figure 4.6.

Figure 4.5: Original Essence specification

```
language Essence 1.3
find f : function (int(1..3), int(1..3)) --> set of int(0..9)
such that
      forAll i : int(1..3) . |f(tuple(i,i))| = i + i
```

Figure 4.6: A Solution to the above Essence specification

```
language Essence 1.3
letting f be
      function((1, 1) --> {8, 9}, (2, 2) --> {6, 7, 8, 9},
              (3, 3) --> {4, 5, 6, 7, 8, 9})
```

We refine the decision variable `f` firstly using the following rule in Figure 4.7 which notes that the domain of the function is a tuple of integers hence we can utilise the elements of the tuple as indices of a 2d matrix. When the range of the function is not a simple type (i.e. int, bool or a matrix containing only simple types) further refinements rules are applied until they are. If the function is non-total we use marker elements to indicate whether each element was selected.

The outcome of the above rule application is further refined using the occurrence representation. The result is a 2d matrix of tuples which is then split into the two find declarations shown below. Other refinement rules have been applied to constraints on decision variable `f` which resulted in the constraints shown below.

```
Algorithm 2: evaluate_tree(tree, mapping, fs, prefix)
Input: Representation tree tree, variable's values mapping, prefix path prefix
switch tree do
    case Leaf(rep)
        /* retrieve the literal for this variable */
        varValues ← lookup(mapping, prefix + rep )
        /* Gets the inverse rule function associated with rep */
        repFunc ← nameToRepFunc(rep)
        return evalFs(fs, repFunc, varValues)
    end
    case Branch(name, children)
        selected ← selectChild(children)
        rep ← nameToRep(name)
        return evaluate_tree(mapping, rep + fs ,prefix + name, selected )
    end
    case Tuple(children)
        /* Transform each child */
        items ← { evaluate_tree(mapping, [] ,prefix, e) | e ∈ children }
        /* Convert the list to a tuple literal */
        tuple ← mkTuple(items)
        return evalFs(fs, idFunc, tuple)
    end
endsw
/* Evaluates a series of inverse rule functions */
Algorithm evalFs(fs, inner, data)
    switch fs do
        case []
            return inner
        end
        case  [g]
            return evalF(g, inner, data)
        end
        case (g:gs)
            return evalF(g,evalFs(gs, inner))
        end
    endsw

Algorithm evalF( (before,after), mid, value)
    vs ← before(value)
    mids ← map(mid,vs)
    res ← after(value, mids)
    return res
```

Figure 4.4: Evaluating a representation tree

Figure 4.7: Rule which transforms f

```
~~> Function~IntPair2D
~~> matrix indexed by [&a, &b] of &to
    where &a hasType `int`
    where &b hasType `int`

*** function (total) tuple (&a, &b) --> &to

*** function (total, injective) (&a, &b) --> &to
    ~~> allDiff(refn)

~~> Function~IntPair2DPartial
~~> matrix indexed by [&a, &b] of (bool,&to)
    where &a hasType `int`
    where &b hasType `int`

*** function tuple (&a, &b) --> &to
```

Figure 4.8: Essence′ refinement

```
language ESSENCE' 1.0

find f_FunctionIntPair2DPartial_tuple1:
        matrix indexed by [int(1..3), int(1..3)] of bool
find f_FunctionIntPair2DPartial_tuple2_SetOccurrence:
        matrix indexed by [int(1..3), int(1..3), int(0..9)] of bool
such that
forAll v__0 : int(1..3)
  . {(sum v__1 : int(0..9)
    . f_FunctionIntPair2DPartial_tuple2_SetOccurrence[v__0, v__0, v__1]
      *
      1) @ such that f_FunctionIntPair2DPartial_tuple1[v__0, v__0]}
  =
  v__0 + v__0
```

The Essence′ is solved using Savile Row and Minion, producing the following Essence′ solution. `f_FunctionIntPair2DPartial_tuple1` is a matrix of switches which indicates which mappings were selected. `f_FunctionIntPair2DPartial_tuple2_SetOccurrence` uses the indices as the domain of the function and the booleans as part of the occurrence representation of a set.

Figure 4.9: Solution to the Essence′ refinement

```
language ESSENCE' 1.0

letting f_FunctionIntPair2DPartial_tuple1 be [
 [true,  false, false ;int(1..3)],
 [false, true,  false ;int(1..3)],
 [false, false, true  ;int(1..3)]
]

letting f_FunctionIntPair2DPartial_tuple2_SetOccurrence be [
 [[false, false, false, false, false, false, false, false, true,  true  ;int(0..9)],
  [false, false, false, false, false, false, false, false, false, false ;int(0..9)],
  [false, false, false, false, false, false, false, false, false, false ;int(0..9)]],
 [[false, false, false, false, false, false, false, false, false, false ;int(0..9)],
  [false, false, false, false, false, false, true,  true,  true,  true; ;int(0..9)],
  [false, false, false, false, false, false, false, false, false, false ;int(0..9)]],
 [[false, false, false, false, false, false, false, false, false, false ;int(0..9)],
  [false, false, false, false, false, false, false, false, false, false ;int(0..9)],
  [false, false, false, false, true,  true,  true,  true,  true,  true  ;int(0..9)]]
]
```

The first step is to translate the occurrence representation where boolean flags are used to indicate if the element is in the set back to the set itself, using the previously discussed inverse rule function from Figure 4.3.

Figure 4.10: Step 1

```
language Essence 1.3
letting f_FunctionIntPair2DPartial_tuple1 be [
 [true,  false, false ;int(1..3)],
 [false, true,  false ;int(1..3)],
 [false, false, true  ;int(1..3)]
]

letting f_FunctionIntPair2DPartial_tuple2 be [
 [{8,9}, {},        {}],
 [{},    {6,7,8,9}, {}],
 [{},    {},        {4, 5, 6, 7, 8, 9} ;int(1..3)]
]
```

The tuple is then recreated in the next step.

Figure 4.11: Step 2

```
language Essence 1.3
letting f_FunctionIntPair2DPartial be [
  [(true,{8,9}), (false,{}),         (false,{}) ;int(1..3)],
  [(false,{}),   (true, {6,7,8,9}),  (false,{}) ;int(1..3)],
  [(false,{}),   (false,{}),         (true,{4, 5, 6, 7, 8, 9}) ;int(1..3)]
]
```

In step 3 & 4 we create the tuple's elements from the indices of the matrix which will eventually be the domain of the function.

Figure 4.12: Step 3

```
letting f_FunctionIntPair2DPartial be [
  [ (1,(true,{8,9})), (2,(false,{})),         (3,(false,{})) ;int(1..3)],
  [ (1,(false,{})),   (2,(true, {6,7,8,9})), (3,(false,{})) ;int(1..3)],
  [ (1,(false,{})),   (2,(false,{})),         (3,(true,{4, 5, 6, 7, 8, 9}))
    ;int(1..3)]
```

Figure 4.13: Step 4

```
letting f_FunctionIntPair2DPartial be [
(1,[ (1,(true,{8,9})), (2,(false,{})),        (3,(false,{})) ;int(1..3)]),
(2,[ (1,(false,{})),   (2,(true, {6,7,8,9})), (3,(false,{})) ;int(1..3)]),
(3,[ (1,(false,{})),   (2,(false,{})),        (3,(true,{4, 5, 6, 7, 8, 9}))
     ;int(1..3)])
```

In step 5 we create the mappings of the function.

Figure 4.14: Step 5

```
language Essence 1.3
letting f_FunctionIntPair2DPartial be [
[(1,1) -> (true,{8,9}),(1,2) -> (false,{}),       (1,3) -> (false,{})]
[(2,1) -> (false,{}),  (2,2) -> (true,{6,7,8,9}), (2,3) -> (false,{})]
[(3,1) -> (false,{}),  (3,2) -> (false,{}),
                       (3,3) -> (true,{4, 5, 6, 7, 8, 9})]
```

We then wrap the mappings in a function container. Note that mappings which have not been picked have not been removed yet.

Figure 4.15: Step 6

```
language Essence 1.3
letting f_FunctionIntPair2DPartial be function (
(1,1) -> (true,{8,9}), (1,2) -> (false,{}),       (1,3) -> (false,{})
(2,1) -> (false,{}),   (2,2) -> (true,{6,7,8,9}), (2,3) -> (false,{})
(3,1) -> (false,{}),   (3,2) -> (false,{}),
                       (3,3) -> (true,{4, 5, 6, 7, 8, 9})
)
```

Finally we remove the mappings which were not picked.

Figure 4.16: Step 7

```
language Essence 1.3
letting f be function(
    (1, 1) --> {8, 9},
    (2, 2) --> {6, 7, 8, 9},
    (3, 3) --> {4, 5, 6, 7, 8, 9} )
```

4.6 Unnamed Types

Consider the previously discussed social golfer's problem (repeated in Figure 4.17) which used unnamed types to avoid introducing additional symmetry to the specification. The difficulty appears when we wish to present a solution to the user, since the golfers are interchangeable we need a way of distinguishing them. We overcome this by explicitly naming each instance of the unnamed type that we require in the solution as shown in Figure 4.18

Figure 4.17: The Social Golfers in ESSENCE

```
language Essence 1.3
given w, g, s : int(1..)
letting Golfers be new type of size g * s
find sched : set (size w) of
                partition (regular, numParts g, partSize s) from Golfers

such that
    forAll week1 in sched . forAll week2 in sched, week1 != week2 .
        forAll group1 in parts(week1) .
            forAll group2 in parts(week2) .
                |group1 intersect group2| < 2
```

Figure 4.18: A solution to the above specification

```
language Essence 1.3
letting named_Golfers be new type enum {g1, g2, g3, g4}
letting sched be {
    partition({g1, g2}, {g3, g4}),
    partition({g1, g3}, {g2, g4}),
    partition({g1, g4}, {g2, g3})
}
```

4.7 Summary

This chapter introduced our first contribution namely *solution translation*, which is a method for translating the low level ESSENCE′ back to the ESSENCE level, meaning that the user can see the solution in terms of the original specification. The solution translation process is easily extensible, requiring only a single rule for a new representation.

Chapter 5

Generating Discriminating Instances

Following the automated modelling approach we described in Chapter 3, where an abstract specification is refined into numerous concrete models by following the various available refinement paths, we need a way to choose among them. We use an improved version of *racing* [Akg+13a] to perform the model selection process. This process assumes the existence of relevant training instance data for the problem class, which is used to determine the effectiveness of the refined models; these instances need to be *discriminating*, that is to say only a subset of the models should be able to successfully solve these instances. If all the models can solve an instance, or none of them can, then that instance is not useful for the purpose of model selection. In this chapter we discuss ways of generating this instance data automatically.

5.1 Racing

For the purposes of the model selection process, we use as the performance metric the sum of the (CPU-)time taken for Savile Row to instantiate the model, in addition to Minion's solving time. The reason that Savile Row's time is included is that it applies advantageous instance specific optimisations such as common sub-expression elimination [Nig+14].

For each instance we generate, we perform a race [Bir+02]. Given a parameter $\rho > 1$ a model is ρ-dominated by a second model if the performance metric of the second model (in our case CPU time) is ρ times faster the first. Resulting from this, the winners are the models which are not ρ-dominated. So that trivial instances are not used to discriminate, we exclude instances that can be solved in under a second.

Each model is entered into the race and for efficiency we terminate any model that is ρ-dominated (i.e. the runtime is ρ times greater than the current fastest model) since it will have no bearing on the results. Another efficiency improvement is that the order in which we run the models is dynamically chosen; the models that have shown superior performance previously are raced first. In the presence of no information such as the case of the first race, we place the model generated by the compact heuristic[1] at the front of the model ordering since, as shown by Akgün et al. [Akg+13a]. There is evidence that the model produced by the compact heuristic has reasonable performance across a range of problem classes.

So that the instances can be meaningfully compared we assign a *discrimination* value based on the percentage of the models illustrated in Figure 5.1. The discrimination value is defined as the D/T where T is the total number of models and D is the number of models that were dominated. We can further utilise this value to guide our instance generation as shown in our *Markov* method in the following section.

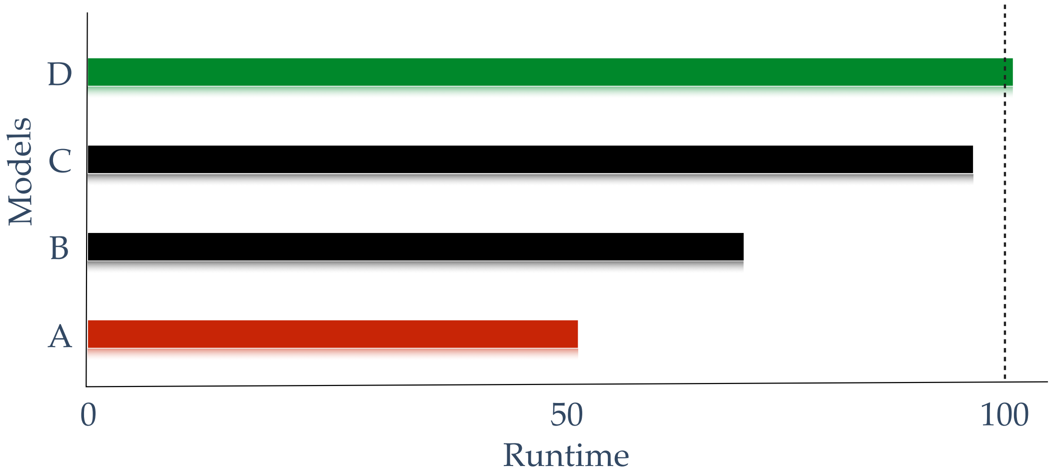


Figure 5.1: Illustration of the discrimination value. Model A ρ-dominates Model D but *not* Model A & B hence the discrimination value is 0.25.

Problem classes may not have an overall set of ‘winning’ models, that is the instances have a set of non-intersecting ‘winners’, in this situation we have *fracturing*. We say a set of instances is ρ-fractured if every model that was raced was ρ-dominated on at least one other instance i.e. there is no set of winners. In the case of fracturing we must be careful when

[1] The compact heuristic greedily chooses the refinement choices that produces the smallest expressions.

producing the results for the complete series of races. We start by finding a minimum hitting set $\{m_1, m_2, m_3, \dots\}$ that contains enough 'winners' to cover each instance. Following this we define the set of models M_i as the models that won *every* race that m_i won. The resulting set $\{M_1, M_2, M_3, \dots\}$ represents a portfolio of the winners over every fracture in the instance space. Each M_i is distinct since we could find a smaller hitting set if the intersection was non-empty.

In the simpler case where there is no fracturing, M_1 the only element of the results is unique. For the fractured case M_i is not uniquely defined since the choice of the hitting set determines the resulting set. However, it gives us a representation of the instance space.

When fracturing occurs, a *fracture* is a set of models where all models were dominated by a model from another fracture.

To illustrate the calculation of the resulting set consider a race with six models (A-F) resulting in the three fractures shown in Figure 5.2

```
{A, C}
{A, B, C, D, F}
{E, F}
```

Figure 5.2: The three fractures for our example.

A possible minimum hitting set is $\{A, F\}$. To calculate the resulting set we consider each member of the hitting set in turn and combine the model with other models that won the same races. Consider model A, model C won the same races hence it is included. Model F is the simpler case since no other models won the same races. The resulting set is then $\{\{A, C\}, \{F\}\}$.

5.2 Method for Generating Instances

The instance space that we sample from is defined by parameters of the problem class. To demonstrate consider Langford's Problem[2] (Figure 5.3), a simple example of a class with two *independent* integer parameters resulting in a two dimensional parameter space. A more

[2] Arrange K sets of the numbers 1 to n such that each occurrence of the number i is i places from the last.

complex example is that of the Knapsack problem[3] (Figure 5.4), which contains two functions that are *dependent* on the other integers. Knapsack's first integer n determines the domains of the two functions (which encode the value and weight of the items) in addition to the number of items available. Our methods for generating discriminating instance are described below. Each of the methods, runs a number of races and combines the results of these races in the manner described in Section 5.1.

Figure 5.3: Langford's Problem in Essence

```
language Essence 1.3
given k : int(2..10)
given n : int(1..50)
letting seqLength be k * n

letting seqIndex be domain int(1..seqLength)

find seq : function (total, surjective) seqIndex --> int(1..n)

such that
    forAll i,j : seqIndex , i < j .
        seq(i) = seq(j) -> seq(i) = j - i - 1
```

Figure 5.4: A Knapsack problem in Essence

```
language Essence 1.3
given n : int(1..100)
given totalWeight : int(1..1000)

given weights, values : function (total) int(1..n) --> int(1..100)

find picked: set(maxSize n, minSize 1) of int(1..n)

maximising (sum i in picked . values(i) )

such that
    (sum i in picked . weights(i) ) <= totalWeight
```

5.2.1 Undirected

The undirected method works simply by running a series of instance races using samples drawn from the instance space. The way a sample is generated is described in detail in section 5.3.

[3] Given a set of weighted items with associated values, find the number of each item to include in the knapsack such that the total weight is less then or equal to some predefined weight limit while maximising the total value of the items.

5.2.2 Markov

Our second method, Markov is inspired by Markov chains [Str14]. This assumes a priori that discriminating instances are likely to be near to other discriminating instances by some distance metric, and that non-discriminating instances are also near each other in a similar fashion. Hence we have developed a method resembling a Markov chain that traverses the instance space and is attracted to areas of discriminating instances and repelled from areas of non-discrimination.

The distance metric that we utilise is defined per parameter type:

Integer The absolute difference between the two integers.

Enumerated & Unnamed types By mapping the values of these types to integers we use our distance metric for integers.

Tuple The pairwise distance of each element of the tuple, aggregated using the Euclidean distance.

Set Given two sets X and Y the distance metric is calculated as $\sqrt{|S \setminus T| + |T \setminus S|}$, that is it calculates the number of elements that the sets disagree on.

Relation By treating the relation as a set of tuples, we use our distance metric for sets.

Total Function Given two functions f and g we calculate the distance (using our distance metric) between $f(i)$ and $g(i)$ where i is defined for both functions, then aggregate using the Euclidean distance.

Matrix Given two matrixes f and g we calculate the distance (using our distance metric) between $f[i]$ and $g[i]$ then aggregate using the Euclidean distance.

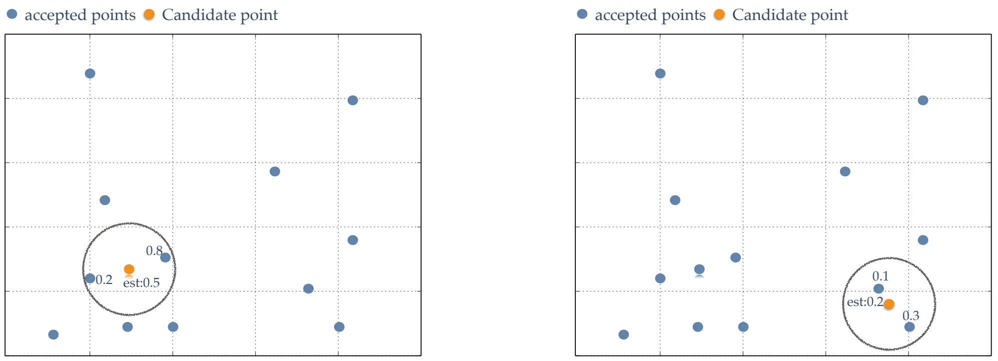
To calculate the overall distance we aggregate the individual components using the Euclidean distance. The initial instance is a special case since we have no previous data therefore we use the method in Section 5.3 to generate it.

Once we have generated an instance, we either reject or accept using the criteria below. If a point is rejected then we generate a new point until we accept it.

To determine if the candidate instance x'_i should be run we use the following acceptance function:

$$A(x_{i-1}, x'_i) = \frac{G'(x'_i)}{G(x_{i-1})}$$

G' estimates the discriminatory quality of the instance in the closed interval $[0, 1]$, utilising the previously accepted instances' quality values within a radius of influence r. The radius of influence is set to 10% of the greatest possible distance between any two instances using our distance metric. $G'(x'_i)$ is the mean of the set of all previous accepted instances within distance r of x'_i – if the set is non-empty $G'(x'_i)$ returns the mean of the true quality value for the set otherwise we return the middle value 0.5. $G(x_{i-1})$ gives the true quality of x_{i-1}. Finally a pseudorandom number a is generated within the closed interval $[0, 1]$, and x'_i is accepted if $A(x_{i-1}, x'_i) \geq a$. If $G'(x'_i)$ is greater than $G(x_{i-1})$ then the candidate instance is always accepted.


● accepted points ● Candidate point
0.8
0.2
est:0.5
● accepted points ● Candidate point
0.1
est:0.2
0.3

(a) A candidate point that has a 50% change of being accepted (b) A candidate point that has a 20% change of being accepted

Figure 5.5: A two dimensional parameter space used to illustrate how the next instance to be run is chosen. The estimated quality of a candidate instance is the average of the instances in the range of influence show by a circle in our example. The candidate instance in (a) is accepted hence it was run and shown in (b) as blue.

5.2.3 SMAC

Our third and final method uses SMAC [HHL11], an automatic algorithm configuration system. When given an algorithm, a set of instances, information on its parameters, it finds a set of parameters for the algorithm that delivers the best performance on the set of given instances. Our goal of finding discriminatory instances is a similar task. The problem class's specification is the algorithm, the parameters to SMAC are the particular `givens` of the problem class. The set of models refined from the specification are the 'instances' given to SMAC. The objective is then to find the set of models ('instances') that has the optimum discriminatory property.

Information about the parameters have to be encoded into SMAC's input format, consisting of integer and categorical variables. Integer `givens` are converted to integer parameters in SMAC directly using the ranges specified in the ESSENCE specification. More complex givens such as `functions` or `relations` are decomposed into multiple SMAC variables and reconstructed just before running a race. An extra complication arises when encoding dependent parameters, such as `weights` which depends on the given integer `n` in the Knapsack problem (Figure 5.4). We have to encode sufficient parameters for SMAC such that the maximum size of the complex given can be generated. The irrelevant parameters are ignored when producing an instance with smaller values of the dependency (e.g. `n` in our example). While SMAC can handle large parameter spaces (as shown in our experiments in Section 5.5), our conservative encoding of the parameter space may impede SMAC's ability to cover the space effectively. Additionally SMAC does not support arbitrary constraints between the parameters which we utilise in some problems such as the Progressive Party Problem in Figure 5.6 on line 6. To overcome this limitation we validate the instances generated by SMAC against the ESSENCE specification we use to generate the instances in our other methods, discarding instances which are found to be invalid. For our range of problem classes this validation process poses no significant overhead, taking less then a second per instance.

Figure 5.6: The Progressive Party Problem

```
1  language Essence 1.3
2  given n_upper, n_boats, n_periods : int(1..100)
3  letting Boat be domain int(1..n_boats)
4  given capacity, crew : function (total) Boat --> int(1..n_upper)
5
6  where forAll i : Boat . crew(i) <= capacity(i)
7
8  find hosts : set of Boat,
9       sched : set (size n_periods) of function (total) Boat --> Boat
10
11 minimising |hosts|
12
```

```
such that
    forAll p in sched . range(p) subsetEq hosts,
    forAll p in sched . forAll h in hosts . p(h) = h,
    forAll p in sched . forAll h in hosts . (sum b in preImage(p,h) . crew(b))
                                            <= capacity(h),
    forAll b1,b2 : Boat , b1 != b2
        . (sum p in sched .  toInt(p(b1) = p(b2)) ) <= 1
```

5.3 Sampling Methods

Our methods apart from SMAC, which has its own sampling process, rely on instances drawn from the problem class under consideration. Since the parameter space can be infinite, we need to provide some guidance on the limit to use. The bounds of the parameter spaces were chosen such that there would be billions of possible instances for problem classes with abstract domains. For Langford's Number problem which has only two integers for its parameters the bounds are much smaller since larger values for the parameters render the problem trivially unsolvable. Two sets of bounds were used on the Social Golfers and the Warehouse Location Problem to demonstrate the scalability of our methods in Section 5.5.1.

Ideally we would like to have the possibility of picking each instance with the same probability (i.e. uniform sampling), since at the beginning of our method we don't have any information to suggest that one area is significantly more discriminating than another. When the parameters of the problem class are independent, as is the case in the Social Golfers Problem and Langford's Problem, uniform sampling is simple: we simply generate the values (the two integers in Langford's) independently. To demonstrate why non independent parameters pose a complication consider the Knapsack problem from Figure 5.4, where the *given* functions on line 5 depend on the input parameters n. If we generate n first because of the dependency, the function mappings would be biased since there are many more functions for large values of n. This means an individual function with a smaller number of mapping is more likely to be picked even though there are many more functions with larger number of mappings.

One way of overcoming this is to generate *all* possible values in the instance space then select an instance uniformly from this set. We encode this simple enumeration as a constraint problem. We can create the enumeration specification $E*$ automatically from the original specification E by replacing each parameter (`find`) with a decision variable (`given`) and discard the rest of the specification.

To refine $E*$ the enumeration version of the specification we utilise the compact heuristic, which is sufficient because enumeration problems are usually easy. We then use Minion to find all solutions. We call this method where we pick an instance from the set of all possible generated instances `uniform`.

In a simple case such as in Langford's Problem the transformation $E*$ would result in the following:

```
find k : int(2..10)
find n: int(1..50)
```

When performing the transformations, we have to be cautious, applying it directly to problem classes with dependent parameters such as the Knapsack Problem results in the following which is invalid:

```
find n : int(1..100)
find totalWeight : int (1..1000)
find weights, values : function (total) int(1..n) --> int(1..100)
```

This is invalid since a decision variable defines the size of another decision variable's domain. The solution to overcome this is to leave n as a parameter, solve for each value in its domain and take the union of the solutions to these instances. In section 5.3.1 we describe how we automate the generation order for the givens in which we minimise the number of stages (i.e. the number of times we perform the enumeration process) required to generate each instance.

The main drawback of the `uniform` method is that of scalability. An alternative which we call `solver-random` is to use a constraint solver with a random variable ordering and take the first solution. This is biased for similar reasons, since the distribution of solutions is unlikely to be uniform but on the other hand is much more scaleable.

To compare our alternative sampling method `solver-random` to `uniform`, we run an experiment on two problem classes, namely the Warehouse Location and the Progressive Party Problem [Gen+14]. To perform a comparison between the two sampling methods we needed to restrict the instance space so that `uniform` is feasible which was achieved by restricting the instance space to around a billion instances. Each problem class was run three times, to assess the efficiency of the sampling methods we compare the number of non-dominated models produced by each method. For the Warehouse Location Problem, both sampling methods produced identical results reducing the number of models in half. For the other class the Progressive Party Problem `solver-random` performs better reducing the number of models to quarter of what `uniform` produces.

These experiments provide some evidence that `solver-random` is a reasonable sampling method. Resulting from the restriction on the size of the instance space to accommodate `uniform` the number of non-dominated models is quite large. Nevertheless `solver-random` can crucially scale to more challenging instances, which provide more discrimination as shown in proceeding sections where `solver-random` is used exclusively.

5.3.1 Generation Order

As discussed in Section 5.3 in the presence of dependent parameters we have to identity the dependencies of these parameters first, n in Knapsack Problem for example. This process may have $m - 1$ stages to complete first if every parameter is dependent on the parameter before it. For 'simple' domains namely Booleans, integers and enumerations we can generate these values directly, for other types we apply the enumeration process at stage i and feed the values into stage $i + 1$.

Using CSPlib [Jef+99] as a source of problems we find that many problems can be modelled naturally using two stages of generations. The generalised version is still useful since when we generate random specifications for the purposes of test case generation (Chapter 6) there can be an arbitrary level of dependencies.

We first build a directed graph representing the dependencies between the parameters with a directed edge from the dependent given the parameter it depends on: `weights` $\longrightarrow$ `n` in the Knapsack Problem (Figure 5.4) and its dependency graph (Figure 5.7).

We represent this graph problem as a constraint problem (Figure 5.8) where we wish to find the `levels`, a mapping which specifies which givens are generated together. Each successive level requires the previous levels' parameters to be generated first. As we minimise the number of levels required for the generation process, we put each parameter in the highest level possible. This makes the simpler case when the givens only depend on 'simple' domains faster since they can be generated directly. To illustrate why this is advantageous consider a

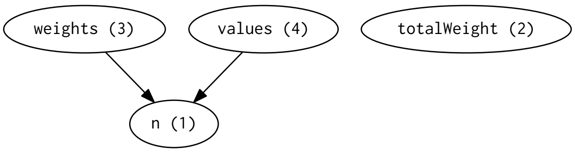


Figure 5.7: The Knapsack Problem's dependency graph

problem with three parameters: A an integer, B a set with a max size of A and C a function. A needs to generated before B but C can generated at any time. If we put C in the same level as A we need to perform the refinement and solving process discussed previously twice. On the other hand if we put C and B on the same level we can generate A, an integer, directly meaning we only need to do the process once, improving efficiency.

Figure 5.8: Finding the generation order in Essence

```
language Essence 1.3

$ Number of nodes (givens).
given N : int(1..20)
letting num_nodes be domain int(1..N)

$ Directed edges, (a,b) means a depends on the value of b.
given Edges : set ( maxSize (N * (N-1)) / 2 ) of (num_nodes, num_nodes)

$ The generation order indexed by the node id.
find ordering : function(total) num_nodes --> num_nodes

$ The groups of node to generate together.
find levels : function num_nodes --> set (maxSize N) of num_nodes

find levelsNeeded : num_nodes

such that
  $ Nodes which depend on other nodes are placed first
  forAll (fro,to) in Edges .
    ordering(to) < ordering(fro),

  levelsNeeded = max(range(ordering)),
  levelsNeeded = max(defined(levels)),

  forAll i in range(ordering) .
    levels(i) = preImage(ordering,i)

$ We want the nodes in the latest level possible
minimising max(range(ordering)) * 1000 - sum([ i * |j| | (i,j) <- levels  ])
```

Figure 5.8 shows how the problem could be modelled in Essence. We give as input a

Figure 5.9: The Instance from Figure 5.7 with its solution

```
$ Instance
letting N be 4
letting Edges be {(3, 1), (4, 1)}

$ Solution
letting levels be function(1 --> {1}, 2 --> {2, 3, 4})
letting levelsNeeded be 2
letting ordering be function(1 --> 1, 2 --> 2, 3 --> 2, 4 --> 2)
```

set of edges. For an edge (a, b) means that a depends on b, $(3, 1)$ represents that `weights` depend on `n` in the knapsack problem. We constrain the `ordering` decision variable to ensure the dependencies are generated first. `levels` partitions the decision variables into individual levels which have to be generated in ascending order. The objective function minimises the number of levels and penalises placing a decision variable in a lower level.

5.4 Models to Race

In the presence of hundreds of thousands of models we have to take a different approach to racing since it is infeasible to generate every possible model. To overcome this problem we utilise random sampling in addition to heuristics to produce a portfolio of models that gives superior performance.

To enhance the diversity of the generated models we include the models generated by restricting the refinement process to consider only models containing no channelling between the variables. Since the number of models produced by this restriction is small (usually under ten) it is an efficient way of adding diversity to the portfolio. In addition we include the model generated by the compact heuristic to act as a baseline for the racing process.

Because of fracturing we can not discard any models which were dominated in a race, in addition our quality metric of an instance is only comparable when using the same models. If new models were added then we would have to run the new models on *every* previous instance for the qualities to be meaningful, hence we opt for the simpler method of pre-generating the models to be raced.

The process proceeds as follows:

To generate the N models, we start off by taking the intersection of the models produced by the compact heuristic and the models produced by the *no-channelling* restriction to give C models. We then generate models using random sampling, removing duplicates (in addition to removing models which only differ in the naming of variables which can happen if the model generated by compact is a member of the set of models generated by the no-channelling restriction and the rules are applied in a different order) until we have generated $N - C$ unique models, thus resulting in N models in total. This approach can be trivially parallelised for efficiency.

To ensure that this method is applicable to problem classes where N is close to the total number of models (since we might not know the difference in magnitude between N and the total number of models) we do the following check before the random sampling:

- Perform the generation process with no restrictions until we generate $N * 2$ models.
- If the number of models is less then $N * 2$ then we pick a random subset with maximum size N.
- Otherwise we perform the sampling process described above.

This means that in the case that the total number of models is less then N as is the case in the Social Golfers (which has 27 models) we select all of them.

5.5 Experimental Results

In this section we report the results of our instance generation methods. Following from Section 5.3 we use `solver-random` throughout. We experiment on eight problem classes as summarised in the tables below . For each class we run three independent races on the sampled models ($N = 64$) . For both `Markov` and `Undirected` we run 64 races with *each* model being raced receiving 600 seconds of CPU time to solve the race's instance. Rather then a maximum number of races, SMAC requires a total time budget: to make sure that it is not disadvantaged we specify the total as the maximum CPU time of the others methods took to complete the race sequence. We repeat the *whole* process for three different solver heuristics, the resulting experiments taking approximately eight CPU-years (66,228 hours) to finish.

The three solver heuristics decide the variable order. `Static` is a lexicographical ordering, `sdf` (Smallest Domain First) picks the variable with fewest values in it dynamically and `wdeg` which stores and utilises extra information about failed assignments to guide the search more efficiently.

For all the experiments, each model was given 4GB of RAM and run on a 2.1GHz machine with an AMD-based architecture. Each problem class for all three methods was run on the same machine for consistency. Since all the classes except the Social Golfers have over 10,000 models we use the method described in Section 5.4 to give us a reasonable number of models to race.

For each problem we record: the number of models raced; The output set(s) as described in Section 5.1; the number of fractures found; and the steps until convergence. for `Markov` and `Undirected`. The steps to convergence is the number of races needed after which no models were pruned from the fracture(s). The specifications used range from only having integers for the `given` statements to having highly nested `given` statements as shown in Table 5.1.

The main statistics which we will use for comparison are the steps to convergence and number of fractures found. The steps to convergence is the point at which no further information was found, a smaller number implies that the algorithm can produce its final resulting set quicker. Many problem classes are fractured, meaning that there is no one model that dominated all others, across the instance space. If an algorithm was able to detect fracturing it can produce a resulting set with better performance across the instance space.

We described each problem class in detail:

The Knapsack Problem All the methods perform well on this problem class. Each method was able to find two fractures: the explicit representation of the set of items in the knapsack for unsatisfiable instances, and the occurrence model for the satisfiable instances. Both `Markov` and `Undirected` show some variability in steps to convergence. In regards to the three heuristics, the same fractures were found. For `Undirected` the `sdf` heuristic was the slowest to converge, in one case finding the second fracture on the last iteration. SMAC was able to detect fracturing on all heuristics.

Problem	Givens

Diagnosis
[Alf+04; Aze]
	1 int
	`function int --> bool`
	`function int --> matrix of int`
	`function int --> function matrix of bool --> bool`

Knapsack Problem [Sel09; Akg]	2 ints 2 `function int --> int`
Minimum Energy Broadcast [BB07; BB]	3 ints `matrix of matrix of int`
Progressive Party [Smi+95; Wala]	4 ints 2 `function int -> int`

Rehearsal Problem
[Smi]
	2 ints
	`function int --> int`
	`relation of int * int`

Social Golfers [HLS05; KH01; Har]	3 ints
Warehouse Location Problem [Van99; Hni]	3 ints 2 `function int --> int` `function (int,int) -> int`
Langford's Number Problem [HSW04; Walb]	2 ints

Table 5.1: Summary of the eight problem classes in the experiments.

The Warehouse Location Problem Both `Markov` and `Undirected` were able to detect fracturing. In contrast to the Knapsack problem, the heuristic that is used to solve the instances affects the likelihood that fracturing is discovered. For the `static` heuristic only `Markov` was able to find three fractures. Using `sdf` each method apart from SMAC was able to find three fractures. Two fractures were found by each method using the `wdeg` heuristic but the size of the fractures found varied for SMAC.

Diagnosis This problem differs in that the size of the resulting set is relatively large and there is no fracturing. The `Markov` method finds smallest resulting set regardless of the heuristic. The heuristic used affects the resulting set size with `wdeg` having the smallest.

Minimum Energy Broadcast The heuristic that is used affected the resulting size. In the case of the `static` heuristic `Markov` and SMAC were able to find the smallest resulting set for the three fractures, with SMAC being more consistent with regards to the size of the fractures.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Knapsack-1	1,1	2	25	1,1	2	1,1	2	36
Knapsack-2	1,1	2	57	1,1	2	1,1	2	42
Knapsack-3	1,1	2	18	1,1	2	1,1	2	27
Warehouse-1	2,1	2	57	2,1	2	1,1	2	56
Warehouse-2	2,1,1	3	39	2,1	2	2,1	2	62
Warehouse-3	2,1	2	59	2	1	2,1	2	57
Diagnosis-1	16	1	49	16	1	17	1	38
Diagnosis-2	15	1	3	16	1	17	1	15
Diagnosis-3	17	1	51	17	1	16	1	7
MEB-1	1,1	2	39	1,2,1	3	1,2,1	3	38
MEB-2	1,1,1	3	64	1,1,1	3	2,1	2	19
MEB-3	1,1	2	42	1,3	2	1,1	2	5
PPP-1	3	1	22	1	1	3	1	14
PPP-2	1	1	44	2	1	1	1	49
PPP-3	3	1	36	18	1	3	1	40
Rehearsal-1	1	1	38	1	1	1	1	51
Rehearsal-2	1	1	8	1	1	1	1	44
Rehearsal-2	1	1	21	1	1	1	1	13
Langford-1	1	1	10	1	1	1	1	24
Langford-2	1	1	53	1	1	1	1	3
Langford-3	1	1	24	1	1	1	1	55

Table 5.2: Results for the problem classes over three independent runs for the `static` heuristic.

For the `sdf` heuristic all methods consistently find one model and no fracturing. On the `wdeg` heuristic `undirected` was able to finding the three fractures with the smallest resulting set.

The Progressive Party Problem While no fracturing occurred, the heuristic affects the rate of convergence. In particular for the `sdf` heuristic, the worst `Markov` run beats the best `Undirected` run. SMAC shows some variance in the resulting set.

The Rehearsal Problem While both `Markov` and `Undirected` were able to detect fracturing when using the `sdf` heuristic only `Markov` was able to find fracturing when using the `wdeg` heuristic. In general `Markov` converges faster then `Undirected`.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Knapsack-1	1,1	2	57	1,1	2	1,1	2	64
Knapsack-2	1,1	2	21	1,1	2	1,1	2	54
Knapsack-3	1,1	2	38	1,1	2	1,1	2	58
Warehouse-1	1,1,1	3	48	2,1	2	1,1,1	3	35
Warehouse-2	1,1,1	3	53	2,1	2	1,1,1	3	57
Warehouse-3	1,1,1	3	59	2	1	1,1,1	3	57
Diagnosis-1	16	1	41	17	1	17	1	64
Diagnosis-2	16	1	29	17	1	17	1	10
Diagnosis-3	16	1	60	17	1	17	1	24
MEB-1	1	1	22	1	1	1	1	17
MEB-2	1	1	5	1	1	1	1	10
MEB-3	1	1	57	1	1	1	1	44
PPP-1	3	1	1	1	1	3	1	47
PPP-2	3	1	1	3	1	1	1	45
PPP-3	1	1	37	1	1	3	1	64
Rehearsal-1	1,1	2	30	1	1	1,1	2	52
Rehearsal-2	1,1	2	51	1	1	1,1	2	58
Rehearsal-2	1,1	2	63	1	1	1,1	2	40
Langford-1	1	1	43	1	1	1	1	47
Langford-2	1	1	39	1	1	1	1	55
Langford-3	1	1	26	1	1	1	1	34

Table 5.3: Results for the problem classes over three independent runs for the `sdf` heuristic.

Langford Number Problem Both `Markov` and `Undirected` find a single model on all three heuristics. `Markov` converges faster then `Undirected` on the `sdf` and `wdeg` heuristic and converges slightly slower on the `static` heuristic. SMAC produces the same results on the `static` and `Undirected` heuristics namely a single model. On the `wdeg` heuristic it was able to find fracturing, which may be because it used its time more effectively on this particular problem class, running 40% more instances on the run that actually found fracturing.

Overall `Markov` was able to find the most fractures across the range of problem classes tested. Additionally `Markov` converges faster then `Undirected` which has a much larger degree of variance. While SMAC found fewer fractures overall with generally larger resulting sets, it was the only method to detect fracturing in Langford's Problem.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Knapsack-1	1,1	2	9	1,1	2	1,1	2	40
Knapsack-2	1,1	2	40	1,1	2	1,1	2	31
Knapsack-3	1,1	2	53	1,1	2	1,1	2	58
Warehouse-1	1,1	2	62	1,1	2	1,1	2	39
Warehouse-2	1,1	2	59	2	1	1,1	2	62
Warehouse-3	1,1	2	59	2,1	2	1,1	2	57
Diagnosis-1	17	1	34	16	1	17	1	44
Diagnosis-2	17	1	62	16	1	15	1	64
Diagnosis-3	13	1	57	17	1	16	1	29
MEB-1	1,2	2	46	1,1	2	1,3	2	41
MEB-2	1,9,1	3	26	1,3	2	1,1	2	49
MEB-3	2	1	47	1,1	2	1,1,1	3	38
PPP-1	2	1	60	3	1	3	1	35
PPP-2	1	1	2	-[1]	-	1	1	1
PPP-3	2	1	14	1	1	3	1	62
Rehearsal-1	1,2	2	63	1	1	1	1	37
Rehearsal-2	1	1	25	1	1	1	1	34
Rehearsal-3	1	1	4	1	1	1	1	16
Langford-1	1	1	25	1	1	1	1	60
Langford-2	1	1	41	1	1	1	1	47
Langford-3	1	1	52	1,1	2	1	1	54

[1] While SMAC did find a single model which can solve a subset of the tested params, it did not dominate any other model because it took approximately 400 seconds to solve.

Table 5.4: Results for the problem classes over three independent runs for the `wdeg` heuristic.

5.5.1 Scalability

After we have run our methods for a problem class, we have produced a set of models which are appropriate for each fracture. We show through experiments that these results generalise if we expand the instance space.

Consider the Warehouse problem, a fractured problem class which has many abstract domains as input data, in addition some of the parameters are *dependent* on other parameters. To test the hypothesis that our method scales to larger instance spaces we double the upper bound of each integer inside each parameter of our initial specification. This results in a vastly larger instance space. For the expanded specification we use the same setting as discussed previously and group each heuristic separately for ease of comparison.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Warehouse$_{30}$-1	2,1	2	57	2,1	2	1,1	2	56
Warehouse$_{30}$-2	2,1,1	3	39	2,1	2	2,1	2	62
Warehouse$_{30}$-3	2,1	2	59	2	1	2,1	2	57
Warehouse$_{60}$-1	2,1	2	55	2,1	2	2,1	2	55
Warehouse$_{60}$-2	2,1	2	52	2	1	2,1	2	32
Warehouse$_{60}$-3	1,1	2	61	2,1	2	2,1	2	27

Table 5.5: Results for the Warehouse problem over three independent runs for the `static` heuristic.

For the `static` heuristic, Markov can still find two fractures on the expanded version but generally converges at a slower rate. Undirected converges faster but was unable to reduce the sizes of the fractures to the extent that Markov did. SMAC is however the most consistent when comparing the set of results of the original specification with that of the expanded version.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Warehouse$_{30}$-1	1,1,1	3	48	2,1	2	1,1,1	3	35
Warehouse$_{30}$-2	1,1,1	3	53	2,1	2	1,1,1	3	57
Warehouse$_{30}$-3	1,1,1	3	59	2	1	1,1,1	3	57
Warehouse$_{60}$-1	1,1,1	3	59	1,1	2	1,1,1	3	47
Warehouse$_{60}$-2	1,1,1	3	61	1	1	1,1,1	3	55
Warehouse$_{60}$-3	1,1	2	19	1,1	2	1,1,1	3	51

Table 5.6: Results for the Warehouse problem over three independent runs for the `sdf` heuristic.

On the `sdf` heuristic, Undirected is the most consistent, producing the fracture with the same size in every case. In comparison Markov shows a slight variance in the number of fractures found. In contrast to the `static` heuristic, SMAC was actually able to reduce sizes of the fractures on the expanded specification.

The `wdeg` heuristic shows similar results to the `sdf` heuristic, namely `Undirected` has the exact same results on every run, `Markov` having some slight variance and SMAC having increased performance.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Warehouse_{30}-1	1,1	2	62	1,1	2	1,1	2	39
Warehouse_{30}-2	1,1	2	59	2	1	1,1	2	62
Warehouse_{30}-3	1,1	2	59	2,1	2	1,1	2	57
Warehouse_{60}-1	1,1	2	54	1	1	1,1	2	35
Warehouse_{60}-2	1,1	2	39	1,1	2	1,1	2	26
Warehouse_{60}-3	1,1,1	3	36	1	1	1,1	2	34

Table 5.7: Results for the Warehouse problem over three independent runs for the `wdeg` heuristic.

We now consider the Social Golfers problem (Figure 4.17), which in contrast to the Warehouse Problem (having many abstract domains as parameters), only has three independent integer parameters. For this comparison, we run our racing experiment using two different restrictions: firstly `int(1..25)` for the three integers (`w`, `g`, `s`), than loosening the restrictions to `int(1..50)` giving an eight fold increases in the size of the instance space.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
SGP_{25}-1	1,1	2	53	1,1	2	1,1	2	19
SGP_{25}-2	1,1	2	64	1,1	2	1,1	2	45
SGP_{25}-3	1,1	2	62	2,1	2	1,1	2	23
SGP_{50}-1	1,1	2	37	1,1	2	1	1	10
SGP_{50}-2	1,1	2	50	24	1	1,1	2	20
SGP_{50}-3	1	1	56	24	1	1	1	35

Table 5.8: Results for the Social Golfers over three independent runs for the `static` heuristic.

With regards to the `static` heuristic, while `Markov` takes longer to converge, it finds fracturing more often when compared to `Undirected` on the expanded specification. While SMAC performs well in the more constrained version in the expanded specification it shows variability with respect to sizes of the fractures it finds.

While `Undirected` finds more fractures than `Markov` on the more constrained version of SGP, they converge at roughly the same rate. In contrast `Markov` converges faster for the expanded specification. SMAC shows some variance in sizes of the fractures found.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
SGP_{25}-1	1	1	36	1,1	2	1,1	2	39
SGP_{25}-2	1,1	2	56	2,1	2	1,1	2	60
SGP_{25}-3	1,1	2	37	2,1	2	1,1	2	34
SGP_{50}-1	1,1	2	41	1,1	2	1,1	2	56
SGP_{50}-2	1,1	2	48	1	1	24	1	64
SGP_{50}-3	24	1	64	24	1	1,1	2	51

Table 5.9: Results for the Social Golfers over three independent runs for the `sdf` heuristic.

	Markov			SMAC		Undirected		
Problem	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
SGP_{25}-1	1,1	2	44	1,1	2	1,1	2	28
SGP_{25}-2	1,1	2	48	1	1	1,1	2	35
SGP_{25}-3	1,1	2	28	1	1	1,1	2	52
SGP_{50}-1	1,1	2	26	1	1	1	1	55
SGP_{50}-2	1	1	22	1,1	2	1,1	2	33
SGP_{50}-3	1	1	4	1	1	1	1	6

Table 5.10: Results for the Social Golfers over three independent runs for the `wdeg` heuristic.

While `Markov` and `Undirected` find both find the same fractures, `Markov` always converges faster then `Undirected` on the large domain version of SGP. SMAC shows similar set of results when compared to the original

5.6 Limitations

We discuss the limitations of our automatic instance generation process and possible solutions. Consider Figure 5.10 (which is the problem of designing a curriculum such that the academic load is as similar as possible subject to a number of restrictions) and a sampled instance (Figure 5.11)

Our methods are able to select a single model from a selection of 64 models. A problem arises if `prereq` has cycles: the resulting instance is likely to be trivially unsolvable, hence the models selected may not have superior runtime performance on satisfiable instances.

A possible solution could be to utilise `where` statements, which place restrictions on the

Figure 5.10: The Balanced Academic Curriculum Problem [HKW02; HKW]

```
given n_periods, maxl, minl, n_courses, maxc, minc, n_credits : int(1..100)

letting Course be domain int(1..n_courses),
        Period be domain int(1..n_periods)

given prereq : relation of (Course*Course),
      credits : function (total) Course --> int(1..n_credits)

find curr : function (total) Course --> Period

such that
  forAll c1,c2 : Course . prereq(c1,c2) -> curr(c1) < curr(c2),
  forAll p : Period . (sum c in preImage(curr,p) . credits(c)) <= maxl /\
                      (sum c in preImage(curr,p) . credits(c)) >= minl,
  forAll p : Period . |preImage(curr,p)| <= maxc /\ |preImage(curr,p)| >= minc
```

Figure 5.11: A sampled instance for Figure 5.10

```
letting n_periods be 9
letting maxl be 86          $ max load
letting minl be 11          $ min load
letting n_courses be 3
letting maxc be 59          $ max courses
letting minc be 3           $ min courses
letting n_credits be 11
letting prereq be
  relation((1, 2), (1, 3), (2, 1), (2, 3), (3, 2))
letting credits be function(1 --> 6, 2 --> 7, 3 --> 1)
```

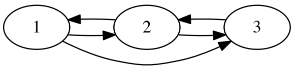


Figure 5.12: prereq of Figure 5.11 visualised showing the loops

values that a parameters can take[4], to specify that there should be no cycles in the prereq parameter, this would be mean the our sampling process would not generate these undesirable instances. The disadvantage is that this requires the user to specify the problem more precisely impeding usability.

5.6.1 Customisation

In addition to just using an enumeration specification E^* derived from the problem specification, the user can also add constraints to guide the instance generation process. Consider the BACP

[4] where statements were used in the Progressive Party Problem (Figure 5.6)

problem from Section 5.6. Figure 5.13 adds additional constraints to prevent cycles in the prereq variable. In contrast with the simpler where statements which can *not* reference existing decision variables or create new decisions, this extension method is more expressive by allowing both. An additional advantage is that unlike where statements which require the user to modify the original specification and hence perform the potentially expensive refinement process again, this process requires neither.

Figure 5.13: Adding extra constraints to prevent cycles in prereq of BACP

```
$ Generated
given n_periods: int(1..20)
find maxl: int(1..20)
find minl: int(1..20)
given n_courses: int(1..20)
find maxc: int(1..20)
find minc: int(1..20)
given n_credits: int(1..20)
find prereq: relation of (int(1..n_courses) * int(1..n_courses))
find credits: function (total) int(1..n_courses) --> int(1..n_credits)
$ Generated (end)

$ paths[i,j] is true if there is a directed path from i to j of prereq.
find paths: matrix indexed by [int(1..20), int(1..20)] of bool

such that

    $ (i,j) are directly connected.
    forAll i,j: int(1..20) .
        prereq(i,j) -> paths[i,j],

    $ (i,j) are connected if we can used a path via node k.
    forAll i,j: int(1..20) .
        forAll k : int(1..20) .
            paths[i,k] /\ paths[k,j] -> paths[i,j],

    $ prevent loops by disallowing both (i,j) and (j,i) from being connected.
    forAll i,j: int(1..20) .
       ! (paths[i,j] /\ paths[j,i])
```

5.7 Summary

This chapter discussed and explored three different methods for generating discriminating instances with the aim of automating constraint model selection. We described the implementation of these methods, how an arbitrary specification with *dependent* parameters can be handled automatically, and how the generated instances can be evaluated. Our experimental analysis shows that our methods can reduce a large number of models to a much smaller subset. In addition our methods can detect *fracturing* and find appropriate models for each fracture. Overall the `Markov` method found the most fractures and has best performance in most problem classes.

Chapter 6

Test Case Generation

Refinement-based automated modelling as employed by systems such as Conjure perform a complex process akin to compilation. To achieve this a set of refinement rules is applied to the given specification which then produces a series of Essence′ models. Our aim is to create an effective automated testing regime which can validate the various components of the toolchain described in Chapter 3. We use the automated generation of constraint specifications to achieve this goal. Our generator of test cases can cover all aspects of the Essence specification language including the generation of both optimisation and satisfiability problems. In particular, it can generate *parameterised* specifications utilising our instance generation methods from Chapter 5.

Examples of some of the errors we wish to find include the following (notable errors are classified and discussed fully in Section 6.5)

Invalid generation of Essence′ Where the two languages overlap, Essence′ has slightly different semantics from Essence, e.g. in Essence′ Booleans can be treated as integers, hence there is no `toInt` method, whereas booleans and integers are distinct types in Essence and conversion from a Boolean to an integer requires the use of the `toInt` method .

Incorrect models When either Conjure produces a model which produces invalid solutions, or Savile Row produces an invalid solution. If this happens we can validate the produced models to see if they are correct.

Type checking errors When the system infers the wrong type of an expression. An example of this was `( {} : `set of relation of (bool * bool)` )` which was inferred as a set of tuples because of a missing case in the code.

Missing refinement rules For example, the system discovered the lack of a refinement rule for functions of a bounded size, `function (maxSize 3, minSize 2) int(1..3) -> int(1..4)`. This caused the attributes to be ignored and an invalid solution to be produced. This error was found during the validation process. After we find a specification that causes a bug we can *reduce* that specification, to a typically significantly simpler form which still causes a similar bug to occur, this will be discussed in Chapter 7.

6.1 Overview

Figure 6.1 gives a simplified overview of the main processes and how they interact (which will be expanded in Section 6.4). The settings are configuration options such as the total time to use.

Generate: Utilises random sampling and the grammar of the language to generate specifications – any *valid* specification can be generated.

Refine: Uses a set of heuristics to produce a set of models to be solved.

Solve: A constraint solver (Minion in our case) is then used to evaluate each model produced.

Reduction: After an error (which can occur at any stage) has been found, we *reduce* the error as far as possible, to produce a minimal test case as discussed in Chapter 7.

Generalise: After finding a minimal test case, we produce addition information to give more insight into the error: for example does the error only occur on sets of a certain size such as the empty set. This is explored in more detail in Section 7.6.

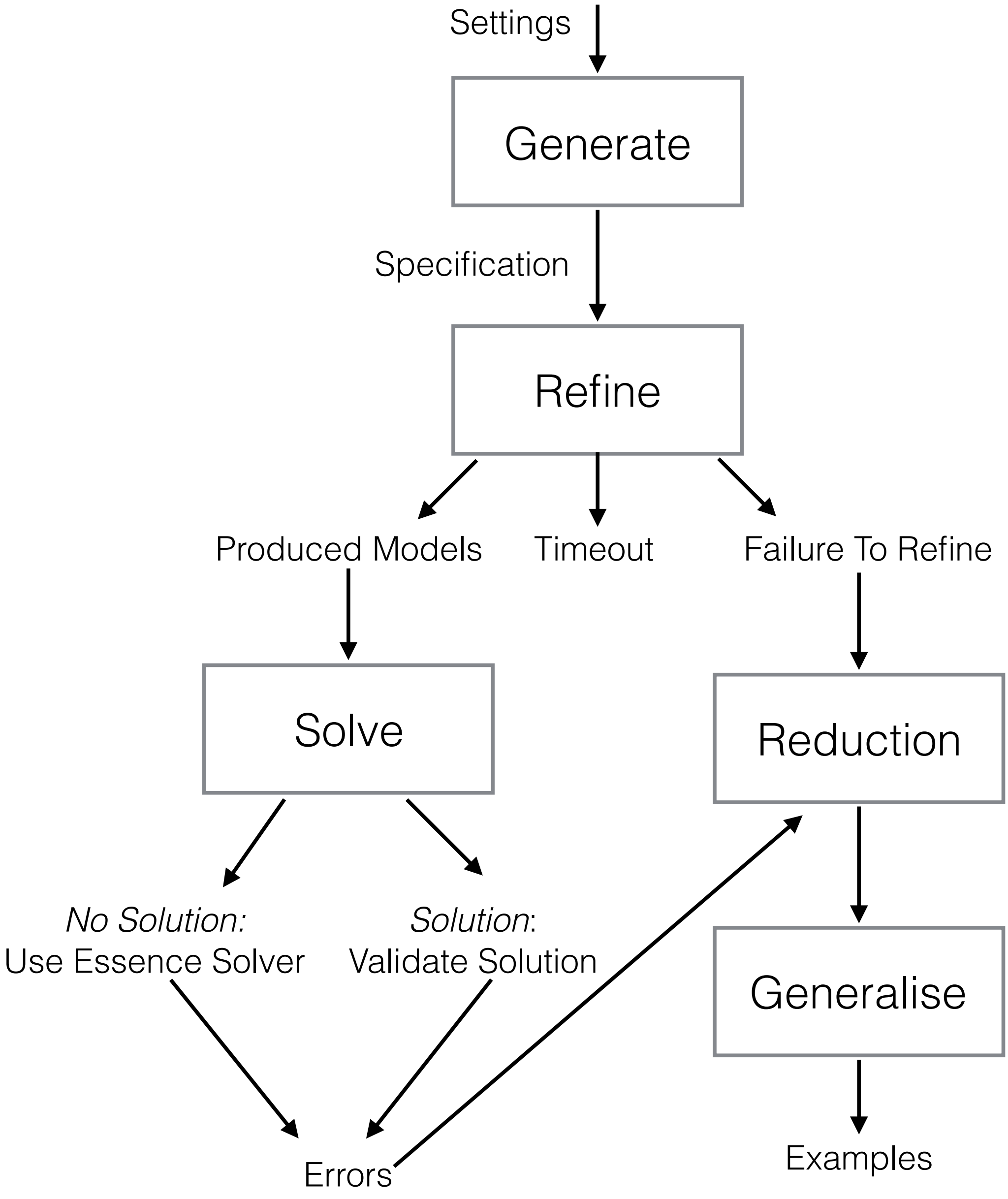


Figure 6.1: Overview of the testing approach

6.1.1 Model Validation

For a given Essence specification many models can be produced during refinement. A *valid* model is a model where are all the expressions have the correct type and methods have the correct arity. This definition also applies to specifications.

If the models are not consistent e.g. if one model is infeasible while the others produce a solution, we need a way of deciding which model(s) are correct.

We validate the models that produce solutions, by utilising solution translation from Chapter 3 to convert the low level Essence′ solution back to the Essence level. To validate we then substitute the translated (Essence) solution back into the original specification and evaluate the solution to see if the produced solution is correct.

The other case when none of the models produce a solution is more complex and costly. To validate, we use a simple backtrack solver (at the Essence level and detailed in Section 6.2.5), to check if the specification actually has no solutions.

Consider Figure 6.2 which we will use to explain the validation process.

Figure 6.2: Specifications who refinement produces inconsistent models

```
find var0:
        matrix indexed by [int(1..4)] of function (maxSize 0, injective)
                                             int(2..2) --> int(3..4)
such that
  var0[0] = function()
```

The specification produced four Essence′ models. Three of the models produced a solution, the other reported that the model was not solvable. To determine which of the model(s) are correct we validate the Essence′ solutions first since if *any* of Essence′ solutions, translated back to the Essence level, is a solution to the original specification then we know that the other model that produced no solution is an invalid refinement.

To validate we firstly convert the Essence′ solutions back to the Essence level producing `[function(), function(), function(), function()]` in all three cases. To validate the Essence solution, we compare it with the original specification, checking if the domain invariants (attributes and indices) are satisfied, this process has a validator for each Essence domain. We then substitute every `find` statement with its value from the Essence solution, checking if all the constraints evaluate to `true`. Since the solution *is* valid we can deduce that model that was insolvable is an invalid refinement.

6.2 Process

Our test case generator can generate any *valid* Essence specification. To illustrate, some examples of what can be generated include:

- Quantified expressions, generic operators (the `| |` operator returns the number of elements in inside any literal expression) and nested expressions.

```
find var1 : int(-1,0..3)
such that
    (sum q_2 : int(0..2) . var1) = |{true, 1 > 3 + -2}|
```

- Matrix comprehension.

```
[ l_1 - l_2 | l_1 : relation (maxSize 2) of (bool, int(2..3))
            , l_2 : relation of (bool, int(1, 6..7 )), true]
```

- An objective which can include an arbitrary expression to maximise/minimise

```
find var1: matrix indexed by [int(2, 2)] of bool
minimising sum([toInt(var1[l_1]) * 1 | l_1 : int(2..3)])
```

The generator can also test how various features of the language interact. Consider Figure 6.2 which contains an error caused by a missing case in the rule for function refinement leading to one of the produced models to output no solution. The specification contains the unusual combination of the attributes `maxSize` 0 and `injective` being applied to the `function`. The expected solution was a matrix of four empty functions.

An obvious way of generating an expression would be randomly selecting a type of the sub-expression to generate, backtracking when necessary if no expression could satisfy the chosen type. The problem with this approach is that the generated specification are usually so large that they can not be refined as well consuming lots of memory during generation, as compared to our approach. Our approach limits the depth of the expression that can be generated at each point in the specification. Additionally it uses a grammar which encodes when an expression should be generated based on the types and depth required, meaning that once we have calculated the allowed expressions to sample from we will not need to backtrack.

6.2.1 Depth Calculation

The depth is defined for each kind of expression recursively, to be 1 + maximum of depth of the sub-expressions. Integers, booleans and enumerated types have a depth 0 as the base case.

Figure 6.3 illustrates this by showing the depth of each of part of the expression {4, 6} union {2 + 1, 0}. The depth of an expression is used in the generation process to restrict the size of the specification that are generated. Additionally it is used in the generalisation process (Section 7.6) to generate complementary test cases.

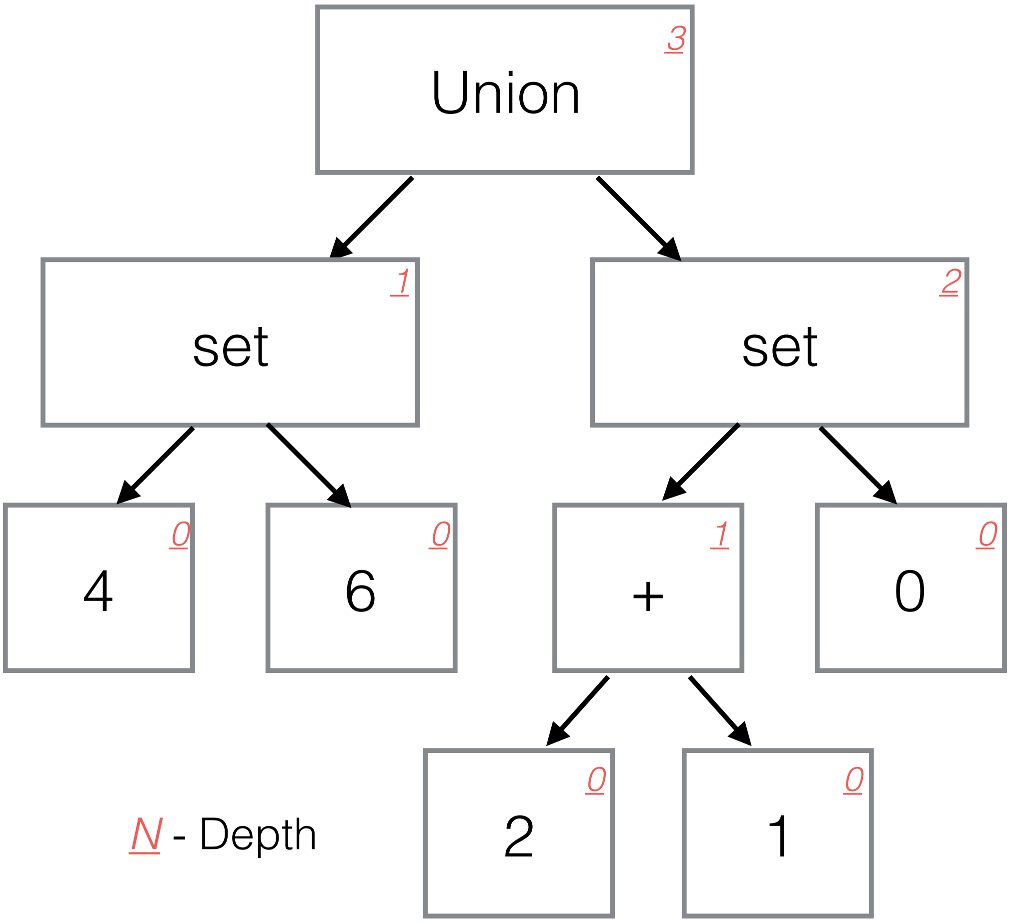


Figure 6.3: Depth of the expression {4, 6} union {2 + 1, 0}

6.2.2 Algorithm

An overview of the generation process is as follows: Given a `constraint_depth` and a `domain_depth` we do the following:

1. Generate up to N domains independently for the `find` statements. Each domain can be nested arbitrarily, up to the `domain_depth`. Examples include `bool` of depth 0 and `function (total, injective) int(1..4*|{4,2}| ) --> bool` of depth 4 which also

shows that domains can have arbitrary expressions inside them. A domain may have associated attributes (equivalent to placing constraints on the domain), which place restrictions on the values that can assigned to that domain. An example would be `surjective` for functions.

2. Generate up to N expressions for the constraints. This utilises weighted random sampling, where the weights for each aspect can be dynamically readjusted (see section 6.4). To generate an expression E of type τ we can choose:

   - A constant e.g. `{1,3,4}`.
   - A literal (which may contain an arbitrary expression inside) for example `function [1,2] --> |var1 union {1,3}|`.
   - A variable reference if applicable.
   - An operator/function e.g. `true \/ var2`, `max(var3)` or tuple indexing.
   - A matrix comprehension.

3. Optionally an objective function is generated where an arbitrary expression can be maximised/minimised. The only restriction on the objective being that a decision variable must be referenced and the resulting type must be an integer. Hence an objective is only generated if we have a decision variable that can be converted to an integer within the `constraint_depth`. A possible example is minimising the size of a set we wish to find.

   The reason for the restriction on the objective function is that if we produced an objective without a decision variable e.g. `maximising 1`, it would be discarded immediately by Conjure. Additionally since the objective is generated after the rest of the specification has been generated we know all the decision variables that could be used.

   After ensuring that it is possible to generate an objective which references a decision variable, we generate a random expression of type int. If this expression does not contain a decision variable we replace a sub-expression with an expression built around a decision variable. This is always possible because we already ensured that it is possible to convert one of the decision variables to an integer within the required depth.

6.2.3 Expression Selection

We use a grammar annotated with additional information (Appendix A) that encodes which types it is valid to generate each (sub-)expression as well as the 'requirements' that need to be satisfied for the generation of that (sub-)expression to be possible. The reason that these requirements are specified explicitly is that we dynamically change the weighting for the purposes of increasing language coverage and to increase the likelihood of finding different errors (Section 6.4).

The requirements are the list of expressions that we need to be able to generate. As an example `Union` works on functions, relation and sets. It also takes into account if we have enough `depth` left for the generation of the expression; 2 in the case of `{1} union var2`.

We give tuple indexing as an example of generating an operator since it has unusual requirements. Unlike other types, tuples can only be indexed by constant expressions. Furthermore the indexing expression has to be within the bounds of the tuple, unlike other types where out of bound indexing is just undefined. This is achieved by firstly generating constant expressions then adjusting them to be within the range of the tuple.

6.2.4 Running and Classification

Once we have generated a specification, we refine $N-1$ random models as well as using the compact heuristic. After each step in running the toolchain (Figure 6.1) we classify the output. If any of the outputs cause an error we proceed straight to the reduction process for that output. For each successful Essence′ produced we run Savile Row and Minion to solve the model. If a solution was produced we can cheaply check if it is valid after translating from Essence′ to Essence. In the other case where there was no solution produced, we use a simple backtrack solver to check if the specification actually has no solutions. Additionally we can detect inconsistencies in the models, such as the case when one model produces a solution and another outputs that the model is insolvable.

6.2.5 An Essence Solver

The aim of this solver at Essence level is about correctness not efficiency. While it would be infeasible in general to use it to check if a specification *actually* had no solutions, after applying the reduction process from Chapter 7 the specification would likely be much simpler, hence amenable to solving.

The backtrack solver is described in Figure 6.4:

Figure 6.4: Backtrack solver

```
process(domains, constraints, assignments)
    if (domains == []) // assigned all values
        return assignments

    dom = domains[0]
    foreach value in domValues(dom)
        new = assign(dom, value) + assignments
        if satisfies(new, constraints)
            return process(domains[1:], constraints, new)

    return null
```

- The `satisfies` function checks that the constraints are not violated by the new assignment.

- The `domValues` function produces all values in a domain. Since Essence allows a wide range of attributes to be placed upon a domain, we converted the problem of generating all values of a domain into a new enumeration specification and solved it to produce all the values, using the method from Section 5.3. This is safe since we can easily check that a particular value generated by the enumeration process is actually a member of the domain.

6.2.6 Errors found during Generation

In addition to generating failing test cases, bugs can be found during the creation of a specification. In the tuple indexing example discussed previously, a constant expression of type `int` within a specific range is needed. This is ensured by utilising Conjure's partial evaluator, to determine the value of randomly generated expression, i. If the partial expression fails, the evaluation failure is recorded as an output of the generation process and a random integer in the range of (1, size of tuple) is used instead. If the evaluation succeeded the adjustment is to choose a random integer x such that $i + x$ is in the required range.

6.3 Parameterised Specifications

Generating specifications with `given` statements is more complex. We firstly have to avoid introducing cycles in the domains that are referenced since it would impede the generation process of the givens. To create the givens we utilise the instance generation process from Chapter 5. In particular we need to synthesise the generation order (Section 5.3.1). Another complication is that a `given` can not depend on a decision variable.

If the generation of an instance fails because there is no instance that would satisfy the constraints on the domains, we try applying the reduction process (Chapter 7) to relax the restrictions on the domains. If this fails we discard the specification and continue the generation process.

Generating parameterised specifications allows us to the test the refinement of parameters with regards to the various available representations.

To demonstrate how this process works consider the givens of the generated specification in Figure 6.5.

Figure 6.5: Generated Givens

```
given given_1: int(1..3)
given given_2: int(2,5, 4)
given given_3: function (injective) int(1..5) --> int(3..given2)

find var_1: set of int(1..4)
such that given_3(var_1) = given_1
```

This specifications shows some of the complexities of the parameterised specifications since the function `given_3` depend on the integer `given_2`.

The next step is synthesise a generation order (Section 5.3.1) which results in `[ {given_2 }, {given_1, given_3} ]` . This means that the instance generation process is a two stage problem firstly generating `given_2` which the function depend on before generating the remaining givens. When we try to create an instance we find that generating instances for our problem is insolvable, the cause being that there are not enough elements in the range of `given_3` for any value of `given_2` for the function to be injective. Using the reduction process from Chapter 5 one of the possible reductions is to remove the `injective` attribute from `given_3`. When we apply the instance generation process to the new specification we generate an instance such as Figure 6.6.

Figure 6.6: A generated instance

```
letting given_1 be 1
letting given_2 be 5
letting given_3 be function( 1 --> 5, 2 --> 5 )
```

Once we have a specification and a generated instance we test them using the process from Section 6.2.4.

6.4 Automatic Reconfiguration of the Generation Process

To further improve our test case generation process we can combine it with our other contributions, namely Test Case Reduction (Chapter 7) and Test Case Generalisation (Section 7.6). The motivation for this is that we would like to find as many distinct errors as possible, so finding errors with the same root cause is not useful. This process creates a feedback loop where the generator passes test cases to the reducer, which feeds positive and negative examples to adjust the generation process. This is illustrated in Figure 6.7 where *Run* encapsulates the process of refining and solving a specification. The *Adjust* stage is the focus of this section where it is described in detail.

6.4.1 An overview of Test Case Reduction & Generalisation

Test case reduction is the process of transforming a specification into a simpler specification that still exhibits the same error by performing a series of transformations, which can produce progressively simpler specifications. The reduction process works on all Essence specifications including parameterised specifications.

Test case generalisation adapts parts of the reduction and generation process to provide more insight into the cause of the error. A notable feature is that the generalisation process is also looking for *passing* specifications since these specifications can give insight into when the error does *not* occur.

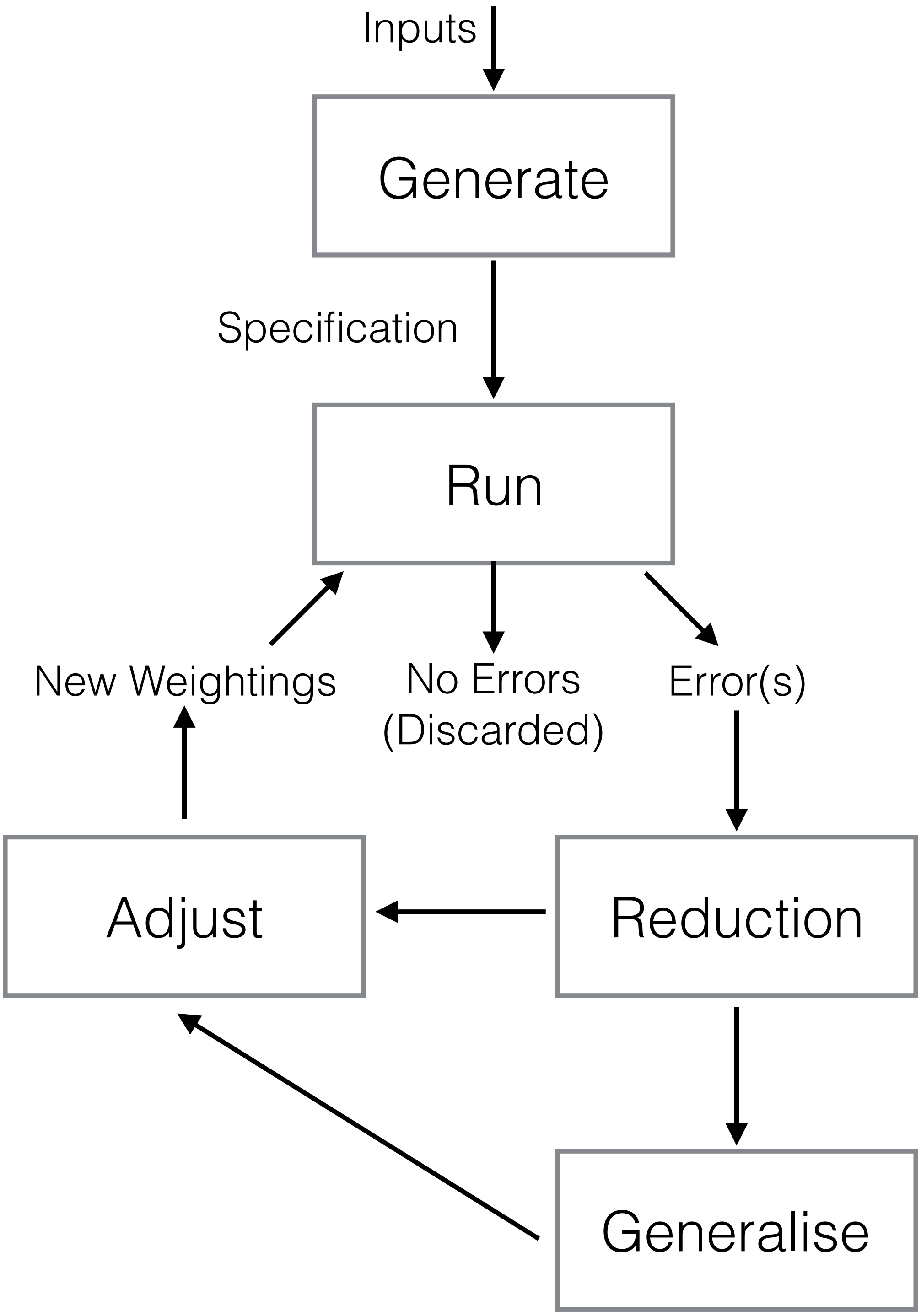


Figure 6.7: An overview of how the reduction and generalisation process can improve the generation process.

6.4.2 Utilising Reduction

Once we have generated a new specification S, we apply the reduction process on S to produce a reduced specification S_R. We then decrease the weighting of the parts that comprise S_R, since they are likely be the part of the cause of the error, hence increasing the likelihood of generating specifications with different errors.

To further guide the generation process we can utilise the previously generated test cases including specifications that did **not** cause any errors.

6.4.3 Worked example

We use the specification in the following diagram which was reduced from a more complex specification to illustrate how the reduction process can improve the diversity of the generated test cases.

Figure 6.8: A specification which produces an Essence′ model which causes an error

```
find var1: set (size 0) of bool
```

The main benefit of using reduced specifications is that we reduce the probability that we generate similar errors as compared to using the original specification, which could contain many irrelevant details. Utilising *just* the above specification (Figure 6.8) would result in decreasing the weighting of sized sets of booleans.

By including specifications generated previously, such as during the reduction process, we can further refine our process. Using the following specifications generated previously the system can deduce that the `size` attribute on sets is the cause of the error since the only difference between one of the non-errors and this error is the size attribute. Resulting from this the weighting of the `size` attribute on sets is decreased, hence meaning this particular error less likely to reoccur. Notable among the specifications found during the reduction process is Figure 6.10 which are *not* errors which demonstrates that passing test cases can be useful.

Figure 6.9: Errors (with the same cause) produced during reduction

```
find var1: set (size 0) of set of function int(2..3) --> int(0,3..5)
---
find var1: set (size 0) of int(1..4)
```

Figure 6.10: Non-errors produced during reduction and generalisation

```
find var1: set set of function int(2..3) --> int(0,3..5)
---
find var1: set of int(1..4)
---
find var1: set of bool
```

While we could keep all passing test cases found during the reduction process, most of these specifications do not provide much value on account of being very large relative to the fully reduced specifications. By employing the generalisation process from Section 7.6 we can find additional specifications which can help pinpoint what the root cause is without having to keep all the intermediate specifications. Additionally, these specifications are likely to be smaller, which helps narrow down the root cause. Notably the passing specifications found during reduction process can be found during the generalisation process of the reduced specifications, but as previously described these specifications are *not* rerun since the results are cached based on the hash of the tuple (Essence specification, Essence parameter).

Among the specifications produced by utilising the generalisation process are the following specifications which do *not* cause an error.

Figure 6.11: Non-errors

```
find var1: set (size 1) set of function int(2..3) --> int(0,3..5)
---
find var1: set (size 1) of int(1..4)
---
find var1: set (size 1) of bool
```

This strengthens the hypothesis that it is specifically the zero value on the `size` of sets that was causing errors. This results in deceasing the weighting of the `size` attribute on sets with a value of zero, reducing the likelihood of generating similar errors. While we correctly inferred the cause in this case, we may not always be successful. Since we never decrease any of the weighting to 0%, in case of under specifying the error condition (any size attribute on sets for example) then it still possible to generate sets with a `size` attribute of 0. Since we do not rerun specifications which are deemed to be identical, over specifying the error conditions (such as if the problem with the specific `size` attribute occurs with *all* domain types) does not impede the generation process, it just does not gives as much benefit as it could have.

6.4.4 Process

For each reduced specification r we produce a set of generalisation specifications g where each specification is represented as a tree (Section 4.1). To determine the expressions that should be adjusted, the sub-expressions that are present in specifications of g that do *not* cause an error are removed from r. The weightings of the remaining expressions are then decreased by 5%. Each expression has a lower bound of 1% for being picked to account for the possibility that we wrongly inferred the cause of the error. Additionally this allows the expression to be picked again in the unlikely case that it is part of a different error.

The main benefit of having a system of adjustable weights is that it reduces the likelihood of generating errors with the same root cause repeatedly hence allowing the generator to find more interesting test cases. This is particularly useful when dealing with errors such as when `toRelation` was parsed incorrectly. If a specification contains `toRelation` anywhere inside, which becomes more likely as the generated specifications become larger, the error is triggered. This can waste significant time during reduction (this example is described in Section 7.5.1) if `toRelation` is inside a highly nested expression since many intermediate steps would have to be tested. Decreasing the weighting of `toRelation` allows other errors to be found more quickly.

6.4.5 Other benefits

Resulting from the weighting system from the automatic reconfiguration of the generation process, we can use it manually to focus on a specific aspect of the ESSENCE language. Consider adding a new set representation; the generation process can be biased towards generating more sets. This is particularly useful to test how the new representations interact with other representations (i.e. channelling) as well as other features of the language.

Another useful feature, of the improved generation process is the generation of ESSENCE′ which is useful to test SAVILE ROW in isolation and also allows the same biasing of the generation process as described previously for the ESSENCE language

The weightings and results can be reused across different runs of the generation process. This is particularly useful for the reduction process since many intermediate reductions would have been tested previously. In addition the generation process allows a list of specifications to be excluded. This could be utilised so that known test cases are not regenerated.

6.5 Case Studies

In this section we will give an overview of the different kinds of errors that were found using our generator. For some of the examples we will only show the *fragment* that caused the error and in Chapter 7 show how we reduced these larger specifications automatically to these smaller fragments presented in this section.

Type Checking: In Figure 6.12 `var1(_,_,_)` is a relation projection that returns `var1` unchanged. Since the arity of the relations differ this equality should have been rejected by the type checker. The root cause of this bug was that the relations were `zipped` together when type checking hence the last element of the longer relation (`var1`) was mistakenly discarded.

Figure 6.12: Example where the specification should be rejected by Conjure's type checker

```
language Essence 1.3

find var1 relation(minSize 1) of (int(1..3) *  int(4..5) * int(8..9))
find var2: relation(minSize 1) of (int(4..5) * int(8..9))

such that
    var2 = var1(_,_,_)
```

Language Differences: As described previously (page 59) Essence and Essence′ have slightly different semantics. During the refinement of Figure 6.13 the `toInt` method is removed because booleans can be treated like integers in Essence′. The problem occurs during refinement of the set literal on right hand side of line 4.

Figure 6.13: toInt removed too early

```
1 language Essence 1.3
2
3 find var1: bool
4 such that {toInt(false), 5} != {toInt(false), 5 / -1}
5 such that 7 / toInt(true) != 6 / (8 / -9)
```

This line was refined to `[false, 5 / -1; int(1..2)]`. The problem is that the matrix is an invalid refinement since it is not homogeneous. The solution to this problem is to only remove the `toInt` at the end of refinement.

Now consider `var1[2, 1]` in Figure 6.14, in ESSENCE this is a valid statement, whereas in ESSENCE′ this would have to be written as `var1[2, 1,..]` making the slicing explicit. This edge case was reduced from a larger specification with the process described in Section 7.5.2.

Figure 6.14: Indexing differences

```
language Essence 1.3

find var1: matrix indexed by [int(5), int(1), int(3)] of int(0)
such that and([allDiff(var1[2, 1]); int(1..1)])
```

Partial Evaluation: Figure 6.15 shows an oversight in the partial evaluator of equality of set literals, which did not take account of duplicates, it evaluated line 5 to `false` instead of `true`.

Figure 6.15: Evaluating set equality

```
1 language Essence 1.3
2
3 find var1: set of bool
4 such that
5     {true} = {true, true} \/ false
```

Function inequality: Listing 6.16 shows a refinement error, in this case the refinement of inequality of functions with nested types resulting from missing refinement rules.

Figure 6.16: Bug in the refinement of equality of functions

```
language Essence 1.3

find var1: function bool --> matrix indexed by [int(-9..-8, 5..5)] of bool
such that
    var1 != var1
```

Inconsistencies: Figure 6.17 should have the solution `[function(), function(), function (), function()]`. Of the three ESSENCE′ models refined, two produced the correct answer and other resulted in an unsolvable model.

Figure 6.17: Produces inconsistent models

```
language Essence 1.3
find var0:
        matrix indexed by [int(1..4)] of function (maxSize 0, injective)
                                             int(2..2) --> int(3..4)
```

Parsing: Figure 6.18 is an error caused by invalid parsing of `toRelation`. This was reduced automatically from a larger specification and will be discussed in detail in Section 7.5.1.

Figure 6.18: `toRelation` parsed incorrectly

```
language Essence 1.3

find unused: bool
such that relation((5, true)) = toRelation(function(10 --> true))
```

A similar error was found with the parsing of `>lex` and `inverse`.

Edge Cases: The generator discovered edges cases that were previously not well defined:

- Placing decision variables in the `factorial` function which is now disallowed.
- Negative exponents were not handled consistently in Conjure and Savile Row. This resulted from Conjure and Savile Row only operating on integers. This was fixed by defining the sub-expression with any negative exponents to be undefined (even for the small number of cases where a negative exponent would give an integer).
- Deciding if `[]` should be allowed as input to a n-dimensional matrix such as this `given` statement with domain `matrix indexed by [int(1..4), int(1..3)] of bool`. This was then defined so that empty matrixes are cast to the right dimensions.

Simplifications: Figure 6.19 caused a stack overflow in Savile Row because `var1` was unified with the *other* `var1` hence looping forever. Running the generalisation process (Section 7.6) showed the error does not occur if the `false` on the right side of the `<->` on line 5 is changed to `true`. The fix was to improve the unification process to avoid this.

Figure 6.19: Stack overflow because of invalid unification

```
1 language Essence 1.3
2
3 find var1: bool
4 such that
5     ((var1 <-> var1) -> false) /\ true
```

A similar error was also found for the equality operator `var5 = (var5 = false)` after the above error was fixed.

Solution Translation: At the Essence′ level consider Figure 6.20 which is one of the Essence′ models produced after refinement of the Essence specification. The bug caused Savile Row to not produce the solution despite outputting that the model was solvable and a solution was found. The error was in the handling of the empty interval in a matrix's bounds, since at the Minion level there were no variables.

Figure 6.20: No Solution shown (Essence′)

```
language ESSENCE' 1.0

find var1: matrix indexed by [int(1..0)] of bool
such that true
```

After this particular error was fixed Figure 6.21 with a similar error was found, caused by an incomplete bug-fix.

Figure 6.21: No solution shown 2nd case (Essence′)

```
language ESSENCE' 1.0

find var3_ExplicitVarSizeWithMarker_Values_Explicit_Explicit:
        matrix indexed by [int(1..1), int(1..0)] of bool
such that true
```

The generation process was also used to test the solution translation process from Essence′ to Essence. The following specification caused an error in the translation of `var1` when using the `Function1DPartial` representation. This resulted in the solution `[function(); int(3..4)]` instead of `[function(), function(); int(3..4)]`

Figure 6.22: Specification triggering an invalid solution translation

```
language Essence 1.3

find var1: matrix indexed by [int(3..4)] of function int(5..5) --> int(2..3)
```

Parameter Refinement Figure 6.23 is an example of a parameterised test case that caused an error resulting from Conjure not calculating the domain of the `given` correctly.

Figure 6.23: Refining Partitions

```
language Essence 1.3

given given1: partition (minPartSize 0) from int(2..5, 4)
find unused: bool
---
$ param
letting given1 be partition({2, 3, 5}, {4})
```

Figure 6.24 is also an error caused by an incorrect domain size calculation. The maximum number of elements in the `mset` is 16. While the `value` of `given2` was correctly refined into 16 element matrix, the domain was not. The domain was refined into a 20 element matrix resulting from not handing duplicates in the domain `int(3, 1..4)`, where `3` is repeated twice.

Figure 6.24: Invalid refinement of `mset` parameter

```
language Essence 1.3

find unused: bool
given given2: mset (maxOccur 4) of int(3, 1..4)
---
letting given2 be mset(1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 4)
```

It was also able to find various bugs in the handling of very nested `given` statement, including calculating the size of the domain and handling of attributes on these domains. Since the generation process also produced instances to these highly nested `given` statements automatically, it made it easier to find the root cause.

6.5.1 Visualisation of Test Cases Found

In this section we discuss aggregated data that was automatically collected when the generated test cases when run during the development process of the toolchain. This data included the date and the versions of the tools in the toolchain where the error was found and when it was fixed. The data was obtained from 2.1GHz machines with an AMD-based architecture.

Figure 6.25 shows when failing tests were found and fixed. Over time the system matures, hence it is naturally more difficult to find new errors, making the automatic reconfiguration of the generation process more useful.

Figure 6.26 shows the constraint and domain depth of each failing test case as described in Section 6.2.1. The reason that there is a high proportion of specifications have a maximum constraint depth of 0 is that these specifications contain domains that could not be refined hence causing an error. Figure 6.27 gives an example of one of these specifications, where the error is the missing refinement rule of matrixes of partition of bool resulting from refinement of `var3`.

Figure 6.27: Constraint depth 0 – domain depth 2

```
language Essence 1.3

find var3: function bool --> partition from bool
such that true
```

It may seem odd that there was were fewer errors detected when increasing the constraint depth, this results from these specifications either timing out before an error occurred or running out of memory.

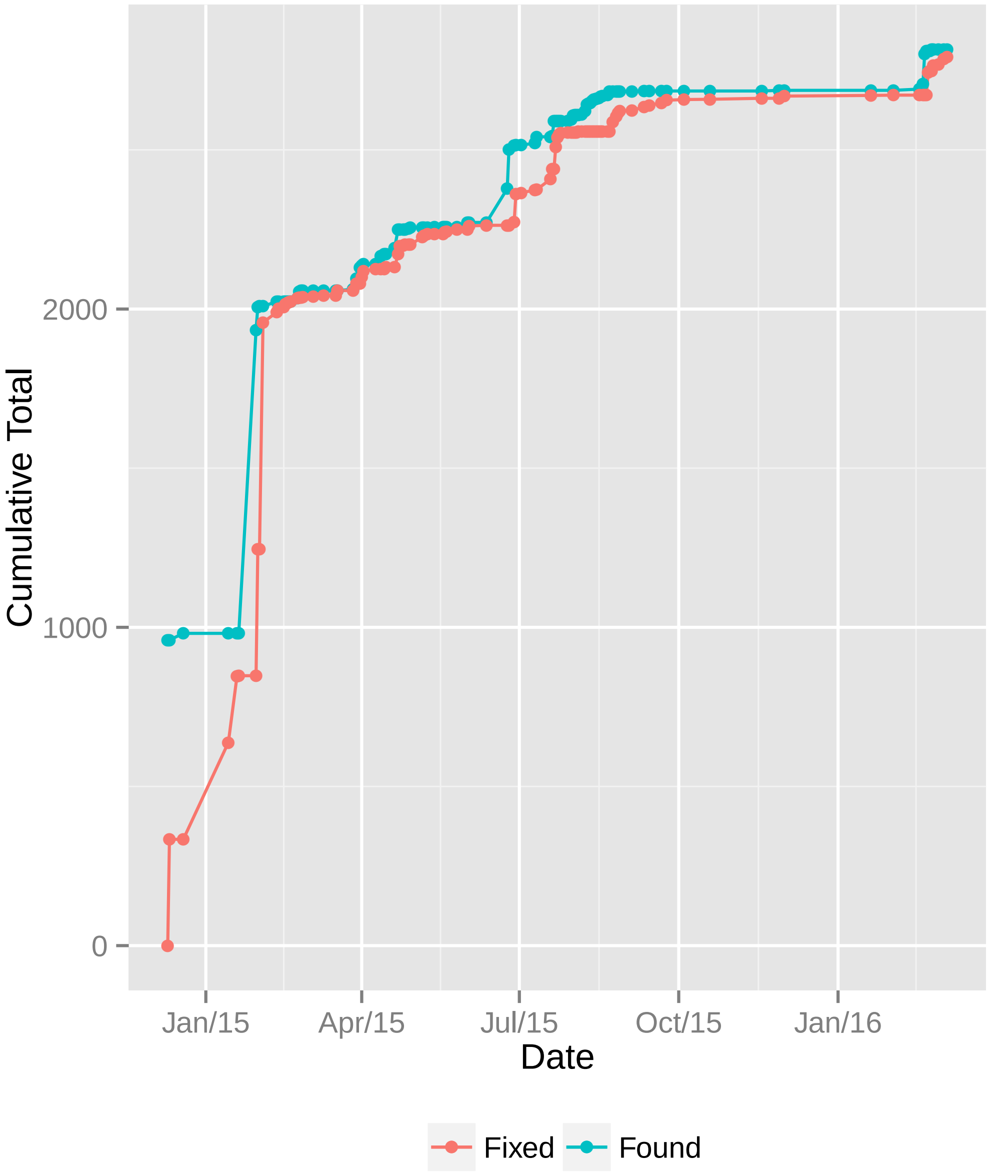


Figure 6.25: Failing test cases over time

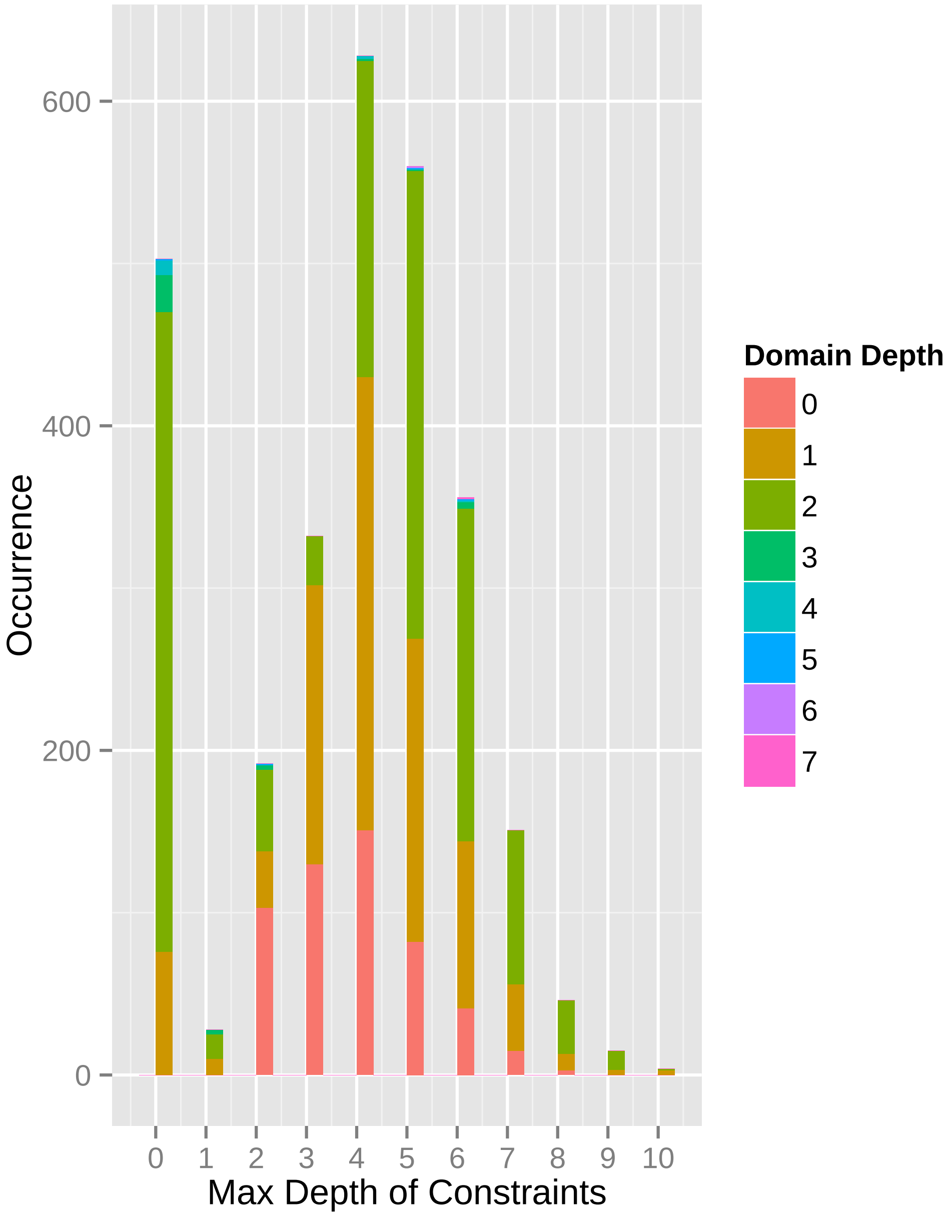


Figure 6.26: Error classified by maximum constraint depth and domain depth

6.6 Summary

This chapter discussed how to generate test cases automatically, for the purposes of validating a toolchain for automated modelling. We described how the various parts of the test cases are generated, with full coverage of the Essence language. Additionally by utilising our instance generation process we showed how we could generate parameterised specifications significantly enhancing the test cases that we can generate. Finally we described how we can automatically adjust the generation process automatically to increase the probability of finding new errors.

Chapter 7

Test Case Reduction

Since the specifications produced by the generator can be quite large, test case reduction is a way of simplifying a specification that is the cause of an error to a simpler specification which has a similar error. This makes it easier for the developer to understand and fix the error. Another use case is if the user finds a specification that causes an error, a developer can then run the reducer to gain insight into the cause of the problem.

The test case reducer works by performing a series of transformations, which can produce progressively simpler specifications. These steps are repeated until a fixed point is reached and no further reductions are possible or the given time budget is exceeded. When selecting the next specification to run from a set of candidate reductions using the simplifications described in Section 7.2, we select the specification that is *simpler*[1], backtracking only when a candidate reduction does not exhibit the error.

After applying a simplification, the candidate specification is run to check if it still produces a similar error[2]. Although we have a log of the rules applied to generate the model, the simplification may have made those rules inapplicable. There are two ways of overcoming this when running the candidate specification:

- Refine all models of the candidate specification. This will find an error if it exists, but is only feasible for small specifications.

[1] The simpler comparison is defined for each kind of domain and constraint and has three values: *less then*, *greater then* and *equal*. The comparison firstly compares the depth of the expressions then uses a tie breaker specific to the expressions being compared. To combine the results of simpler comparisons (e.g. on a list) we select the dominant result.

[2] An error is similar if the cause is in the same place in the toolchain. e.g. in the same Java method.

- Apply the rules from the logs as far as possible then refine the selection of models using the available rules at that point.

The reduction process is greedy since it only considers the reductions of the 'simplest' specification that has an error that is classified in the same way as the original.

7.1 Log Following

Since refining all models is not, in general, feasible we need a way of generating a similar Essence$'$ model that still causes the original error.

Since Conjure's refinement process works by applying a series of transformation rules, at any point there can be multiple transformation rules that can be applied. An Essence$'$ model is then the result of a series of successive choices.

Just keeping the index of each of choice that was made is not sufficient since the choices may change depending on the transformations applied. As an example consider a function which was inferred to be total, if we remove a constraint in the process of reduction then the function would no longer be total, hence having different transformation rules available.

For each rule used we store the expression that was transformed, the rule's identifier, and the *context* that the rule was applied in. The context is necessary since the same expression can occur multiple times in the specification, each with a different transformation applied e.g. sets as either the `occurrence` or the `explicit` representation. Additionally for *domain representation rules* we store the domain before it was transformed.

When refining a reduced specification, we utilise the stored choices(S) as follows. If there is a *matching* rule then we choose that rule. If there is no *matching* rule (which could happen if a rule was removed, or extra restrictions were placed upon that rule) we produce a separate model for each applicable rule, this ensures the model we wish to find is not missed.

Each stored rule can only be matched once, if a rule *was* in fact applied multiple times then it would have to be stored multiple times in the stored choices.

At each point in the reduced specification we have a list of choices C we rank each choice C_i by how well it matches a stored rule. For each C_i we pick the stored rule that is most similar.

We match C_i with a choice from the stored choices S using the following criteria which starts with the most important criterion.

1. Rule name.

2. Partial matching of the context.

3. Domain matching.

4. We prefer rules that were applied earlier as the final tiebreaker.

When comparing domains we say two domains match if all the representations are the same. Using the notation `{representation}`, `set {explicit} of set (maxSize 3) {occurrence} of int(10)` would match `set {explicit} of set {occurrence} of int(1)`.

Consider Figure 7.1 which contains a specification that caused an error due to the missing refinement rule for the `hist` function, and a candidate reduction. We use this example to illustrate how the process works.

Figure 7.1: A specification and a candidate reduction

```
find var1: mset (minSize 2, minOccur 1 , maxOccur 5) of bool
such that allDiff(hist(var1))
---
find var1: mset (maxOccur 5) of bool
such that allDiff(hist(var1))
```

The first choice that we have to make is the representation of the variable `var1`, we have two choices:

```
1) mset {ExplicitWithFlags} (maxOccur 5) of bool
2) mset {ExplicitWithRepetition} (maxOccur 5) of bool
```

The *context* at this point specifies that `var1` is the expression that will be transformed. Additionally it contains the locations where the expression is referenced, in our case only `allDiff (hist(var1))`. The context matches the context used in the original specification since the constraints are unchanged. For the original specification we chose `mset {ExplicitWithFlags} (minSize 2, minOccur 1, maxOccur 5)of bool` which using our domain matching process matches the first choice.

The log following process is also useful for revalidation purposes and regression testing since rules may be added or removed during development. Log following saves the developer from having to refine all models to ensure the error can be reproduced.

Log following is utilised in the test generation process for similar reasons, when a specification or parameter can not be refined within the given timeout. Log following allows the generation process to produce a similar model from the reduced specification of the original without having to refine all models.

7.2 Simplification Process

The reduction process starts off with coarse-grained simplifications that try to remove large portions of the specification. After these simplifications have been performed we focus on fine-grained simplifications that involve more sophisticated reasoning such as simplifications that change the type of an expression. The following simplifications are run until a fixed-point has been reached or the given time budget has been exceeded.

As discussed previously a reduction is successful if the resulting specification has a *similar* error. If a reduction produces a *different* error, it is stored for later processing.

1. *removeObjective*: Converts an optimisation problem to a satisfaction problem by removing the objective. If this is successful it means we can infer that the cause of the error was not in the objective. Otherwise it strengthens the argument that the cause was in-fact the objective function.

2. *removeUnusedDomains*: Removing unused domains (which are domains that are not referenced from any other sub-expression) is a cheap way to simplify a specification. In the unlikely case where removing an unused domain causes the error to disappear, we try the removed domain in isolation to see if that domain was the root cause. The major use of this simplification is after a constraint has been reduced which can cause a domain to have no references left, hence unused and a candidate for removal.

3. *removeConstraints*: If we have multiple constraints, we try removing different subsets of them to see if some of the constraints are redundant. The subsets that are chosen

alternate between removing all but one of the remaining constraints and removing a single constraint.

4. *inlineGivens*: Inlining parameters, which is the process of replacing references to a parameter with its *value*, has two main benefits. Firstly it simplifies the specification, since it makes the specification more self contained. The other benefit is that it makes the specification more amenable to further simplification, since if we achieve the goal of inlining *all* parameters, we can skip the generation of an instance when reducing a parameter's domain. This step may be necessary since the current instance may be invalidated by the reduction of the domain of a parameter.

5. *simplifyExpression*: The most involved part of the reduction process. When this process is applied to a domain, additional simplifications are applied such as relaxing the attributes on a domain. This process also applies to the objective function with an additional restriction that a decision variable must still be referenced. This simplification is explained in more detail in Section 7.2.1

6. *reduceRefinement*: If we reach a fixed point when performing the simplifications at the Essence level, we consider simplifications at the Essence$'$ level for non-refinement errors i.e a valid Essence$'$ model was successfully created. This can result in a much simpler test case when the error is in the later stages of the toolchain, since structural and symmetry constraints added from the use of abstract domain and their operators can be candidates for removal. This simplification utilises all the previous simplifications and is described in Section 7.4

7.2.1 Simplification of an Expression

Depending on the kind of expression we can apply a variety of different simplifications. We start by describing the more general simplifications and then follow on to simplifications that only apply to specific kinds of expressions such as comprehensions.

Expression replacement: We replace the whole expression with the simplest expressions of the same type. For example if we have a conjunction of boolean expressions we could replace it with `true`, for sets we use a set containing a single element in addition to the empty set. At each iteration of this process a different literal is chosen: for the conjunction of booleans example `false` would be chosen on the second iteration, for sets a different element from the set is chosen for the single element set each time. If we run out of replacements to try this part of the process is skipped.

Sub-term reduction: We try combinations of reductions on the sub-terms of the expression. Sub-terms are defined for each type of expression. For containers such as a set, the sub-terms are the elements of the of the set. This process is applied recursively.

Sub-term promotion: If the sub-terms have the same type as the expression (such as arithmetic operators) we replace the expression with the sub-term. Consider the expression `((var1 <-> var1) -> false) /\ true`, there are two candidate reductions namely `true` and `((var1 <-> var1) -> false)`. The second candidate has two further reductions.

Size reduction: We try to decrease the 'size' of the expression, in the case of literals such as sets, functions and relation we try different subsets of the elements, mapping and tuples respectively.

Attribute relaxation: There are two main attributes we will consider, the first are unary constraints on a domain such as `bijective` on functions, the second kind take an expression as an argument such as `size 5`. We firstly try to remove a subset of these attributes in a manner that is similar to the removal of constraints discussed previously, but with additional restrictions. In the case of a `mset` we must ensure that it still has a finite domain by requiring one of `maxSize`, `size` and `maxOccur` to be present even after a simplification. Examples of reducing an unary attribute include changing `bijective` to `injective`. Of the second kind of attribute, containing an expression, we just apply our reduction process for an expression upon it.

Type changes: Depending on how a variable is referenced we can simplify its domain. In Figure 7.2 we can convert `var1` to just its second element because the first element of the tuple is never referenced.

Figure 7.2: Example of a domain simplification

```
find var1 : (bool, set of int)
find var2 : int
such that |var1[2]| = var2
-->
find var1 : set of int
find var2 : int
such that |var1| = var2
```

Literal specific reductions: When reducing function literals we treat the mappings as a list of pairs. We perform the reduction process on each mapping to get a list of candidate reductions for each element of the pair. We then try combinations of the pairs where certain pairs are reduced and others are *fixed* with their original values. We start with having all pairs reduced and gradually revert the values back to their original values if the error disappears. While this simplification leads to many candidate specifications to test, note that this simplification is applied after trying to reduce the size of the function meaning that the function is likely to have few elements.

As an example we will consider the reduction of the nested function in Figure 7.3.

Figure 7.3: Function specific reductions

```
function( function() --> (true,var2)
        , function(false /\ var1 --> false) --> (false,1+var3)
        )
```

For the first mapping `(function(), (true,var2))` there is one possible reduction namely `( function(), (true,false))`. For the second mapping `(function(false /\ var1 --> false) , (false,1+var3))`, the first element has the reductions `function(var1 --> false)`, `function (false --> false)` and `function(false /\ true --> false)`, the second element has the reductions `(false,1)`, `(false,var3)` and `(false,1+1)`. The reductions of the second mapping are then the product of the reductions of each element of the pair. Figure 7.4 shows how we use the reductions of the mapping starting with all mappings being reduced, then trying various combinations of these reductions.

Figure 7.4: Reductions tested

```
$ All mappings reduced
function( function() --> (true,false)
        , function(false --> false) --> (false,1)
        )
```

```


$ Reducing only the first mapping
function( function() --> (true,false)
        , function(false /\ var1 --> false) --> (false,1+var3)
        )

$ Reducing only the second mapping
function( function() --> (true,var2)
        , function(false --> false) --> (false,1)
        )

$ Reducing only the the first element of the second mapping
function( function() --> (true,var2)
        , function(false --> false) --> (false,1+var3)
        )
```

For partitions note that we have already tried to decrease the number of parts in the partition when performing the size reduction. We firstly try to decrease the number of elements *inside* each part of the partition. Following this, we apply a process similar to the reduction of function literal, with a variable sized set instead of a pair which generally leads to many more specifications to test.

Comprehension-specific simplifications: The first stage is to remove the `conditions` of the generator `a > 2` and `a < 4` in Figure 7.5 using the *removeConstraints* simplifications previously described.

The second stage is to remove unused generators. A generator is unused if there are no references in the other parts of the comprehension. Since the reduction as a whole is applied until a fixed point is reached, this would catch the case where a series of the generators are unused but reference other generators as is the case in Figure 7.5 where `c` would be removed in the first iteration and the `b` on the second.

Figure 7.5: Removing unused generators

```
[ a | a: int(1..4), b: int(0..[2; int(1..1)][a], c: int(1..b), a >2, a <4]
```

To see if the used generators (`a` in Figure 7.5) are necessary we would like to instantiate them. The main problem with this is that the domains may be very large hence leading to many candidate reductions. To overcome this we first apply the domain reduction process on the generators which usually results in much smaller domains. If the comprehension's conditions in Figure 7.5 were redundant, after instantiating `a` with one of it values, the result would be a one element matrix namely `[4]`.

We then apply the reduction process to the inner expression. If there are any conditions left in the comprehension we try to reduce subsets of these conditions starting with all the conditions being reduced, generally reverting the conditions to their original values if the error disappears. Since this reduction process is repeated until a fixed point is reached, generators and conditions that become unused, because of simplifications applied the to inner expression would be removed in future iterations of the reduction process.

This process also applies to quantified expressions, since we convert these expressions to comprehensions for simplicity and to reuse our reduction methods for comprehensions.

7.3 Parameterised Reduction

Parameterised specifications add another layer of complexity to the reduction process. If we don't have an instance for the specification, we generate it using the process described in Chapter 5[3]. Once we have a parameterised specification with its associated instance, we apply the reduction process. In the simplest case when we remove an unused parameter from the specification, we just need to adjust the instance by removing the (unused) parameter from it.

When performing a reduction upon a domain of a parameter we must *validate* the instance to see it is still valid. To show why this is necessary consider Figure 7.6 which replaces the expression inside the `maxOccur` attribute with a constant.

Figure 7.6: Domain reduction

```
given N, M : int(1..10)
given V : mset(maxOccur N * M)  of int(1..5)
-->
given V : mset(maxOccur 1)  of int(1..5)
```

If `V` was the multi-set `mset(1,1)` the instance would now be invalid. To generate instances in the general case we use our instance generation method from Chapter 5. We further use the instance generation process in the next section, to investigate if cause of the error was a particular `value` assigned to a parameter of the instance.

[3] In particular it only uses the generation part i.e. no racing is performed.

7.3.1 Kinds of Parameterised Errors

In the presence of `given` statements there can two kinds of errors namely with the *domain* of the parameter or the *value* of a particular parameter.

An example of an error associated with the *domain* of the parameter is the unusual combinations of attributes (enforcing totality in addition to specifying the number of mappings) on a function shown in Figure 7.7 which was a refinement error. The error was that the refinement rule for total functions did not take into account of the `maxSize` attribute resulting in assignments which were not valid.

Figure 7.7: Error associated with the domain of the parameter

```
given n : int(1..5)
given values: function (total, maxSize n) int(1..5) --> int(1..9)
```

On the other hand errors associated with *values* of the domain, includes edge cases such as the handling of empty matrixes resulting from undefinedness. In this case the domain itself is irrelevant as long as it admits the empty matrix. To detect these differences, we try to generate new values (using our instance generation process from Chapter 5). If the new values do *not* cause the same error then we have some evidence that it is the *values* themselves which are causing the error.

7.4 Reducing a Refined Model

Consider Figure 7.8, a fully reduced specification at the Essence level:

Figure 7.8: Fully reduced at the Essence level

```
language Essence 1.3

find unused: bool
given given2: mset (minSize 3, maxOccur 4) of set (size 0) of bool
---
$ Param
letting given2 be mset({}, {}, {})
```

Since the error occurs when the refinement (Figure 7.10) is given to Savile Row, we consider reductions at the Essence′ level as described in Figure 7.9 with the aim of producing a simpler specification.

After ensuring that we don't have a refinement error (since this would not have produced an Essence′ model) we select the *simplest* (using our metric) of the refinements of the Essence

```
Algorithm 3: reduceReductions($R, E$)
    Input: A set $R$ of pairs (refinement, refined instance) from the Essence specification,
           error classification $E$ for determining if the reduced error is similar to the
           original specification[a].
    Output: A reduction of one of the Essence$'$s of $R$ if possible
    if refineError $E$ then
        return null
    end
    $r \leftarrow$ simplestRefinement $R$
    $(r', p') \leftarrow$ toEssence $r$
    $res \leftarrow$ runCompactAndSolve r' p'
    if similarError $E$ $res$ then
        $reduced \leftarrow$ reduce E $r'p'$
        return $reduced$
    else
        return null
    end
```

[a] An error is similar if the error occurs in the same tool in the toolchain, and is classified (e.g missing refinement rule or stack overflow

Figure 7.9: The process of reducing the refinements.

specifications. We then convert the Essence$'$ model and instance back to the Essence level, which involves introducing stricter types since one of the main differences between Essence and Essence$'$ is that Booleans are not implicitly converted to integers. To validate that the new Essence specification r' has a similar error in the same way as the original specification we run r' using the compact heuristic which is sufficient because we should not need to make any decisions to reach a new Essence$'$ model[4]. We apply a similar process for the new Essence instance p'. We then run our reduction process of the new specification and instance and return the results. Reducing Figure 7.10 yields modest gains, only removing `given2_ExplicitWithFlagsR3_Flags`. In our case studies (Section 7.5.3) we will show how this process can be applied to more complex specifications to yield significant gains.

[4] Using the log following process (Section 7.1) would not help in this case as it keeps a log of the decisions made to convert the Essence specification to Essence$'$.

Figure 7.10: A refinement of Figure 7.8

```
language Essence 1.3
find unused: bool
given given2_ExplicitWithFlagsR3_Flags: matrix indexed by [int(1..4)] of int(0..4)
given given2_ExplicitWithFlagsR3_Values_Explicit:
        matrix indexed by [int(1..4), int(1..0)] of bool
---
$ param
letting given2_ExplicitWithFlagsR3_Flags be  [3, 0, 0, 0; int(1..4)]
letting given2_ExplicitWithFlagsR3_Values_Explicit, [[], [], [], []; int(1..4)]
```

7.5 Case Studies

In this section we discuss some notable features through worked examples of the reduction process. We use both errors that were created in the generation process in addition to errors found by users.

7.5.1 Sub-term Promotions and Type Changing Reductions

Consider Figure 7.11, an error that has a simple explanation (`toRelation` was incorrectly parsed) but when reduced demonstrates how reasoning about the types used can lead to significant gains.

Figure 7.11: Starting specification

```
language Essence 1.3

find var1: bool
such that true
such that
    ([var1; int(-2)], relation((2, false), (-8, false), (-3, false)))
    =
    ([true /\ false, var1; int(-4, 5)], toRelation(function(10 --> true)))
```

In Figure 7.12, we firstly remove the redundant constraint (`such that true`). We then replace `var1` with a constant, since it is now unused. Following this we replace the relation in the left hand side of the equality with a simpler relation containing only one tuple.

Figure 7.12: Replacement by a constant

```
language Essence 1.3

find unused: bool
such that
    ([true; int(1..1)], relation((5, true)))
    ([true /\ false; int(1..1)], toRelation(function(10 --> true)))
```

We then perform sub-term promotion on the conjunction from the right hand side of the equality replacing `true /\ false` with `false`, as shown in Figure 7.13.

Figure 7.13: Sub-term promotion

```
language Essence 1.3

find unused: bool
such that
    ([true; int(1..1)], relation((5, true))) =
    ([false; int(1..1)], toRelation(function(10 --> true)))
```

Since we have an equality between comparable types of expressions, a tuple in our case, a valid reduction would be to change the expression to an equality on a member of each tuple as shown in Figure 7.14, this causes the same error as the original specification. This changes the type locally, on each side of the equality but the overall type of the expression, namely a Boolean, remains the same. Similar reasoning is utilised when reducing other operations in Essence.

Figure 7.14: Type changes

```
language Essence 1.3

find unused: bool
such that relation((5, true)) = toRelation(function(10 --> true))
```

7.5.2 Domain Simplifications

Consider Figure 7.15, a specification that will be used to demonstrate how we convert quantified expressions into comprehensions in addition to showing how to simplify domains. As discussed previously in Section 6.5 the error is in the handling of matrix indexing, in Essence `var1[2, 1]` is a valid statement, whereas in Essence′ this would have to be written as `var1[2, 1,..]` making the slicing explicit.

Figure 7.15: Starting specification

```
language Essence 1.3

find var1 : matrix indexed by [int(2,4),int(1,3,6..9),int(1..3,7..8,9)]
  of int(0..20)

such that
    forAll i : int(1,3,7..8,9) .
        allDiff(var1[2,i]) /\ allDiff(var1[4,i])
```

The first simplification, reducing `allDiff(var1[2,i]) /\ allDiff(var1[4,i])` to `allDiff (var1[2,i])`, is an example of sub-term promotion as discussed previously.

For integers there are three main simplifications. The first reduces the number of ranges inside the integer, simplifying `int(1,3,6..9)` to `int(1)`. Note that the resulting domain is a subset of the original. The second simplifications replaces a range with a constant, simplifying `int(2,4)` to `int(5)`, note that the new domain is *not* necessarily a subset of the original. Since a domain can contain an arbitrary expressions[5], the third simplification applies the reduction process on each expression inside the domain recursively. The resulting expression is shown in Figure 7.16.

Figure 7.16: Applying domain simplifications

```
language Essence 1.3

find var1: matrix indexed by [int(5), int(1), int(3)] of int(0)
such that and([allDiff(var1[2, i]) | i : int(1)])
```

We can further refine Figure 7.16 by inlining `i` in the comprehension. This results in Figure 7.17

Figure 7.17: Instantiating the generator `i`

```
language Essence 1.3

find var1: matrix indexed by [int(5), int(1), int(3)] of int(0)
such that and([allDiff(var1[2, 1]); int(1..1)])
```

The previous reduction allows us to further simplify the specification, resulting from the generators of the comprehension having been removed. Since the return type of the `allDiff` function is a Boolean, we can promote it to the top level resulting in Figure 7.18. As compared to the original specification the final reduction contains no quantified expressions, a simpler domain for the decision variable and a single reference to it.

Figure 7.18: Promoting the `allDiff` function

```
language Essence 1.3

find var1: matrix indexed by [int(5), int(1), int(3)] of int(0)
such that allDiff(var1[2, 1])
```

7.5.3 From Users

The specification we consider (Figure 7.19) was found by an expert user, and reduced manually from a much larger specification. As we will show in this section, this error (Savile Row

[5] with the restriction that a decision variable's domain can not depend on another decision variable.

producing invalid Minion because of undefinedness) can be reduced much further resulting in Figure 7.20.

Notably we can also generate instance data, to fully automate the reduction of the specification. This is useful for several reasons such as if no instance data was given or the instance data is too large.

Figure 7.19: The manually reduced specification that we will consider

```
language ESSENCE' 1.0

given module_EnumSize: int(1..10)
given fin4: int(1..10)
given fin5: int(1..10)
given fin6: int(1..10)
given var2: matrix indexed by [int(1..module_EnumSize), int(1..fin4)]
            of int(fin5..fin6)
find b : bool
such that b = and([true | m1 : int(1..module_EnumSize)
                        , q5 : int(1..fin4)
                        , t1 : int(0..var2[m1, q5] - 1)
                        ])
```

Figure 7.20: Reduction of Figure 7.19

```
language Essence 1.3

find unused: bool
such that and([true | q5 : int(1..2), t1 : int(0..[2; int(1..1)][q5])])
```

To consider whether we are missing possible reductions we tried reducing the original specification from which Figure 7.19 was derived. This resulted in Figure 7.21 which similar in simplicity to the reduction of the user reduced specification in Figure 7.20.

Figure 7.21: Reduction of the original specification which Figure 7.19 was derived from

```
language Essence 1.3

find unused: bool
such that and([true | q2 : int(1..3), t1 : int(0..[0; int(1..1)][q2])])
```

Figure 7.22 show the original specification after the first iteration of unused domains (no effect) and remove constraints (removing 3 very nested constraints) was performed[6]. We will summarise the process that led to the final reduced specification (i.e. Figure 7.21) by noting *only* reductions that reduced the candidate specification. The reduction process for simplifying the manually reduced specification (Figure 7.19) to its final form (Figure 7.20) is similar to

[6] The letting statement have been inlined by this stage. In addition the syntactic sugar for function projection was replaced by `image`, e.g. `moduleAlloc(m1)--> image(moduleAlloc,m1)`

reduction process of the original specification described below with some steps omitted since they were unnecessary.

Figure 7.22: The specification that Figure 7.19 was derived from after one iteration of remove constraints.

```
language Essence 1.3

given student_EnumSize: int
given lecturer_EnumSize: int
given module_EnumSize: int
given room_EnumSize: int
given student_module: matrix indexed by [int(1..student_EnumSize)] of
    set of int(1..module_EnumSize)
given lecturer_module: matrix indexed by [int(1..lecturer_EnumSize)] of
    set of int(1..module_EnumSize)
given module_suitableRooms: matrix indexed by [int(1..module_EnumSize)] of
    set of int(1..room_EnumSize)
given module_blocks:
        matrix indexed by [int(1..3)] of sequence of int
find moduleAlloc: function (total) int(1..module_EnumSize)
  --> function int(1..4) --> record {mRoom : int(1..room_EnumSize),
                                     mDay : int(1..5),
                                     mTime : int(9..17)}
such that
    and([(image(image(moduleAlloc, m1), b1)[mRoom],
          image(image(moduleAlloc, m1), b1)[mDay],
          image(image(moduleAlloc, m1), b1)[mTime] + t1)
         !=
         (image(image(moduleAlloc, m2), b2)[mRoom],
          image(image(moduleAlloc, m2), b2)[mDay],
          image(image(moduleAlloc, m2), b2)[mTime] + t2)
             | m1 : int(1..module_EnumSize),
               m2 : int(1..module_EnumSize),
               (b1, b1Len) <- module_blocks[m1],
               (b2, b2Len) <- module_blocks[m2],
               t1 : int(0..b1Len - 1), t2 : int(0..b2Len - 1),
               (m1, b1) ~< (m2, b2), (m1, b1) ~< (m2, b2)])
```

Figure 7.23: The initial instance for Figure 7.22

```
letting student_EnumSize be 3
letting lecturer_EnumSize be 2
letting module_EnumSize be 3
letting room_EnumSize be 4
letting module_suitableRooms be [{1, 2}, {2, 3}, {1, 3, 4}; int(1..3)]
letting module_blocks be [sequence(2, 2), sequence(1, 1, 1), sequence(3); int(1..3)]
letting lecturer_module be [{1}, {2}; int(1..2)]
```

The next simplification that was performed is the removal of the `given` statements that are now unused resulting in Figure 7.24. This simplification was able to vastly decrease the size of the specification, requiring only a single parameter to trigger the error.

The reduction process on the find statements yields modest gains namely removing the totality restriction. The decrease of the size of the outer function's domain a byproduct. The

Figure 7.24: Removing the given statements that are now unused.

```
language Essence 1.3

given module_blocks:
        matrix indexed by [int(1..3)] of sequence of int
find moduleAlloc: function (total) int(1..module_EnumSize)
  --> function int(1..4) --> record {mRoom : int(1..room_EnumSize),
                                     mDay : int(1..5),
                                     mTime : int(9..17)}

such that
    and([(image(image(moduleAlloc, m1), b1)[mRoom],
          image(image(moduleAlloc, m1), b1)[mDay],
          image(image(moduleAlloc, m1), b1)[mTime] + t1)
         !=
         (image(image(moduleAlloc, m2), b2)[mRoom],
          image(image(moduleAlloc, m2), b2)[mDay],
          image(image(moduleAlloc, m2), b2)[mTime] + t2)
             | m1 : int(1..3), m2 : int(1..3),
               (b1, b1Len) <- module_blocks[m1],
               (b2, b2Len) <- module_blocks[m2],
               t1 : int(0..b1Len - 1), t2 : int(0..b2Len - 1),
               (m1, b1) ~< (m2, b2), (m1, b1) ~< (m2, b2)])
--
$ param
letting module_blocks be [sequence(2,2),sequence(1,1,1),sequence(3); int(1..3)]
```

result of this simplification is shown in Figure 7.25.

Figure 7.25: After simplifying the find statements

```
language Essence 1.3

given module_blocks:
        matrix indexed by [int(1..3)] of sequence of int
find moduleAlloc: function int(5)
  --> function int(1..4) --> record {mRoom : int(1..4),
                                     mDay : int(1..5),
                                     mTime : int(9..17)}
such that
    and([(image(image(moduleAlloc, m1), b1)[mRoom],
          image(image(moduleAlloc, m1), b1)[mDay],
          image(image(moduleAlloc, m1), b1)[mTime] + t1)
         !=
         (image(image(moduleAlloc, m2), b2)[mRoom],
          image(image(moduleAlloc, m2), b2)[mDay],
          image(image(moduleAlloc, m2), b2)[mTime] + t2)
             | m1 : int(1..3), m2 : int(1..3),
               (b1, b1Len) <- module_blocks[m1],
               (b2, b2Len) <- module_blocks[m2],
               t1 : int(0..b1Len - 1), t2 : int(0..b2Len - 1),
               (m1, b1) ~< (m2, b2), (m1, b1) ~< (m2, b2)])
--
$ param
letting module_blocks be [sequence(2,2),sequence(1,1,1),sequence(3); int(1..3)]
```

We now start to apply the comprehension specific simplifications discussed previously (Section 7.2.1), starting with instantiating `m1` and `m2` with values from their domains. This step results in Figure 7.26 which is simpler since the comprehension depends on fewer variables.

Figure 7.26: Instantiating `m1` and `m2` of the comprehension

```
language Essence 1.3

given module_blocks:
        matrix indexed by [int(1..3)] of sequence of int
find moduleAlloc: function int(5)
  --> function int(1..4) --> record {mRoom : int(1..4),
                                     mDay : int(1..5),
                                     mTime : int(9..17)}
such that
    and([(image(image(moduleAlloc, 1), b1)[mRoom],
          image(image(moduleAlloc, 1), b1)[mDay],
          image(image(moduleAlloc, 1), b1)[mTime] + t1)
         !=
         (image(image(moduleAlloc, 1), b2)[mRoom],
          image(image(moduleAlloc, 1), b2)[mDay],
          image(image(moduleAlloc, 1), b2)[mTime] + t2)
             | (b1, b1Len) <- module_blocks[1],
               (b2, b2Len) <- module_blocks[1],
               t1 : int(0..b1Len - 1), t2 : int(0..b2Len - 1),
               (1, b1) ~< (1, b2), (1, b1) ~< (1, b2)])
--
$ param
letting module_blocks be [sequence(2,2),sequence(1,1,1),sequence(3); int(1..3)]
```

We achieve Figure 7.27 by replacing the inner (inequality) expression with a constant (`true`). This vastly simplifies the comprehension while still exhibiting the error. This additionally allows us to easily remove the remaining generators and conditions since they are now unused.

Figure 7.27: Replacing the inner expression of the comprehension with a constant as well as removing redundant conditions of it.

```
language Essence 1.3

given module_blocks:
        matrix indexed by [int(1..3)] of sequence of int
find moduleAlloc: function int(5)
  --> function int(1..4) --> record {mRoom : int(1..4),
                                     mDay : int(1..5),
                                     mTime : int(9..17)}
such that
    and([true
             | (b1, b1Len) <- module_blocks[1],
               (b2, b2Len) <- module_blocks[1],
               t1 : int(0..b1Len - 1), t2 : int(0..b2Len - 1)])
--
$ param
letting module_blocks be [sequence(2,2),sequence(1,1,1),sequence(3); int(1..3)]
```

The `moduleAlloc` decision variable is now unused hence removed. It was only referenced inside the inner expression of the comprehension. The resulting specification is shown in Figure 7.28.

Figure 7.28: Removing `moduleAlloc` decision variable since it is now unused

```
language Essence 1.3

find unused: bool
given module_blocks:
        matrix indexed by [int(1..3)] of sequence of int
such that
    and([true
             | (b1, b1Len) <- module_blocks[1],
               (b2, b2Len) <- module_blocks[1],
               t1 : int(0..b1Len - 1), t2 : int(0..b2Len - 1)])
--
$ param
letting module_blocks be [sequence(2,2),sequence(1,1,1),sequence(3); int(1..3)]
```

We now try removing the unused generators and conditions to see if these are redundant. This results in removing two of the generators. Removing `t1` makes the error disappear, hence *both* `t1` and its dependencies are kept.

Figure 7.29: Removing the unused generators of the comprehension

```
language Essence 1.3

find unused: bool
given module_blocks:
        matrix indexed by [int(1..3)] of sequence of int
such that
    and([true
             | (b1, b1Len) <- module_blocks[1],
               t1 : int(0..b1Len - 1)])
```

Since Figure 7.29 is fully reduced at the Essence level, we now try reducing the refinement (Figure 7.30) at the Essence′ level. Since the error is exhibited in Savile Row, one alternative would be to reduce the Essence′ directly. We have two main reason to reduce the Essence specification first even when error occurs further down the toolchain. The first is that Conjure may still be the cause of the error, such as when the models are inconsistent[7]. The second being that reducing at the Essence level usually converges faster since each domain/expression may have many additional constraints imposed during the refinement process for purposes including ensuring the invariants of a domain and symmetry breaking. Additionally this allows us to

[7] As discussed previously, this is when one model produces a solution and another outputs that the problem is insolvable

output two results, the most reduced Essence specification and the most reduced Essence′ model which may give more insight into the error.

Figure 7.30: The refinement of Figure 7.29

```
language Essence 1.3

find unused: bool
given fin1: int
given fin2: int
given fin3: int
given module_blocks_ExplicitBounded_Length:
        matrix indexed by [int(1..3)] of int(0..fin1)
given module_blocks_ExplicitBounded_Values:
        matrix indexed by [int(1..3), int(1..fin1)] of int(fin2..fin3)
such that
  and([true | q2 : int(1..fin1)
     , t1 : int(0..module_blocks_ExplicitBounded_Values[1, q2])
     , q2 <= module_blocks_ExplicitBounded_Length[1]])

---
$param
letting module_blocks_ExplicitBounded_Length be [2, 3, 1; int(1..3)]
letting module_blocks_ExplicitBounded_Values be
 [[2, 2, 1; int(1..3)], [1, 1, 1; int(1..3)], [3, 1, 1; int(1..3)]; int(1..3)]
letting fin2 be 1
letting fin3 be 3
letting fin1 be 3
```

Figure 7.31 is the result of successfully inlining four of the given statements reducing the dependancy on external data.

Figure 7.31: Inlining the `given` statements

```
language Essence 1.3

find unused: bool
given module_blocks_ExplicitBounded_Length:
        matrix indexed by [int(1..3)] of int(0..3)
such that
    and([true
             | q2 : int(1..3),
               t1 : int(0..[[2, 2, 1; int(1..3)], [1, 1, 1; int(1..3)],
                            [3, 1, 1; int(1..3)]; int(1..3)]
                               [1, q2]),
               q2 <= module_blocks_ExplicitBounded_Length[1]])
---
$Param
letting module_blocks_ExplicitBounded_Length be [2, 3, 1; int(1..3)]
```

We simplify the 2d matrix literal into a single element 1d matrix and adjust the indexing from `[1,q2]` to `[q2]` to take account of the fewer dimensions, resulting in Figure 7.32

Figure 7.32: Simplifying the matrix inside `t1`

```
language Essence 1.3

find unused: bool
given module_blocks_ExplicitBounded_Length:
        matrix indexed by [int(1..3)] of int(0..3)
such that
    and([true
             | q2 : int(1..3), t1 : int(0..[0; int(1..1)][q2]),
               q2 <= module_blocks_ExplicitBounded_Length[1]])
---
$Param
letting module_blocks_ExplicitBounded_Length be [2, 3, 1; int(1..3)]
```

We now remove the condition on `q2`, this leads to the last `given` statement to be unused and hence a candidate for removal, which we succeed in.

Figure 7.33: The final reduction

```
language Essence 1.3

find unused: bool
such that and([true | q2 : int(1..3), t1 : int(0..[0; int(1..1)][q2])])
```

As compared to the user-reduced specification Figure 7.19, our final reduction shows the benefits of our reduction process, producing a specification that is significantly smaller, depending on no external data and having a comprehension with only two generators.

7.6 Test Case Generalisation

In this chapter we showed how to reduce test cases automatically, Chapter 6 showed how we generate test cases automatically. In this section we will show how to *generalise* these test cases to provide insight into the errors found by utilising these contributions.

The generalisation process adapts parts of reduction and generation process. Note that since this process is run after the generation and reduction process, it can reuse the results of testing those candidate specifications, improving efficiency since some candidate specifications are likely to have been tested during the reduction process. In addition the generalisation process in contrast to the generation and reduction processes is explicitly looking for *passing* test cases in addition to failing test cases, to help narrow down the cause of the error.

7.6.1 Process

The generalisation process works using a top down approach replacing larger expressions before replacing smaller expressions. In Figure 7.34 the whole expression on line 5 would be replaced using the method from Section 7.2.1 before the individual sub-expressions would be processed. There are three main kinds of transformations that we apply during the generalisation process: expression replacement as in our example from the reduction process, operators replacement which utilises the information about the operators, and generating expressions of varying complexity utilising the generation process.

Replace with similar: We replace a sub-expression of the specification with sub-expressions of the same type. In Figure 7.34 the Boolean is replaced with other boolean values such as replacing `false` with `true` on the right hand side of the implication. In this example we find that the error only occurs when the right hand side of the implication is `false`.

Figure 7.34: Stack overflow because of invalid unification

```
1 language Essence 1.3
2
3 find var1: bool
4 such that
5     (var1 <-> var1) -> false
```

Replace with similar operators We try replacing the operators in the specifications with other operators such that when given the arguments, the result has the same type as the original. In our example replacing the implication with `<->` still causes the error, but any other applicable operator causes the error to disappear.

Replace with generated: The generation process can be constrained to only generate expressions of a certain resulting type, as well as restricting the depth of the resulting expression. We utilise this to replace a sub-expression with new sub-expressions of the same type but with varying complexity. While this is likely to make the specifications more complex, it may give insight into when the error occurs, additionally these new specifications can be use to validate that a patch for the error is applicable in the general case.

7.6.2 Case studies

Generalisation may at seem to be at odds with reduction since it usually produces specifications which are more complex then the reduced specification, but it gives useful information about when the bug occurs. The output of the generalisation process is also useful for validating if a proposed patch for the original error fixes the root cause and not just the specific case.

7.6.2.1 Generalising Constraints

The following specification (Figure 7.35) is an error that causes invalid flattening in Savile Row.

Figure 7.35: The specification after reduction

```
language Essence 1.3

find var1: matrix indexed by [int(2, 2)] of bool
minimising sum([toInt(var1[l_1]) * 1 | l_1 : int(2..3)])
```

We will describe some of the generalisations which include specifications that did *not* cause the error.

Figure 7.36: Passing

```
language Essence 1.3

find var1: matrix indexed by [int(2, 2)] of bool
minimising sum([toInt(var1[l_1]) * 4 | l_1 : int(2..3)])
```

Showing that the constant multiplier is actually important.

Figure 7.37: Passing

```
language Essence 1.3

find var1: matrix indexed by [int(2, 2)] of bool
minimising sum([toInt(var1[4]) * 1 | l_1 : int(2..3)])
```

Showing that a local variable reference `l_1` is required.

Figure 7.38: Failure

```
language Essence 1.3

find var1: matrix indexed by [int(2, 2)] of bool
minimising sum([toInt(var1[l_1]) * 1 | l_1 : int(2..2)])
```

Figure 7.38 shows that the error occurs even if the matrix indexer's domain and the quantified local variable's domain are the same. Meaning that undefinedness resulting from out of bound indexing is unlikely to be the cause.

7.6.2.2 Generalising Instances

Our second specification causes an exception in Savile Row.

Figure 7.39: Specification where we generalise the instance

```
language Essence 1.3

given n: int(3..20)
find m: matrix indexed by [int(4..n)] of bool
---
$ Starting param Param
letting n be 10
```

For parametrised specifications the generalisation process is extended to generate new instances until either the time budget is exceeded or an (optionally) user specified number of instances have been generated. We generate the instances using our instance generation methods from Chapter 5, for our example in Figure 7.39 the generated values are in the range 3–20 (duplicates are ignored). The generated values show that specification only causes an error when `n` is less then 4. This error was fixed in Savile Row by treating ranges such as `int(3..2)` as the empty domain i.e. `int()`.

7.7 Summary

In this chapter, we have discussed how to reduce an arbitrary Essence specification exhibiting an error to a much simpler form. We show how to produce a refinement of a reduced specification without having to refine all models, by adapting the choices that were made for the original specification. Notably we can reduce parametrised specifications, generating new instances when required using our generation methods from Chapter 5. Additionally we have shown how to generalise a specification to provide more insight into the cause of the error. We conclude with a case study describing the reductions in depth using both generated specifications and specifications given by users.

Chapter 8

Conclusion

This thesis presented two main contributions to automate the model selection process and the testing of a constraint modelling toolchain.

Automatic modelling allows a non-expert user to realise the potential of constraint programming by abstracting over modelling decisions. When using a refinement based approach from an abstract specification in languages such as Essence, many candidate models can be produced which represent the different ways the problem can be encoded, each with their own performance characteristics. Additionally a specification may be parameterised representing a problem class meaning that external input data is utilised. Resulting from this the most appropriate model may depend on the given instance.

This thesis presented a process to automate the model selection process by generating instances from the instance space defined by an abstract specification in Essence (Chapter 5). This process can usually select a small subset of the models that exhibit superior performance. Notably this process has full coverage of the Essence specification language including specifications whose parameters depend on the value of another parameter, which was achieved by synthesising a generation order for the parameters such that parameters on which other depend on are generated first. Through experimentation we have shown that our instance generation process is applicable to a wide range of problem classes. Additionally we have shown through experimentation that our process is scaleable to larger instance spaces.

For the purposes of testing and validating the toolchain described in Chapter 3 this thesis presented a way of generating complete specifications, this process provides full coverage of the

Essence language (Chapter 6). Notably this process can generate parameterised specification by utilising our instance generation methods. Additionally we showed how the generation process can be automatically reconfigured to guide the generation process to different errors. We showed though cased studies the various errors that were found by the generator.

Once we generated a specification we showed a way to *reduce* a specification to a much simpler form which still exhibits the same error (Chapter 7). Additionally this process is also applicable to specifications that were found by a user and can help pinpoint the error. The reductions range from course grained simplifications, such as removing whole constraints to more intricate simplifications which reason about the types of the expressions involved to perform transformations such as replacing a tuple with an element of the tuple. The reduction process also utilised our instance generation process to generate new instances when performing simplifications upon the domains of the parameters. We have shown through case studies and worked examples how the reduction process works as the well as the gains in simplicity that can be achieved. This chapter also included a *generalisation* extension which generates complementary specifications that provide insight into when the error occurs. Notably the generalisation process produces similar specifications that do *not* cause an error, to help find the cause of the error.

In conclusion this thesis showed a way of generating instances data for an arbitrary Essence specification for the purpose of model selection, a way of generating complete Essence specifications for the purpose of validating a constraint modelling toolchain and a method of simplifying a specification that causes an error to a much simpler form that still exhibits the same error.

8.1 Future Work

8.1.1 Enhance the Generation Process

Currently the generation process generates specifications that use the default settings for tools in the toolchain. Enhancing the generator to be able to produce arbitrary combinations of search orderings and heuristics would allow testing how these features interact.

8.1.2 Extend Testing to Other Paradigms

Instead of just using a single constraint solver for solving a generated specification, we can utilise multiple solvers, by translating the ESSENCE$'$ models into alternative paradigms such as MiniZinc or SAT. This would be particularly useful when verifying if a specification actually had no solution, since if we use multiple solvers, and there is a solution to the specification then one the of the solvers is likely to produce a solution, which we can easily verify. This will additionally allow us to find inconsistencies in the translation process to these paradigms.

8.1.3 Apply the Reduction Process to Other Paradigms

Our test case reducer works at both the ESSENCE and the ESSENCE$'$ level. If more target languages (e.g. MiniZinc or SAT) were added to the refinement process it would be useful to extend the reduction process to able to apply reductions to these paradigms directly.

8.1.4 Extend the Instance Generation Process

Our instance generation process works for any parameterised ESSENCE specification and can select a set of instances that have good performance across the instance space. As discussed in Section 5.6, users might want the instances to have specific characteristics such as always being solvable. A way to achieve this could be to determine which constraint was the cause of the instance being insolvable (using Quickxplain [Jun04] for example). We can then try to regenerate only parts of the of the instances that were referenced in the offending constraint.

8.1.5 Extend the Reduction process

Our system can reduce a test case to a much simpler form which exhibits a similar error. One extension is to apply the reduction process to a set of test case and consider other metrics such as code coverage. This could be used to reduce a set of test cases to a much smaller set of tests which still provide the same code coverage.

Appendix A

Annotated Grammar

Each kind of sub-expression has a weighting associated with it ranging from 1 to 1000 with higher values increasing the chance that the sub-expression will be selected, with the default value set to 1000. When a choice needs to be made a weighted random distribution is utilised after removing sub-expressions that would be inapplicable at that point, meaning that an individual sub-expression's probability is proportional to the other available sub-expressions' probabilities. As an example when there are two applicable operators `subset` and `supset` with the weightings 10 and 40 respectively, `subset` would be 4 times more likely to be picked.

To demonstrate how the selection process work and how expression is deemed to be applicable we will consider how an operator is chosen. To determine if an operator is applicable we use the algorithm outlined in Figure A.1. The algorithm firstly checks if we are able to generate the required sub-expressions, such as sets for the `subset` operator. This is required since the generator can be configured, to not use certain types, which is used among other things for the generation of Essence′. Secondly the algorithm checks if requirements of the operation are satisfied, these including type restrictions, and the depth required. Finally the weights for the allowed operators are retrieved and operator chosen randomly in proportion to the operator's weight.

A.1 Worked Example

Consider `{1} subset toSet(mset(1,1,2))` (visualised in Figure A.5 on page 114) generated using a depth of 3 as an example of the `subset` operator, annotated in Figure A.2. The *requires*

Figure A.1: Operator Selection

```
retTy is type we want to generate
depthLeft is the maximum depth we can use

generate-op(retTy, depthLeft)
    choices <- [ op | op <- all-ops(), op.is-valid(retTy,depthLeft) ]
    if choices == [] then
        return None
    else
        chosen <- weightedChoice(choices)
        return chosen.generate()

Operator.is-valid(retTy, depthLeft)
    if ! can-generate(self.requires, retTy) then
        return False

    if ! satisfy(self.restrictions, retTy, depthLeft) then
        return False

    return True
```

section specifics the expressions that we need to be able to generate. The *restrictions* section specifies the properties of operator, in our example, we always return a boolean and we need at least a depth of 2. To actually generate the `subset` we pick a type for both sides of the operator, then generate two expressions of those types.

Figure A.2: Subset Operator

```
requires:      any [ Set, MSet, Function, Relation ] /\  any [ Boolean ]
restrictions:  depthLeft >= 1
               retTy == Boolean
generate:
    ty <- weightedChoice [ Set, MSet, Function, Relation ]
    left <- generate ty
    right <- generate ty
    return (subset left right)
```

To generate an expression of type `set of int` for our example we use the process from Section 6.2.2 where the main options are a literal, referencing a decision variable or an another operator. For the left hand side of the `subset` operator, a literal is chosen. Since we need a `set of int`, there only a single option namely a set literal. To the generate the inner expression of the set namely the 1 we use the generation process to produce an expression of type int, since a literal was chosen, a random integer is the result. The right side is more interesting since an another operator is chosen. From the applicable operators which include set operators (`Union`) and type conversions (`toSet`, `preImage`), `toSet` was chosen which is annotated in Figure A.3. We have two choices for the inner expression of `toSet`, a `mset` as in our example and a set such `toSet({1,2,3})`, a function is inapplicable since it would return a set of tuples. When generating the inner expression of `toSet` only literals that are not nested are available since the `depthLeft` is 1.

Figure A.3: toSet Operator

```
requires:      any [ Set, MSet]
restrictions:  depthLeft >= depthOf retTy
               retTy == TypeSet tau
generate:
    ty <- weightedChoice [ Set tau , MSet tau, Function tau if tau is a tuple
]
    inner <- generate ty
    return (toSet inner)
```

If `preImage` (Figure A.4) was chosen for the right side of the `subset` a possible choice would be `{1} subset preImage(function(1 --> true, 2 --> false), false )`. `preimage(f, val)` returns the sets of values that are mapped to `val`, hence the type of the codomain is unconstrained, nevertheless there are only three choices for it (booleans, integers and enumerated types), since `depthleft` is 0.

Figure A.4: preImage Operator

```
requires:      all [ Function, tau ]
restrictions:  depthLeft >= depthOf retTy
               retTy == TypeSet tau
generate:
    codomain <- generateType
    func <- generate (Function tau codomain)
    val  <- generate codomain
    return (preImage func val)
```

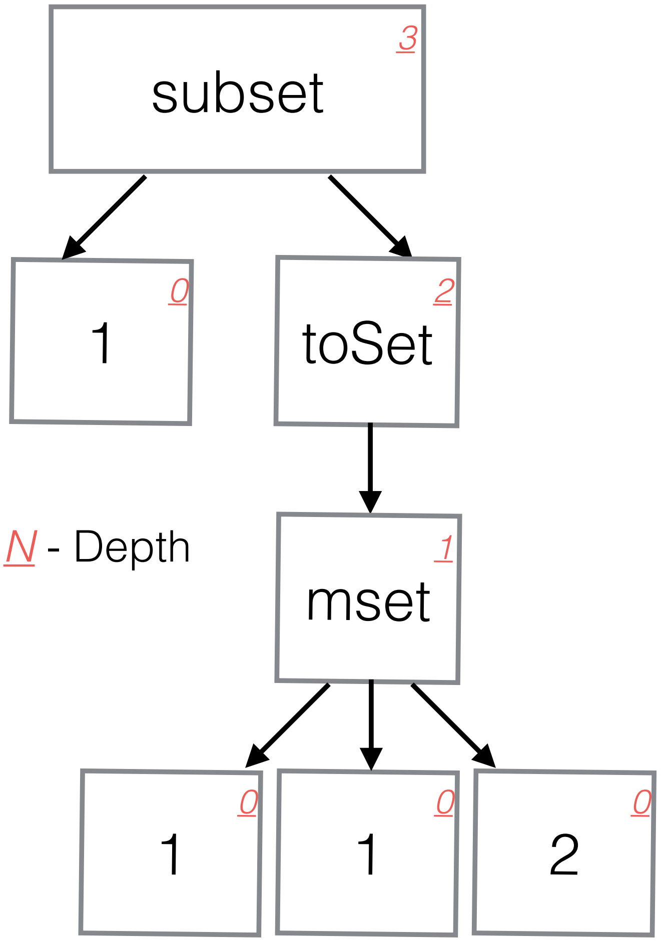


Figure A.5: Depth of our example expression: {1} subset toSet(mset(1,1,2))

Bibliography

[ABL09] Carlos Ansótegui, Maria Luisa Bonet and Jordi Levy. “Towards Industrial-like Random SAT Instances”. In: *The 21st International Joint Conference on Artificial Intelligence*. IJCAI’09. 2009, pp. 387–392.

[ABM98] Paul E Ammann, Paul E Black and William Majurski. “Using model checking to generate tests from specifications”. In: *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods*. 1998, pp. 46–54.

[Ach+00] Dimitris Achlioptas, Carla P Gomes, Henry A Kautz and Bart Selman. “Generating Satisfiable Problem Instances”. In: *The Seventeenth National Conference on Artificial Intelligence*. Ed. by Bruce W Porter. 2000, pp. 256–261.

[Akg] Özgür Akgün. *CSPLib Problem 133: Knapsack Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob133.

[Akg+13a] Özgür Akgün, Alan M Frisch, Ian P Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel and Peter Nightingale. “Automated Symmetry Breaking and Model Selection in Conjure”. In: *The 19th International Conference on Principles and Practice of Constraint Programming*. Ed. by Christian Schulte. Sept. 2013, pp. 107–116. ISBN: 978-3-642-40626-3. DOI: 10.1007/978-3-642-40627-0_11.

[Akg+13b] Ozgur Akgun, Alan M Frisch, Bilal Syed Hussain, Christopher A Jefferson, Lars Kotthoff, Ian Miguel and Peter Nightingale. "An Automated Constraint Modelling and Solving Toolchain". In: *20th Automated Reasoning Workshop*. Apr. 2013.

[Akg14] Özgür Akgün. "Extensible automated constraint modelling via refinement of abstract problem specifications". PhD thesis. 2014.

[Alf+04] José Júlio Alferes, Francisco Azevedo, Pedro Barahona, Carlos Viegas Damásio and Terrance Swift. "Deductive diagnosis of digital circuits". In: *Artificial Intelligence Applications and Innovations*. 2004, pp. 155–165.

[Ana+13] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold and Phil Mcminn. "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation". In: *Journal of Systems and Software* 86.8 (Aug. 2013), pp. 1978–2001. DOI: 10.1016/j.jss.2013.02.061.

[Ans+12] Carlos Ansótegui, Maria Luisa Bonet, Jordi Levy and Chu Min Li. "Analysis and Generation of Pseudo-Industrial MaxSAT Instances". In: *15th International Conference of the Catalan Association for Artificial Intelligence*. Ed. by David Riaño, Eva Onaindia and Miguel Cazorla. 2012, pp. 173–184. DOI: 10.3233/978-1-61499-139-7-173.

[Ans+13] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy and Mateu Villaret. "Solving weighted CSPs with meta-constraints by reformulation into satisfiability modulo theories". In: *Constraints* 18.2 (Apr. 2013), pp. 236–268. ISSN: 1572-9354. DOI: 10.1007/s10601-012-9131-1.

[Aze] Francisco Azevedo. *CSPLib Problem 042: diagnosis*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob042.

[BB] David A. Burke and Kenneth N. Brown. *CSPLib Problem 048: Minimum Energy Broadcast (MEB)*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob048.

[BB07] David A Burke and Kenneth N Brown. "Using relaxations to improve search in distributed constraint optimisation". In: *Artificial Intelligence Review* 28.1 (2007), pp. 35–50.

[Bes+05] Christian Bessiere, Remi Coletta, Frédéric Koriche and Barry O'Sullivan. "A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems". In: *Machine Learning: ECML 2005*. Vol. 3720. 2005, pp. 23–34. ISBN: 978-3-540-29243-2. DOI: 10.1007/11564096_8.

[BFG03] Marius Bozga, Jean-Claude Fernandez and Lucian Ghirvu. "Using static analysis to improve automatic test generation". In: *International Journal on Software Tools for Technology Transfer* 4.2 (2003), pp. 142–152. ISSN: 1433-2779. DOI: 10.1007/s10009-002-0098-x.

[Bir+02] Mauro Birattari, Thomas Stützle, Luis Paquete and Klaus Varrentrapp. "A Racing Algorithm for Configuring Metaheuristics." In: *GECCO 2002*. Vol. 2. 2002, pp. 11–18.

[BLN01] Philippe Baptiste, Claude Le Pape and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Springer US, 2001, pp. 1–18. ISBN: 978-1-4615-1479-4. DOI: 10.1007/978-1-4615-1479-4_1.

[BS11] Nicolas Beldiceanu and Helmut Simonis. "A Constraint Seeker: Finding and Ranking Global Constraints from Examples". In: *The 17th International Conference on Principles and Practice of Constraint Programming*. Sept. 2011, pp. 12–26. DOI: 10.1007/978-3-642-23786-7_4.

[BS12] Nicolas Beldiceanu and Helmut Simonis. "A Model Seeker: Extracting Global Constraint Models from Positive Examples". In: *The 18th International Conference on Principles and Practice of Constraint Programming*. Ed. by Michela Milano. 2012, pp. 141–157. DOI: 10.1007/978-3-642-33558-7_13.

[Cav+02] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman and Laurent Mounier. "Using UML for automatic test generation". In: *In International Symposium on Software Testing and Analysis (ISSTA)*. 2002.

[CF10] Avik Chaudhuri and Jeffrey S Foster. “Symbolic Security Analysis of Ruby-on-rails Web Applications”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS ’10. 2010, pp. 585–594. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866373. URL: http://doi.acm.org/10.1145/1866307.1866373.

[CH11] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *SIGPLAN Notices* 46.4 (2011), pp. 53–64. DOI: 10.1145/1988042.1988046.

[CZ00] Holger Cleve and Andreas Zeller. “Finding failure causes through automated testing”. In: *4th International Workshop on Automated Debugging* (2000).

[Dis+12] Andreas Distler, Christopher Jefferson, Tom Kelsey and Lars Kotthoff. “The Semigroups of Order 10”. In: *The 18th International Conference on Principles and Practice of Constraint Programming*. Ed. by Michela Milano. 2012, pp. 883–899. DOI: 10.1007/978-3-642-33558-7_63.

[EW04] M V Emmerik and T Waddington. “Using a decompiler for real-world source recovery”. In: *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. Nov. 2004, pp. 27–36. DOI: 10.1109/WCRE.2004.42.

[FPÅ04] Pierre Flener, Justin Pearson and Magnus Ågren. “Introducing ESRA, a Relational Language for Modelling Combinatorial Problems”. In: *Logic Based Program Synthesis and Transformation*. Ed. by Bruynooghe Maurice. 2004, pp. 214–232. DOI: 10.1007/978-3-540-25938-1_18.

[Fri+08] Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández and Ian Miguel. “Essence: A constraint language for specifying combinatorial problems”. In: *Constraints* 13.3 (Sept. 2008), pp. 268–306. ISSN: 1383-7133. DOI: 10.1007/s10601-008-9047-y.

[Fuk02] Alex Fukunaga. “Automated Discovery of Composite SAT Variable-selection Heuristics”. In: *The Eighteenth National Conference on Artificial Intelligence*. 2002, pp. 641–648. ISBN: 0-262-51129-0.

[G+08] Patrice Godefroid, Michael Y Levin, David A Molnar et al. “Automated Whitebox Fuzz Testing.” In: *Network Distributed Security Symposium (NDSS)*. Vol. 8. 2008, pp. 151–166.

[Gecode] Gecode Team. *Gecode: Generic Constraint Development Environment*. 2006.

[Gen+14] Ian P Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Glenna F Nightingale and Peter Nightingale. “Discriminating Instance Generation for Automated Constraint Model Selection”. In: *The 20th International Conference on Principles and Practice of Constraint Programming*. Sept. 2014, pp. 356–365. DOI: 10.1007/978-3-319-10428-7_27.

[Gen+99] Ian P Gent, Holger H Hoos, Patrick Prosser and Toby Walsh. “Morphing: Combining Structure and Randomness”. In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence*. Ed. by Jim Hendler and Devika Subramanian. 1999, pp. 654–660. ISBN: 0-262-51106-1.

[GJM06] Ian P Gent, Christopher Jefferson and Ian Miguel. “Minion: A Fast Scalable Constraint Solver”. In: *The 17th European Conference on Artificial Intelligence*. Ed. by Gerhard Brewka, Silvia Coradeschi, Anna Perini and Paolo Traverso. Vol. 141. Frontiers in Artificial Intelligence and Applications. 2006, pp. 98–102.

[GS97] Carla P Gomes and Bart Selman. “Algorithm portfolio design: Theory vs. practice”. In: *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*. 1997, pp. 190–197.

[Har] Warwick Harvey. *CSPLib Problem 010: Social Golfers Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob010.

[HHL11] Frank Hutter, Holger H Hoos and Kevin Leyton-Brown. “Sequential Model-Based Optimization for General Algorithm Configuration”. In: *Learning and Intelligent Optimization*. Jan. 2011, pp. 507–523. DOI: 10.1007/978-3-642-25566-3_40.

[HKW] Brahim Hnich, Zeynep Kiziltan and Toby Walsh. *CSPLib Problem 030: Balanced Academic Curriculum Problem (BACP)*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob030.

[HKW02] Brahim Hnich, Zeynep Kiziltan and Toby Walsh. "Modelling a Balanced Academic Curriculum Problem". In: *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*. 2002, pp. 121–131.

[HLH97] Bernardo A Huberman, Rajan M Lukose and Tad Hogg. "An economics approach to hard computational problems". English. In: *Science* 275 (Mar. 1997), pp. 51–54. ISSN: 00368075.

[HLS05] Peter Hawkins, Vitaly Lagoon and Peter J Stuckey. "Solving Set Constraint Satisfaction Problems using {ROBDDs}". In: *Journal of Artificial Intelligence Research* 24 (2005), pp. 109–156.

[HMG13] Bilal Syed Hussain, Ian Miguel and Ian P Gent. "Instance Generation for Constraint Model Selection". In: *19th International Conference on Principles and Practice of Constraint Programming - Doctoral Program*. Sept. 2013.

[Hni] Brahim Hnich. *CSPLib Problem 034: Warehouse Location Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob034.

[Hni03] Brahim Hnich. "Thesis: Function variables for constraint programming". In: *AI Communications* 16.2 (2003), pp. 131–132.

[HPB04] Lu Hong, Scott E Page and William J Baumol. "Groups of Diverse Problem Solvers Can Outperform Groups of High-Ability Problem Solvers". In: *Proceedings of the National Academy of Sciences of the United States of America* 101.46 (2004), pp. 16385–16389. ISSN: 00278424.

[HSW04] Brahim Hnich, Barbara M. Smith and Toby Walsh. "Dual modelling of permutation and injection problems". In: *Journal of Artificial Intelligence Research* 21 (1 Feb. 2004), pp. 357–391. ISSN: 1076-9757.

[Hur+14] Barry Hurley, Lars Kotthoff, Yuri Malitsky and Barry O'Sullivan. "Proteus: A Hierarchical Portfolio of Solvers and Transformations". In: *The 11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by Helmut Simonis. May 2014, pp. 301–317. ISBN: 978-3-319-07046-9. DOI: 10.1007/978-3-319-07046-9_22.

[Jef+99] Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent, eds. *CSPLib: A problem library for constraints*. http://www.csplib.org. 1999.

[Jun04] Ulrich Junker. "QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems". In: *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*. Ed. by Deborah L McGuinness and George Ferguson. 2004, pp. 167–172.

[KBS10] Leslie De Koninck, Sebastian Brand and Peter J Stuckey. "Data Independent Type Reduction for Zinc". In: *9th International Workshop on Constraint Modelling and Reformulation*. 2010.

[KH01] Zeynep Kiziltan and Braham Hnich. "Symmetry Breaking in a Rack Configuration Problem". In: *the IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints*. 2001.

[Kin75] James C King. "A New Approach to Program Testing". In: *Proceedings of the International Conference on Reliable Software*. 1975, pp. 228–233. DOI: 10.1145/800027.808444.

[Kot14] Lars Kotthoff. "Algorithm Selection for Combinatorial Search Problems: A Survey". In: *{AI} Magazine* (2014), pp. 1–17.

[KPS10] Amba Kulkarni, Sheetal Pokar and Devanand Shukl. “Sanskrit Computational Linguistics: 4th International Symposium”. In: ed. by Girish Nath Jha. Springer Berlin Heidelberg, 2010. Chap. Designing, pp. 70–90. ISBN: 978-3-642-17528-2. DOI: 10.1007/978-3-642-17528-2_6. URL: http://dx.doi.org/10.1007/978-3-642-17528-2%7B%5C_%7D6.

[LF11] John Lenz and Nick Fitzgerald. *Source Map Revision 3 Proposal*. Tech. rep. Feb. 2011. URL: https://docs.google.com/document/d/1U1RGAehQwRypUTovF1KRlpiOFze0b-_2gc6fAH0KY0k/.

[LGL12] Nadjib Lazaar, Arnaud Gotlieb and Yahia Lebbah. “A CP framework for testing CP”. In: *Constraints* 17.2 (Apr. 2012), pp. 123–147. ISSN: 1572-9354. DOI: 10.1007/s10601-012-9116-0.

[Li+14] Huiqing Li, Simon J Thompson, Pablo Lamela Seijas and Miguel Angel Francisco. “Automating property-based testing of evolving web services”. In: *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation*. Ed. by Wei-Ngan -. N Chin and Jurriaan Hage. 2014, pp. 169–180. DOI: 10.1145/2543728.2543741.

[Lit+03] James Little, Cormac Gebruers, Derek Bridge and Eugene C Freuder. “Using Case-Based Reasoning to Write Constraint Programs”. In: *The 9th International Conference on Principles and Practice of Constraint Programming*. Jan. 2003, p. 983. DOI: 10.1007/978-3-540-45193-8_107.

[Mar+08] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J Stuckey, Maria Garcia la de Banda and Mark Wallace. “The Design of the Zinc Modelling Language”. In: *Constraints* 13.3 (Sept. 2008), pp. 229–267. DOI: 10.1007/s10601-008-9041-4.

[Mea+11] Christopher Mears, Todd Niven, Marcel Jackson and Mark Wallace. “Proving Symmetries by Model Transformation”. In: *The 17th International Conference on Principles and Practice of Constraint Programming*. Sept. 2011, pp. 591–605. DOI: 10.1007/978-3-642-23786-7_45.

[MGM14] Morten Mossige, Arnaud Gotlieb and Hein Meling. "Using CP in Automatic Test Generation for ABB Robotics Paint Control System". In: *The 20th International Conference on Principles and Practice of Constraint Programming*. Ed. by Barry O'Sullivan. Sept. 2014, pp. 25–41. DOI: 10.1007/978-3-319-10428-7_6.

[Net+07] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck and Guido Tack. "MiniZinc: Towards a standard CP modelling language". In: *The 13rd International Conference on Principles and Practice of Constraint Programming*. 2007, pp. 529–543. DOI: 10.1007/978-3-540-74970-7_38.

[Nig+14] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson and Ian Miguel. "Automatically Improving Constraint Models in Savile Row through Associative-Commutative Common Subexpression Elimination". In: *The 20th International Conference on Principles and Practice of Constraint Programming*. Sept. 2014, pp. 590–605. DOI: 10.1007/978-3-319-10428-7_43.

[NSM15] Peter Nightingale, Patrick Spracklen and Ian Miguel. "Automatically Improving SAT Encoding of Constraint Problems Through Common Subexpression Elimination in Savile Row". In: *The 21th International Conference on Principles and Practice of Constraint Programming*. Ed. by Gilles Pesant. 2015, pp. 330–340. ISBN: 978-3-319-23219-5. DOI: 10.1007/978-3-319-23219-5_23.

[Pug04] Jean-Francois -. F Puget. "Constraint Programming Next Challenge: Simplicity of Use". In: *The 10th International Conference on Principles and Practice of Constraint Programming*. Sept. 2004, pp. 5–8. DOI: 10.1007/978-3-540-30201-8_2.

[Reg+12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison and Xuejun Yang. "Test-case Reduction for C Compiler Bugs". In: *SIGPLAN Notices* 47.6 (June 2012), pp. 335–346. DOI: 10.1145/2345156.2254104.

[Ric76] John R Rice. "The Algorithm Selection Problem". In: *Advances in Computers* 15 (1976), pp. 65–118.

[Ruda] Jesse Ruderman. *Introducing jsfunfuzz*. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. 2007, accessed on 2016-03-23.

[Rudb] Jesse Ruderman. *jsfunfuzz Project Page*. https://github.com/MozillaSecurity/funfuzz/. Accessed on 2016-03-23.

[Rudc] Jesse Ruderman. *lithium Project Page*. https://github.com/MozillaSecurity/lithium. Accessed on 2016-03-23.

[RVW06] Francesca Rossi, Peter Van Beek and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. ISBN: 0 444 52726 5.

[Sel09] Meinolf Sellmann. "Approximated consistency for the automatic recording constraint". In: *Computers & Operations Research* 36 (8 Aug. 2009), pp. 2341–2347. ISSN: 0305-0548. DOI: 10.1016/j.cor.2008.08.009.

[SM07] Horst Samulowitz and Roland Memisevic. "Learning to solve QBF". In: *The Twenty-Second Conference on Artificial Intelligence*. Vol. 7. 2007, pp. 255–260.

[SM14] Mirko Stojadinović and Filip Marić. "meSAT: multiple encodings of CSP to SAT". In: *Constraints* 19.4 (Oct. 2014), pp. 380–403. DOI: 10.1007/s10601-014-9165-7.

[Smi] Barbara Smith. *CSPLib Problem 039: The Rehearsal Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob039.

[Smi+95] Barbara M Smith, Sally C Brailsford, Peter M Hubbard and H Paul Williams. "The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared". In: *The 1st International Conference on Principles and Practice of Constraint Programming*. 1995, pp. 36–52.

[SML96] Bart Selman, David G Mitchell and Hector J Levesque. "Generating hard satisfiability problems". In: *Artificial Intelligence* 81.1 (1996), pp. 17–29. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/0004-3702(95)00045-3.

[Str14] Daniel W Stroock. *An Introduction to Markov Processes*. Springer Berlin Heidelberg, 2014, pp. 1–23. ISBN: 978-3-642-40523-5. DOI: 10.1007/978-3-642-40523-5_1.

[SV11] Valdivino de Alexandre Santiago Júnior and Nandamudi Lankalapalli Vijaykumar. "Generating model-based test cases from natural language requirements for space application software". In: *Software Quality Journal* 20.1 (2011), pp. 77–143. ISSN: 1573-1367. DOI: 10.1007/s11219-011-9155-6.

[Van+99] Pascal Van Hentenryck, Laurent Michel, Laurent Perron and J-C Régin. "Constraint programming in OPL". In: *Principles and Practice of Declarative Programming*. 1999, pp. 98–116.

[Van99] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999. ISBN: 0-262-72030-2.

[Wala] Toby Walsh. *CSPLib Problem 013: Progressive Party Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob013.

[Walb] Toby Walsh. *CSPLib Problem 024: Langford's number problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh and Ian P. Gent. http://www.csplib.org/Problems/prob024.

[War04] M P Ward. "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations". In: *Sci. Comput. Program.* 52.1-3 (Aug. 2004), pp. 213–255. ISSN: 0167-6423. DOI: 10.1016/j.scico.2004.03.007. URL: http://dx.doi.org/10.1016/j.scico.2004.03.007.

[WM97] D H Wolpert and W G Macready. "No free lunch theorems for optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1 (Apr. 1997), pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893.

[WMP13] Hao Wu, Rosemary Monahan and James F Power. "Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver". In: *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*. July 2013, pp. 175–182. DOI: 10.1109/TASE.2013.31.

[XHL10] Lin Xu, Holger Hoos and Kevin Leyton-Brown. "Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection." In: *The Twenty-Fourth AAAI Conference on Artificial Intelligence*. Vol. 10. 2010, pp. 210–216.

[Xu+08] Lin Xu, Frank Hutter, Holger H Hoos and Kevin Leyton-Brown. "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *Journal of Artificial Intelligence Research* 32.1 (June 2008), pp. 565–606. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=1622673.1622687.

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *SIGPLAN Notices* 46.6 (June 2011), pp. 283–294. DOI: 10.1145/1993316.1993532.

[ZGA14] Chaoqiang Zhang, Alex Groce and Mohammad Amin Alipour. "Using Test Case Reduction and Prioritization to Improve Symbolic Execution". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. 2014, pp. 160–170. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610392.

List of Figures

List of Tables