

*A Novel Representation for Search-Based  
Model-Driven Engineering*

*James R. Williams*

*Submitted for the degree of Doctor of Philosophy*

*University of York  
Department of Computer Science  
September, 2013*



# *Abstract*

Model-Driven Engineering (MDE) and Search-Based Software Engineering (SBSE) are development approaches that focus on automation to increase productivity and throughput. MDE focuses on high-level domain *models* and the automatic management of models to perform development processes, such as model validation or code generation. SBSE on the other hand, treats software engineering problems as *optimisation problems*, and aims to automatically discover solutions rather than systematically construct them. SBSE techniques have been shown to be beneficial at all stages in the software development life-cycle. There has, however, been few attempts at applying SBSE techniques to the MDE domain and all are problem-specific. In this thesis we present a method of encoding MDE models that enables many robust SBSE techniques to be applied to a wide-range of MDE problems. We use the model *representation* to address three in-scope MDE problems: discovering an optimal domain model; extracting a model of runtime system behaviour; and applying sensitivity analysis to model management operations in order to analyse the uncertainty present in models. We perform an empirical analysis of two important properties of the representation, *locality* and *redundancy*, which have both been shown to affect the ability of SBSE techniques to discover solutions, and we propose a detailed plan for further analysis of the representation, or other representations of its kind.



# *Contents*

<b>I</b>	<b>Introduction</b>	<b>1</b>
1	Introduction	3
2	Literature Review	9
<b>II</b>	<b>Making Models Searchable</b>	<b>39</b>
3	Proof of Concept Representation for Models	41
4	A Generic Representation for Models	61
<b>III</b>	<b>Applying Search to Models</b>	<b>99</b>
5	Searching for Optimal Models	101
6	Sensitivity Analysis in Model-Driven Engineering	135
<b>IV</b>	<b>Evaluation and Conclusions</b>	<b>153</b>
7	Analysis of the Properties of the Model Representation	155
8	Conclusion	195
<b>V</b>	<b>Appendix</b>	<b>203</b>
A	Literature Review	205
B	Code Listings	211
<b>VI</b>	<b>References</b>	<b>225</b>
	Bibliography	227



# List of Figures

2.1	Illustration of the MOF metamodeling architecture. Adapted from [16] and [120].	11
2.2	A metamodel for a simple language which allows zookeepers to assign animals to cages.	11
2.3	A example model which conforms to the metamodel in figure 2.2.	12
2.4	A simple illustration of hill-climbing on a rugged landscape. In this example, only a local optimum will be reached.	21
2.5	Illustration of the crossover and mutation genetic operators used in genetic algorithms.	22
2.6	An abstract illustration of the terminology of representations and evolutionary algorithms.	23
2.7	An illustration of locality and non-locality. Each adjacent cell has the minimum genotypic distance. The colour intensity of each cell illustrates the phenotypic distance to its genotypic neighbours.	25
2.8	An illustration of the two types of redundancy. Each square is a genotype and its shading represents its phenotype.	25
3.1	The Ecore metamodel for the Fighter Description Language used in SAF.	44
3.2	The experimental process to address the two SAF experimental goals.	46
3.3	Illustration of the choose operator.	46
3.4	An example of genotype-to-phenotype mapping in Grammatical Evolution. Simon Poulding is credited for this image and example.	47
3.5	The genotype: a metamodel encoding individuals from the metaheuristic search framework.	48
3.6	The process of transforming the genotype (an integer string used by the search framework) into the phenotype (a specific fighter description in FDL, for use in SAF).	49
3.7	The Characteristic grammar rule shown in listing 3.2 from the FDL Xtext grammar, represented as an object diagram. The diagram visualises how the model of the grammar expresses rule choice. We expand the first choice of the grammar rule: to specify the punchReach characteristic. Leaves are terminals in the grammar.	50

3.8	The results of analysing the locality and redundancy of GE with respect to the FDL. ....	58
3.9	Four syntactically different, but semantically equivalent, FDL fighters. ....	59
4.1	A conceptual overview of the components of our generic model representation. ....	62
4.2	The metamodel for the finitisation model, written in MOF. The types Object, Element, Class, and Property refer to MOF classes, and represent references to the metamodel being finitised. ....	64
4.3	A simple metamodel illustrating the use of the optional parentClass field in the FeatureFinitisation class of our finitisation model. ....	65
4.4	An illustration of specifying the root meta-class to focus the search space on a particular area of a metamodel. ....	66
4.5	Selecting parts of the phenotype for crossover. ....	67
4.6	The genotype of our representation. <i>Segments</i> are the building block for representing models as integers. (a) is the <i>feature selector bit</i> , and (b) is the <i>feature value bit</i> . ....	69
4.7	An illustration of the processes involved in transforming from our genotype to a model conforming to a given metamodel. ....	70
4.8	A simple metamodel for describing the layout of zoos. (Repeated from figure 2.2.) ....	71
4.9	Step 1: Finitisation information for the zoo metamodel. ....	71
4.10	Step 2: Assigning identifiers to the zoo metamodel. ....	71
4.11	Step 3: Mapping a segment to an Animal object. ....	72
4.12	An illustration of the reference construction phase. ....	73
4.13	Illustration of the crossover operator. ....	76
4.14	Illustration of the mutation operator. ....	76
4.15	An illustration of a metamodel which, without proper finitisation, will result in a representation with no redundancy. An individual with more than three segments will be unable to encode the solution. ....	77
4.16	The building blocks of the model-based metaheuristic framework that uses our representation at its core. ....	78
4.17	The metamodel that defines the structure of individuals in Crepe. ....	79
4.18	The metamodel used to capture the execution of a search algorithm. ....	80
4.19	A conceptual view of the inputs to and outputs of MBMS. The diagonally shaded box represents an optional input. .	83
4.20	The initial screen presented to the user. ....	85
4.21	The search model visualised as a graph. ....	86
4.22	An individual transformed into its HUTN format and presented to the user for inspection. ....	87
4.23	Visualisation of map models. ....	92
4.24	Converging on the optimal solution for the delivery man problem. ....	93



5.1	The Ecore metamodel for the Fighter Description Language; repeat of figure 3.1.....	103
5.2	A search-amenable Ecore metamodel for the Fighter Description Language. ....	105
5.3	Illustration of how long it took each repetition to converge on an optimal solution. The Y-axis shows the proportion of repetitions that have converged. ....	108
5.4	The control loop of self-adaptive systems. Adapted from [24]. ....	112
5.5	The process of extracting a model of the environment and using it to search for optimal system behaviour. ....	114
5.6	The proposed adaptation loop for automatically selecting opponents in SAF based on the logs of a player's behaviour.	117
5.7	The data metamodel defined to meaningfully capture the information contained in SAF log files. ....	118
5.8	The process of extracting a model of the player used for the experimentation. ....	122
5.9	The results of the experimentation. The lines in the centre of the boxes represents the median. ....	125
5.10	The SAF adaptation experimental results, grouped by factor. ....	126
5.11	The results of random search. The lines in the centre of the boxes represents the median. ....	128
5.12	The results of running the experiment at the optimal parameters discovered using CCD. ....	131
6.1	The two modelling languages of CATMOS. ....	137
6.2	The Pareto front and first non-dominating rank produced by CATMOS for the ACM problem. ....	138
6.3	The process of applying sensitivity analysis to a model. .	144
6.4	The metamodel for expressing uncertainty in models. ...	145
6.5	An example uncertainty model used to capture the uncertainty regarding the impurity of elements in a collection.	146
6.6	Utilising our representation of models to create variants of a model with respect to a set of specified uncertainties....	147
6.7	Three different types of contribution towards the response of CATMOS. ....	150
6.8	Part of the random sampling response distribution for each model in the Pareto front and first non-dominating rank produced by CATMOS. Box width simply denotes the series – the wider boxes being related to the Pareto front. ....	151
7.1	A simple illustration of the effects that the structure of a metamodel has on the representation. The shaded boxes represent redundant feature pairs. The ovals containing numbers are the meta-element identifiers used by the genotype-to-phenotype transformation. ....	159
7.2	Frequency distribution of the total number of meta-classes ( $C$ , $C_c$ , $C_a$ ). ....	164

7.3	Frequency distribution of featureless meta-classes.....	164
7.4	Analysis of meta-features. ....	165
7.5	Violin plots representing the distribution of distances caused by mutation. The white dot in the box plots represents the median. ....	171
7.6	Violin plots for the redesigned experiment. The white dot in the box plots represents the median.....	174
7.7	EG-MOF locality frequency distributions. The dotted line is the culmulative frequency.....	176
7.8	DslModel locality frequency distributions. The dotted line is the culmulative frequency.....	177
7.9	Java locality frequency distributions. The dotted line is the culmulative frequency.....	178
7.10	ATOM locality frequency distributions.....	178
7.11	A surjective function.....	180
7.12	Illustration of surjective redundancy. The ordering of the segments has no effect on the encoded model. The two unshaded feature pairs assign the same reference (from object A to object B) even though their feature value bit differs.	181
7.13	Illustration of overriding a single-valued feature, resulting in a non-coding feature pair. The dashed box represents the feature pair currently being decoded. The shaded boxes represent redundant feature pairs. ....	182
7.14	Proportion of non-coding segments and feature pairs of the four representative metamodels.....	184
7.15	A containment reference reassignment that results in a non-coding segment.....	186
7.16	Repetition of the non-coding redundancy analysis using the alternative reference mapping process. ....	187
8.1	The components and uses of our representation.....	200

# List of Tables

2.1	Case studies used in the literature.....	37
3.1	Parameter settings for the genetic algorithm, genotype-to-phenotype mapping, and fitness metric. ....	52
3.2	The proportion of successful runs (those that find an ‘optimal’ fighter) for the four experiments. The ranges in parentheses are the 95% confidence intervals. Values are rounded to 2 significant figures.....	52
3.3	Parameter settings for locality analysis of the GE-based representation of FDL. ....	57
3.4	Illustration of how semantically equivalent models may be syntactically distinct. ....	59
4.1	The set of default parameters for use in MBMS.....	81
4.2	Parameter settings for the delivery man problem.....	91
5.1	Parameter settings for the genetic algorithm, genotype-to-phenotype mapping, and fitness metric. ....	106
5.2	The results for the four experiments. Where an algorithm was 100% successful at finding an optimal solution, we also list the average number of generations that it took to converge. ....	107
5.3	The 18 sub-experiments we perform to evaluate three factors that affect adaptation in SAF.....	123
5.4	Parameter settings for the workflow.....	124
5.5	Kruskal-Wallis rank sum test for each factor. DF = degrees of freedom. ....	127
5.6	Optimal coded parameter configuration predicted by the central composite design. ....	130
5.7	Kruskal-Wallis rank sum test for each factor of the tuned experimental results. DF = degrees of freedom.....	130
7.1	Structural metrics defined for meta-classes. ....	161
7.2	Structural metrics defined for meta-features.....	162
7.3	The metric scores for four metamodels, and the median scores of the corpus.....	167
7.4	The parameters used to analyse locality with respect to a single metamodel.....	170

7.5	The results of analysing the locality of the representation against the four MUTs. CB = class bit, FSB = feature selector bit, FVB = feature value bit.....	172
7.6	The metamodel-specific ranges used when generating individuals for locality analysis.....	173
7.7	The results of analysing the locality of the representation against the four MUTs. CB = class bit, FSB = feature selector bit, FVB = feature value bit.....	175
A.1	Papers addressing the discovery of models.....	207
A.2	Papers addressing the discovery of model management operations. DT: discovering transformations; DR: discovering refactorings; DM: discovering merges.....	209

For those lost along the way.



# *Acknowledgements*

First and foremost, I'd like to thank my two supervisors, Dr Fiona Polack and Professor Richard Paige. They have both been incredibly supportive throughout my entire PhD; always on hand to provide guidance and encouragement when needed. Particular thanks to them for giving me the opportunities to attend a number of international conferences and the Marktoberdorf Summer School. I will forever be indebted to them for all they have given me. I'd also like to thank my assessors, Professor Jim Woodcock and Professor Mark Harman, for giving me a thorough yet enjoyable viva. I'm very grateful for their feedback and encouragement, and am proud to have been awarded a PhD by them.

Humongous thanks to my family for their unwavering love and support throughout the 9 years I've been at university. Thanks to my partner, İpek, for believing in me and pushing me through the hard times.

Three colleagues that deserve endless praise are Simon Poulding, Dimitrios Kolovos, and Louis Rose. Many thanks to Simon for countless interesting discussions, for providing mini statistics tutorials, and for always encouraging me. Thanks to Dimitris and Louis for the numerous times they helped with an Epsilon-related issue, for encouraging me into interesting collaborations, and for always offering good advice.

I'd also like to extend my gratitude to the people I've lived with during my PhD: Jan Staunton, Christopher Poskitt, David George, Imran Nordin, and Alan Millard have all enriched my life and my PhD in different ways. I'd like to thank Anna Bramwell-Dicks for taking time to explain complicated statistics to me, and Frank Burton for allowing me to apply my research to his CATMOS tool. I'd also like to thank Jason Reich, Sam Devlin, and Gary Plumbridge – who started their PhDs' at the same time as me – and many other PhD students and post-doctoral researchers in the department for providing a network of friendly discourse and support.

Finally, I'd like to thank the EPSRC whose financial support via the Large-Scale Complex IT System grant (EP/F001096/1), has made this thesis possible.





# Declaration

I declare that the work presented in this thesis is my own, except where stated. Chapter 3 and section 7.2 present collaborative work and clearly state the contributions of this author and the collaborators. Parts of this thesis have been previously published in the following:

- [186] **Identifying Desirable Game Character Behaviours Through the Application of Evolutionary Algorithms to Model-Driven Engineering Metamodels** James R. Williams, Simon Poulding, Louis M. Rose, Richard F. Paige and Fiona A. C. Polack. *Proceedings of the Third International Symposium on Search Based Software Engineering, Szeged, Hungary, September 2011.*
- [184] **Generating Models Using Metaheuristic Search** James R. Williams and Simon Poulding. *Proceedings of the Fourth York Doctoral Symposium on Computing, York, UK, October 2011. Best paper award winner.*
- [179] **Sensitivity Analysis in Model-Driven Engineering** James R. Williams, Frank R. Burton, Richard F. Paige and Fiona A. C. Polack. *Proceedings of the Fifteenth International Conference on Model Driven Engineering Languages and Systems, Innsbruck, Austria, September 2012.*
- [182] **Searching for Model Migration Strategies** James R. Williams, Richard. F. Paige and Fiona. A. C. Polack. *Proceedings of the 2012 International Workshop on Models and Evolution. Innsbruck, Austria, September 2012.*
- [181] **Search-Based Model-Driven Engineering** James R. Williams, Simon Poulding, Richard. F. Paige and Fiona. A. C. Polack. *Technical Report YCS-2012-475. Department of Computer Science, The University of York, 2012.*
- [185] **Exploring the use of Metaheuristic Search to Infer Models of Dynamic System Behaviour** James R. Williams, Simon Poulding, Richard. F. Paige and Fiona. A. C. Polack. *Proceedings of the Eighth International Workshop on Models at Runtime. Miami, U.S.A., September 2013.*
- [187] **What do metamodels really look like?** James R. Williams, Athanasios Zolotas, Nicholas Matragkas, Louis M. Rose, Dimitrios S. Kolovos, Richard. F. Paige and Fiona. A. C. Polack.

*Proceedings of the Third International Workshop on Experiences and Empirical Studies in Software Modelling. Miami, U.S.A., October 2013.*

# *I*

## *Introduction*



# Introduction

# 1

## 1.1 Motivation

IT WAS OVER 25 YEARS ago that Frederick P. Brooks, Jr. proclaimed that there will never be one ideal approach to software engineering – no *silver bullet* [17]. To this day he has not been disproven – there has not been a new development that has made software engineering an inexpensive or uncomplicated task. As time has progressed, however, the evolution of the tools, techniques, and theories that we use to build software have enabled us to produce software that is increasing in size and complexity.

Since the beginnings of software engineering, we have tried to address size and complexity, whilst improving productivity and expanding the kinds of systems being built, through the use of *abstraction*. We have progressed from writing software in terms of low-level hardware functionality to writing software in a way in which humans think. From machine code, via structured languages, to functional languages, object-oriented languages, and domain-specific languages: software is arguably becoming easier to produce by raising the level of abstraction at which we develop software from opcodes and operands to concepts from the domain that the software targets.

As well as advancing the languages with which we can develop software, new techniques and methodologies have been proposed to aid the development process. Like other engineering disciplines, today's software engineers will commonly make *models* of the systems that they develop. These models are abstractions of the system being developed and the domain being modelled; they allow developers to not only better devise their system, but also reason about various aspects of the system. It is widely argued that spending time developing a model can enable information to be discovered about a system without suffering the cost of actually implementing it [102, 162, 163]. Models can broadly be classified into five categories related to their purpose: *documentation* – used to describe the problem; *instructions* – prescriptive information about some activity; *explorative* – used to test the consequences of actions without harming the real system; *educa-*

### Contents

1.1 Motivation . . . . .	3
1.2 Research Hypothesis . . . . .	5
1.3 Thesis Contributions . . . . .	6
1.4 Thesis Structure . . . . .	7

*tional* – interactive replacements to real objects/systems for practical or ethical reasons; and *formal* – mathematical descriptions of the problem [102]. Many notations are used for software models, from notebook sketches to mathematical descriptions to graphical notations such as the Unified Modeling Language (UML). Historically, however, models have played a secondary role in software development [162] and are often discarded once development begins whilst becoming out-dated as requirements change or new understanding about the system is gained.

### **1.1.1 Model-Driven Engineering**

*Model-Driven Engineering* (MDE) is an engineering approach that treats models as first-class development artefacts. Models are no longer (just) simple sketches on paper or forgotten designs. Instead, models are the driving force in the software or system development process. Models are tangible, interactive artefacts that are manipulated by *model management operations*. Through a series of automatic *transformations* [164, 31], high-level models can be used to generate the final production system. Focusing development on models allows them to remain useful and up-to-date, but also allow engineers to develop systems at the level of the application domain. This potentially moves the development task from programmer to domain expert and arguably increases productivity, whilst reducing time-to-market [162, 164, 163]. Recent studies have supported these claims. Hutchinson and Whittle recently performed an in-depth analysis on the usage of MDE in industry and discovered that the perceived benefits of using MDE in practice were the ability to quickly respond to changing requirements, improving communication with stakeholders, and increasing productivity, maintainability, and portability [77, 76].

### **1.1.2 Search-Based Software Engineering**

*Search-Based Software Engineering* (SBSE) is an engineering approach that focuses on reformulating software engineering problems as optimisation problems and applying metaheuristic optimisation techniques to discovering (near) optimal solutions to the problem [65, 28]. The interest in SBSE has been growing in recent years [67], with it being successfully applied to all aspects of the software development life-cycle [65, 62, 67]. To reformulate a software engineering problem as an optimisation one needs to perform two actions. Firstly, define a *representation* of the problem's solution space that can be used within the metaheuristic technique; and secondly, devise a *fitness function* to measure the quality of each candidate solution [28]. Commonly, complex solution spaces are encoded in simplified forms that are more amenable to optimisation techniques. These simplified forms are translated into the pure representation of the solution to be evaluated by the fitness

function. The output of the fitness function is used to guide the metaheuristic technique towards optimal solutions.

### 1.1.3 Combining MDE and SBSE

In recent years, as MDE has matured, researchers in the MDE community have started to make use of SBSE techniques. Notable research comes from Betty H.C. Cheng whose work (summarised in [23]), focusing on the fields of dynamically adaptive systems and uncertainty, has included evolving target system models at runtime [136], evolving behavioural models of systems [59, 58], and evolving goal models with relaxed constraints so as to accommodate uncertainty [137]. In contrast, Marouane Kessentini has addressed the challenge of searching for model management operations, using example mappings between models to discover model transformation rules [85, 87, 86, 43, 44, 10, 114], and merging sets of models whilst reducing conflicts [88]. These two researchers (and their colleagues) are the earliest known attempts at applying SBSE techniques to discovering or optimising both models and model management operations, and interest in this area is growing [68]. Currently there exist no standard practices to adapting SBSE techniques to MDE problems, and there are few common case studies or examples with which to experiment or compare approaches. The early work in this area, however, shows promise that SBSE can benefit MDE as it has traditional software engineering.

## 1.2 Research Hypothesis

The research hypothesis addressed in this thesis is as follows:

*A generic, search-amenable representation of models would enable the wealth of existing research into SBSE to be applied to a broad range of problems found in the MDE domain.*

Model-Driven Engineering has been shown to increase productivity in software development projects [77, 76, 113], and is seen by advocates as the next evolutionary stage in software development. Search-Based Software Engineering have been successfully applied to optimising all aspects of the software engineering life-cycle. However, there currently exists no standard way to apply SBSE techniques to MDE problems. A representation that can express any model, and is amenable to many well-defined SBSE techniques, would enable MDE practitioners to benefit from the wealth of knowledge being produced by the SBSE community and more easily apply SBSE techniques to their MDE-related problems. Models are at the heart of MDE; the ability to automatically discover, optimise, and evaluate models and the operations that manipulate them, would greatly improve the MDE practitioners toolbox.

The objectives of this thesis, therefore, are:

- To identify existing research that combines SBSE and modelling, and propose extensions where their synergy would be fruitful.
- To design and implement an encoding that can represent any (and all) model(s) that conforms to a given metamodel, and that is applicable to existing SBSE techniques.
- To design and implement a model-driven SBSE framework that uses this encoding and provides standard SBSE algorithms.
- To use the model representation to address a set of known challenges in the MDE domain.
- To evaluate properties of the representation that have been shown to be important for evolutionary search, and use the knowledge gained from the evaluation to provide guidance to users of the representation and to propose improvements to the representation.

### 1.3 *Thesis Contributions*

The primary contributions made in this thesis are summarised below.

- The definition of a novel generic representation of models that is amenable to a wide range of existing metaheuristic optimisation techniques.
- The demonstration of the feasibility of using the representation to solve in-scope problems:
  - Discovering optimal behaviour models;
  - Extracting a model of runtime system behaviour for use in self-adaptive systems;
  - Applying *sensitivity analysis* to model management operations to analyse the uncertainty present in models.
- A large-scale analysis of the structure of metamodels, providing detailed insight into the ways in which practitioners commonly build their metamodels.
- The identification of a set of representative metamodels, taken from the corpus, that act as benchmarks for evaluating representations of models.
- The empirical analysis of two important properties of the representation.

The secondary contributions made in this thesis are summarised below.



- A prototype *Grammatical Evolution*-based prototype representation of models that demonstrated the feasibility of combining modelling and search and inspired the development of the generic model representation.
- The implementation of the model representation using MDE technologies.
- The implementation of a metaheuristic search framework, built using state-of-the-art MDE technologies, where all aspects of the framework are expressed as models, and the core algorithms are defined as model management operations.
- An interactive, web-based visualisation of the search algorithm, made possible through the use of MDE technologies and by capturing the progress of the search algorithm in a model.
- A model-driven approach to runtime adaptation which utilises metaheuristic search for decision making.
- The identification of three areas where uncertainty arises in MDE.
- A framework and methodology for the application of sensitivity analysis to models.
- A detailed plan for future empirical analysis of the representation, or any similar representation of models.

## 1.4 Thesis Structure

Chapter 2 presents a review of the related literature. Firstly, we provide the necessary background information related to both MDE and SBSE that is necessary to understand the thesis. We then present a review of the existing work that applies SBSE to MDE problems, examining how each of the addressed MDE problems has been adapted for use with SBSE techniques. In particular, we group the existing literature into two categories: those that discover or optimise models, and those that discover or optimise model management operations. Furthermore, we enumerate the case studies that have been used in the literature as a new field needs common examples with which they can compare and contrast their work.

Chapter 3 describes our prototype search-amenable representation for MDE models. The prototype focuses on representing textual models, and utilises a grammar-based metaheuristic search technique known as *Grammatical Evolution* [151, 127]. We use the representation to discover optimal models of characters in a video game, demonstrating the efficacy of the approach and

of the ability to define a generic representation of models. Limitations of the prototype representation are discussed and empirically analysed.

Chapter 4 introduces our generic, search-amenable representation of models. We describe the key components of the representation, and present *Crepe* – an implementation of the representation that targets a well-known and widely-used modelling platform. We also describe *MBMS* – a model-based metaheuristic search framework, built using state-of-the-art MDE technologies, that utilises *Crepe* as its representation. *MBMS* and *Crepe* provide a platform that lowers the entry barrier for MDE practitioners to applying SBSE techniques to their problems. *Crepe* and *MBMS* are illustrated using an simple MDE implementation of the Travelling Salesment Problem, and limitations of the representation are discussed.

Chapter 5 utilises the representation to discover optimal models. We apply *Crepe* to the video game problem addressed by our prototype representation, demonstrating its efficacy whilst also illustrating an issue with the representation. We then use *Crepe* to tackle the challenge of discovering a model of system behaviour based on sensory information, and present a model- and search-based approach to runtime system adaptation. We perform a principled experiment to analyse the adaptation approach on the video game problem.

Chapter 6 addresses the inherent *uncertainty* that is found in models. Uncertainty in models can lead to unexpected behaviour, and slow down the development process. We describe three areas where uncertainty can arise in MDE and discuss the effects that these uncertainties can have. We define a framework for applying *sensitivity analysis* to models and show how this can be used to qualify the output of model management operations and thus improve both the confidence in, and understanding of, models. This chapter also highlights the fact that the representation can be used for tasks that don't require search.

Chapter 7 presents an evaluation of the representation. In particular, we analyse two important properties of search-based representations which have been shown to influence the performance of evolutionary search: *locality* and *redundancy*. To enable this analysis, we perform a systematic examination of a large corpus of metamodels, and propose a set of benchmark metamodels that can be used to evaluate representations similar in nature to ours. Furthermore, we propose a detailed plan for future analysis of the representation, or other representations of its kind.

Chapter 8 summarises the contributions made in this thesis and discusses potential avenues of research that build atop of the work presented here.

# Literature Review

# 2

THIS THESIS DESCRIBES a novel encoding of Model-Driven Engineering (MDE) models that is designed to be amenable to many standard Search-Based Software Engineering (SBSE) techniques. In this chapter, we provide overviews of MDE and SBSE that are necessary to understand this thesis, and present a critical review of existing research in the area of combining MDE and SBSE.

**Chapter Structure** Section 2.1 presents background information on MDE. In particular, we describe the key *principles* of MDE – models, modelling languages, and automated model management – and the key *practices* of MDE – the modelling and model management technologies used. Section 2.2 presents background information on SBSE. We overview two popular metaheuristic search-based optimisation techniques, discuss the considerations needed when defining a search-amenable representation for a problem, and describe the process of reformulating a software engineering problem as an optimisation problem. Section 2.3 reviews existing work that utilises metaheuristic search-based optimisation techniques for addressing MDE problems, highlighting the ways in which the MDE problems were represented for use with search. We also present work that proposes opportunities where MDE and SBSE can be combined, and enumerate the case studies that were used in the literature to enable a set of benchmark examples to be devised for the field.

## Contents

2.1 Model-Driven Engineering . . .	9
2.2 Search-Based Software Engineering . . . . .	19
2.3 Combining MDE and SBSE . . . . .	26
2.4 Summary . . . . .	36

## 2.1 Model-Driven Engineering

*Models* have been used in other engineering disciplines for years. They allow engineers to understand and reason about the artefact being constructed, and provide an effective means of communication with stakeholders [163, 162]. In software engineering, however, models have historically played a secondary role, even though the benefits of using models in software engineering are potentially greater than in any other engineering discipline [162]. Modelling complex software engineering problems at a higher level of abstraction – i.e. using domain concepts rather than implementation concepts – makes the modelling process *easier* as

the modeller need not be concerned with implementation detail [162]. Moreover, the process is more *maintainable*: as requirements change, the domain model can easily be updated, which has the added benefit of enabling a domain expert to become the developer of the system [162, 16]. From these domain-level models systems can be automatically generated, thus increasing productivity and reliability [162]. Thus, the two key principles behind MDE are the primary use of models, and the automated management of models.

In this section we present the background on MDE that is required to understand this thesis. Section 2.1.1 overviews the key principles of MDE: models and automated management of models; and section 2.1.2 describes the key *practices* of MDE: introducing the technologies used for modelling, and for model management. Finally, section 2.1.3 discusses some of the promised (and realised) benefits of MDE as well as some of its shortcomings.

### 2.1.1 Key Principles of MDE

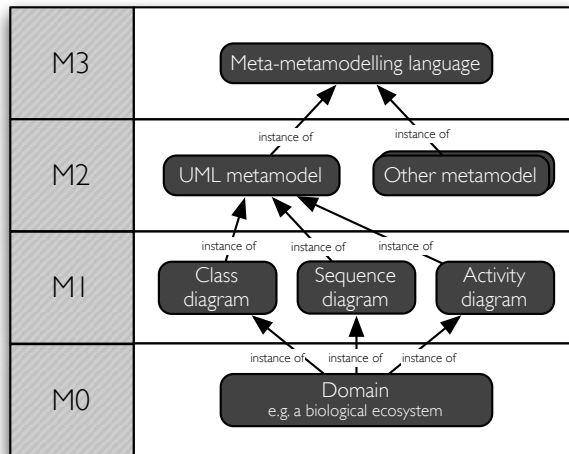
**Models** Models in MDE are required to have a formal definition of their structure. Unstructured models, such as sketches and natural language, are useful for communication but cannot (easily) be used with automated processes due to their ill-defined semantics and syntax [91]. Most commonly in MDE, a model's structure is defined by a *metamodel* – another model that specifies the language, concepts, and constraints available to a model – but this need not be the case. As Paige and Rose [131] point out:

“MDE [...] require(s) the construction, manipulation and management of well-defined and structured models - but you don't have to make use of OMG standards, or a particular style of development to do it.”

For example, a model's structure could be defined using an abstract syntax tree or some form of schema. The key point is that there is some formal notion of structure. In this thesis, we will only address models that use metamodels to define their structure. A model that uses only the concepts defined by a metamodel and doesn't break any of its constraints, is said to *conform* to that metamodel [130]. Metamodels can be also referred to as *modelling languages*.

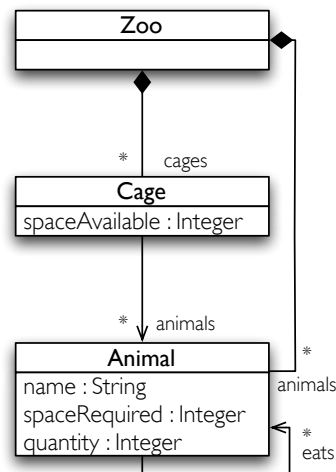
The Object Management Group<sup>1</sup> (OMG) have defined a standard for a metamodelling architecture, called the *Meta-Object Facility* (MOF) [120]. The architecture is composed of four layers – illustrated in figure 2.1. The top layer, M<sub>3</sub>, provides a metamodelling language (i.e. a meta-metamodel) for specifying metamodels in the second layer, M<sub>2</sub>. The *Unified Modeling Language* (UML) [122], the de facto modelling language, is an example of an M<sub>2</sub> metamodel. The third layer, M<sub>1</sub>, contains models that conform to metamodels in M<sub>2</sub>, for example UML class diagrams. Finally,

<sup>1</sup> www.omg.org



**Figure 2.1:** Illustration of the MOF metamodeling architecture. Adapted from [16] and [120].

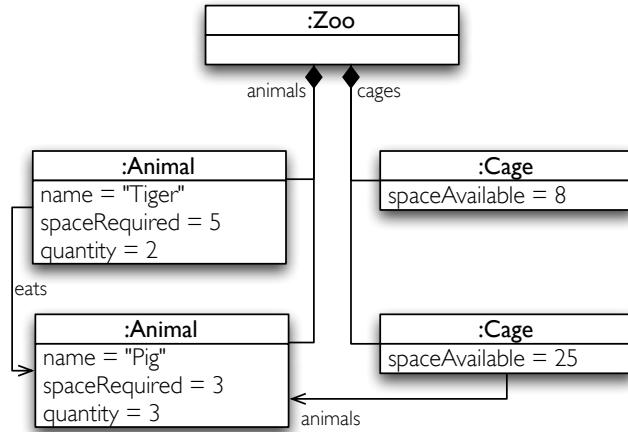
M0 is the object layer - i.e. the real-world problem being modelled. MOF is unable to express all constraints that are commonly needed when defining a metamodel – it focuses on simple structural constraints only. To define more complex constraints, such as invariants on all instances of a particular meta-class, another language is needed. Commonly, this is the *Object Constraint Language* (OCL) [61]. A metamodeling architecture, such as MOF, enables interoperability between MDE tools. The OMG defines an XML-based persistence format for models, *XML Metadata Interchange* (XMI), which allows MOF-compliant tools to interact with any model or metamodel specified in this format.



**Figure 2.2:** A metamodel for a simple language which allows zookeepers to assign animals to cages.

For illustration purposes, figure 2.2 shows a metamodel, defined in MOF, that specifies a simple modelling language for designing the layout of a zoo. Each box is a *meta-class* and the lines represent *references* between meta-classes. References and attributes of a meta-class are collectively known as *meta-features*. In figure 2.2, a Zoo references Cages and Animals. The black diamonds on the relations represent *composition*, meaning that cages and animals can only exist if they are *contained* by a zoo. If the

**Figure 2.3:** A example model which conforms to the metamodel in figure 2.2.



zoo object is deleted all cages and animals that it is composed of are also deleted. The asterisks on a reference specifies the *multiplicity* of the reference. In this case, a zoo can be composed of *many* (\*) cages and animals. Alternatively, explicit values (e.g. 5) or ranges (e.g. 2..5) can be specified. If a multiplicity value isn't given, the default value is 1. Figure 2.3 demonstrates an instance of a model that conforms to the zoo metamodel. The model is expressed using *object diagram* syntax [122]. Each box is an object: the colons specify that the object is an instance of a particular meta-class. The reference names relate to those defined in the metamodel. It is worth saying that although we use object diagram syntax here, we do so only because it is a familiar syntax. Commonly, developers will define one or more explicit concrete syntaxes for their modelling languages. For the zoo example, we could perhaps define a textual syntax or even define a graphical syntax that physically looks like a zoo and allows users to drag and drop cages and animals. The tools which enable the development of concrete syntaxes, and much more, are discussed in section 2.1.2.

**Automated Management of Models** Historically, models in software engineering have commonly been used only for design or documentation and often become outdated as requirements change and the development progresses. In MDE, however, the formally-defined structure of models enables models to be processed by programs known as *model management operations* (MMO). These operations are what automatically converts a model from being only a piece of documentation into a being the artefact driving the development of a system. We briefly summarise the most common MMOs.

**Model transformation** A model transformation is an operation that translates a model, or set of models, into a new representation. Czarnecki and Helsén [31] list a number of applications for model transformations, which include: generating lower

level models from high level models, generating code, refactoring models, and reverse engineering models from existing systems. There are two main categories of model transformation: model-to-model transformations, and model-to-text transformations. Model-to-model transformations operate on two (sets of) models: the *source* model(s) – i.e. the model(s) to which transformation will be applied, and the *target* model(s) – i.e. the resulting model(s). Model-to-model transformations have a number of properties: a *direction* – horizontal or vertical, relating to the levels of abstraction of the source and target model [108]; a *type* – endogenous or exogenous, relating to whether the source and target model conform to the same metamodel or not [31]; a *multiplicity* – relating to the number of input models and the number of output models (e.g. a one-to-many transformation might transform a *platform-independent model* into many different *platform-specific models* [108]). Mens and Van Gorp [108] have constructed a *taxonomy* of model transformation with the aim to aid the decision of which model transformation approach to use for a particular kind of problem.

Model-to-text transformations are commonly used to generate code from a (set of) models(s), but can also be used to generate other textual artefacts, such as documentation (e.g. [143]), formal specifications for verification purposes (e.g. [183, 3, 35]), or to serialize a model into a format for interchange (e.g. XMI) or storage (e.g. in a database) [131]. Additionally, there are transformations that parse a textual notation into a model (e.g. Xtext<sup>2</sup> [37], EMFText<sup>3</sup> [70]). This illustrates a commonly misunderstood point: models in MDE need not be of a graphical nature.

<sup>2</sup> [www.eclipse.org/Xtext](http://www.eclipse.org/Xtext)

<sup>3</sup> [www.emftext.org](http://www.emftext.org)

Model transformation was once seen as the *heart and soul of model-driven development* [164], though it is now realised that transformation is not the only important operation that can be applied to models, and in fact the heart and soul is the automated aspect of interacting with models [131].

*Model validation* Models often need to satisfy certain constraints. Problems can arise when models are *incomplete* (missing information) or *inconsistent* [99, 91]. Models can be inconsistent with respect to their syntax – i.e. they do not conform to their metamodel – or their semantics – i.e. they do not satisfy semantic constraints [38]. Syntactic and semantic consistency relate to *intra-model* consistency, i.e. they are properties of a single model [38]. *Inter-model* consistency relates to cases where different models (perhaps different views of the same system) capture the same information in conflicting ways [38]. Due to the vital role that models play in MDE, inconsistency needs to be discovered and addressed otherwise it can propagate through a chain of transformations and arise in the deployed system.

*Model comparison* Comparing two or more models is an essential prerequisite for a number of model-related tasks. These include: calculating the differences between models, merging a set of models, and testing that the output of a model transformation was as expected. Comparing models is challenging due to their structure. It is not possible to compare models based on their underlying XMI representation as two XMI representations of the same model can have many differences (e.g. different element identifiers, or different orderings of elements) [173]. Existing comparison methods include: signature-based techniques (e.g. [51]) where element identities are computed at comparison time; graph-based approaches (e.g. [173, 191]) where models are treated as typed attribute graphs; or using a task-specific language for precisely defining the matching rules between elements (e.g. [90, 180]).

We now look at some of the technologies that have been developed to support the principles described above.

### 2.1.2 Key Technologies of MDE

This section describes some of the existing technologies that enable MDE practices. Specifically, we focus on the technologies that are used throughout this thesis, namely the *Eclipse Modeling Framework* and *Epsilon*.

<sup>4</sup> [www.eclipse.org](http://www.eclipse.org)

<sup>5</sup> [www.eclipse.org/emf](http://www.eclipse.org/emf)

<sup>6</sup> [www.eclipse.org/modeling/gmp](http://www.eclipse.org/modeling/gmp)

<sup>7</sup> <http://www.eclipse.org/graphiti>

<sup>8</sup> <http://www.eclipse.org/modeling/emft/emfatic>

**Modelling Technologies** The Eclipse Foundation<sup>4</sup> have implemented their own metamodeling architecture that aligns with MOF's four-layer architecture (see previous section). The *Eclipse Modeling Framework*<sup>5</sup> (EMF) [170] has a MOF-equivalent metamodeling language called *Ecore* and provides stable and well-maintained tool support for modelling activities, such as a graphical editor for defining metamodels and tools for automatically generating model editors from a metamodel. EMF is presently the de facto modelling framework and has a very active community that build on top of the core framework to provide a suite of interoperable tools that aid model development. For example, both the *Graphical Modeling Framework*<sup>6</sup> (GMF) and Graphiti<sup>7</sup> provide tool support for generating and customising graphical editors for models, Xtext and EMFText enable users to define textual modelling languages, and Emfatic<sup>8</sup> provides a textual language (built using Xtext) for developing Ecore metamodels. For illustration, the metamodel from figure 2.2 is expressed using Emfatic in listing 2.1.

```

1 package zoo;
2
3 class Zoo {
4     val Animal[*] animals;
5     val Cage[*] cages
6 }
7

```



```

8 class Cage{
9   attr int spaceAvailable;
10  ref Animal[*] animals;
11 }
12
13 class Animal {
14   attr String name;
15   attr int spaceRequired;
16   attr int quantity;
17   ref Animal[*] eats;
18 }

```

**Listing 2.1:** *The zoo metamodel expressed in the Emfatic language. The `val` keyword represents a composition reference and `ref` represents a standard reference.*

**Model Management Technologies** Transformations were originally written in general purpose languages; modelling tools, such as Rational Rose<sup>9</sup>, provided APIs to manipulate models [164]. General purpose programming languages, however, are not suited to writing transformations because they don't capture the necessary level of abstraction meaning that specialised languages are required [164, 94]. More recently, a number of languages have been developed for the sole purpose of specifying transformations. These include the Atlas Transformation Language (ATL) [79], the OMG's QVT [121], and the Epsilon Transformation Language (ETL) [94].

<sup>9</sup> <http://www-03.ibm.com/software/products/us/en/ratirosefami/>

ETL is part of the family of model management languages provided by *Epsilon*<sup>10</sup> [97, 91] – a platform for model management. Other languages provided by Epsilon to support model management include: the Epsilon Generation Language (EGL) [143] – a model-to-text transformation language; the Epsilon Validation Language (EVL) [93] – an OCL-inspired language for model validation; the Epsilon Comparison Language (ECL) [90, 180] – a rule-based language for precise model comparison; the Epsilon Merging Language (EML) [95] – a rule based language for merging multiple models; and the Epsilon Pattern Language (EPL) – a rule-based language to detect patterns in models. Moreover, Epsilon provides tool support for unit testing MMOs [53], rapidly prototyping GMF-based graphical editors [96], and monitoring inter-model references in EMF models [142].

<sup>10</sup> [www.eclipse.org/epsilon](http://www.eclipse.org/epsilon)

All of the Epsilon languages build atop and reuse a common general purpose model management language – the Epsilon Object Language (EOL) [92]. EOL is an imperative language that reuses familiar syntax and expressions from OCL but addresses many of OCL's flaws [92]. We briefly describe EOL with a simple example, as it is used throughout this thesis. Furthermore, in understanding EOL, one will be able to gain an understanding of the other Epsilon languages.

Listing 2.2 illustrates many features of EOL using the Zoo example. The operation transfers animals from an existing zoo to a newly created zoo. A novel feature of EOL is that it can man-

age multiple models simultaneously. In our example we have one input model and one output model. The input model is an existing zoo model, and we output a new zoo model. To distinguish between objects in different models, we use a prefix before the meta-class name: in this case YRK represents the new zoo being created, and LDN represents the existing zoo from which animals will be transferred. EOL allows you to create model elements: lines 2–4 create new objects inside the YRK model. Line 3 and the `addAnimal` operation illustrates how one can define operations for given meta-classes. Any Zoo instance can invoke the `addAnimal` operation. The `self` keyword (lines 18 and 30) represents the object that the operation has been invoked upon. Features (attributes and references) are accessed using the dot notation. References are treated as collections, and so the `add` operation is used to add an object to a reference (e.g. line 30).

```

1 // Create the new zoo and add some animals and a cage
2 var zoo : Zoo = new YRK!Zoo;
3 var zebras = zoo.addAnimal("Zebra", 5, 2, Sequence{});
4 var lions = zoo.addAnimal("Lion", 6, 1, Sequence{zebras});
5 var cage = zoo.addCage(30, Sequence{zebras, lions});
6
7 // Now transfer some animals from LDN
8 var ldnZoo : LDN!Zoo = LDN!Zoo.all.first();
9 var tigers = ldnZoo.animals.selectOne(animal | animal.name
10 == "Tiger");
11 zoo.animals.add(tigers);
12 cage.animals.add(tigers);
13
14 operation YRK!Zoo addCage(spaceAvailable : Integer, animals
15 : Sequence) : Cage {
16     var cage : YRK!Cage = new YRK!Cage;
17     cage.spaceAvailable = spaceAvailable;
18     cage.animals.addAll(animals);
19
20     self.cages.add(cage);
21
22     return cage;
23 }
24
25 operation YRK!Zoo addAnimal(name: String, spaceRequired :
26 Integer, quantity : Integer, eats: Sequence) : Animal {
27     var animal : YRK!Animal = new YRK!Animal;
28     animal.name = name;
29     animal.spaceRequired = spaceRequired;
30     animal.quantity = quantity;
31     animal.eats.addAll(eats);
32
33     self.animals.add(animal);
34
35     return animal;
36 }

```

**Listing 2.2:** A simple EOL program to transfer animals to a new zoo.

Line 8 creates a variable that represents the Zoo object in the LDN model. There is no way to identify the existing zoo object without referring to its meta-class. The `all` operation returns the collection of all instances of the specified meta-class. In this case there is only one zoo object in the LDN model, and so we select it from the resulting collection using the `first` operation. EOL sup-

ports first-order logic operations such as `select`, `collect`, `exists`, and `forall`. On line 9, we use the `selectOne` operation to return the first animal object whose name is "Tiger". We then add this object into the YRK zoo object's `animals` collection. Due to the rules of object containment and the fact that the `animals` reference is a composition, by adding the tiger to the YRK model, it removes it from the LDN model.

The most powerful aspect of the Epsilon platform is that it is agnostic to metamodelling technologies. Epsilon provides lots of integration with the Eclipse platform, but can be used totally independently. It provides a *connectivity layer* which allows different metamodelling technologies to easily integrate with Epsilon. Furthermore, models from different technologies can be used within the same model management operation. At the time of writing, Epsilon supports EMF models, XML files (with a schema), XML files (without a schema) [98], CSV files, Bibtext files, MetaEdit models, spreadsheets [50], and Z specifications.

### 2.1.3 *Benefits and Shortcomings of MDE*

In 2006, Schmidt noted that it was difficult to find literature that assesses the benefits of MDE in practice [159]. Since then a number of studies of the benefits and shortcomings of MDE have been undertaken. One example of an experience report written by industrial practitioners is by Weigert and Weil [177]. They published their experiences of using MDE for fifteen years at Motorola for developing trustworthy computing systems. They found that using MDE increased the quality and reliability of their software, whilst also increasing the productivity of their engineers. Code generators were the main reason for success here – much effort was placed on developing robust generators, which allowed the design models to become more abstract and therefore easier to produce and analyse. Furthermore, they found that using MDE enabled them to develop efficient working team structures, as different concerns were separated between the design models and the code generators [177].

In 2008, Mohagheghi et al. [113] performed a systematic review of papers published between the years 2000 and 2007 that relate to experiences with using MDE in industry. They found that MDE had been applied in a wide range of domains, including telecommunications, financial organisations, defence organisations, and web applications. The reasons stated for using MDE were the often cited benefits of MDE: e.g. increasing productivity by automating labour intensive or error prone tasks, improving quality of software, and increasing maintainability of software. In practice, however, Mohagheghi et al. found that the view was divided on productivity, with a number of people experiencing a reduction in productivity. This reduction was largely due to the immaturity of supporting MDE tools and the high cost of

adoption. The general consensus, however, was that there was an increase in the quality of the produced software.

Hutchinson et al. [77] performed an in-depth empirical study into the use of MDE in industry. Using questionnaires, interviews, and on-site observations, they aimed to discover how MDE is being used and what the perceived benefits or drawbacks are. They found that, overall, MDE was seen to increase productivity and maintainability, showing in particular a time reduction in responding to requirements changes. Code generation was seen as a key activity for improving productivity, and that the increase in productivity outweighed the cost of incorporating generated code into existing systems. The need for significant training was seen as a drawback, and many organisational factors affect the success or failure of adopting MDE. The role of organisational factors is further addressed in [76], which focuses on the experiences of three commercial organisations adopting MDE. They found that a wide range of organisational, managerial and social factors affect the success of adopting MDE. These include: the way in which MDE is introduced into the company; the commitment of the organisation to make MDE work, integrate it with their development processes and motivate their employees to use it; and the way MDE is positioned with respect to the business's focus [76].

Despite the empirical evidence produced by Hutchinson et al. [77, 76] and Mohagheghi et al. [113], the adoption of MDE in industry has been slow [163]. Petre [133] focused specifically on the use of UML in industry by interviewing 50 software engineers in 50 companies to see how (or if) UML is being used. Petre identified five patterns of usage: not at all, retrofit to satisfy customers/stakeholders, automate code generation, informally for as long as useful, and wholeheartedly. The majority of respondents fit into the first category – not using UML at all. Only 3 out of the 50 respondents used UML for code generation. In support of this, Hutchinson et al.'s study found that the majority of practitioners used domain-specific modelling languages rather than the UML [77].

France and Rumpe [49] laid out a research roadmap that lists three categories of challenges faced in MDE:

*Modelling language challenges:* MDE researchers need to provide robust support for developing, utilising, and analysing models. In particular, we need to address the challenges of *abstraction* – i.e. aiding the development of problem-level models – and *formality* – i.e. specifying the semantics of modelling languages [49].

*Separation of concerns challenges:* Multiple models can be used to specify different views on a system, particularly for large and complex systems. MDE needs to support multiple, potentially heterogeneous, models [49].

*Model manipulation and management challenges:* Rigorous support for model composition and decomposition, synchronisation to maintain relationships among models, analysing transformations, and utilising models at runtime is required [49].

Selic [163] groups France and Rumpe’s challenges into categories that tackle the industrial adoption of MDE: *capability* challenges, *scalability* challenges, and *usability* challenges. Regarding capability challenges, Selic argues that we need advances in: understanding modelling language design and specification; supporting synchronisation and validation of multiple models; providing semantic theory for model transformations; and developing static and dynamic methods for validating that models are correct [163]. To address scalability issues we need to support incremental modelling and facilitate efficient operations on large or fragmented models [163]. Finally, usability challenges relate to the tool support for MDE processes: currently they are too complex [163].

## 2.2 *Search-Based Software Engineering*

Search-Based Software Engineering (SBSE) [65, 28, 62] is a software engineering approach that treats software engineering problems as *optimisation* problems. SBSE is based on the observation that it is often easier to *check* that a candidate solution solves a problem than it is to *construct* a solution to that problem. Indeed, solutions to some software engineering problems may be theoretically impossible or practically infeasible; SBSE techniques can help discover acceptable solutions to these problems [65]. Applications of SBSE have been growing in scope and sophistication over the last 10 years, with researchers demonstrating success in applying search-based optimisation techniques to a variety of problems that span the breadth of software engineering [62]. Since Harman and Jones coined the term in 2001 [65], hundreds of papers have been published on applying SBSE to software engineering problems, so much so that numerous survey papers have been published including: search-based software testing [106, 107, 2], search-based software maintenance [132], software architecture optimisation [1], software design optimisation [135], as well as reviews of the entire field [28, 62, 67].

In this section we describe the key concepts of SBSE, briefly introducing metaheuristic search, and focusing on the key areas of relevance for this thesis: representing and reformulating problems for search. Section 2.2.1 overviews *metaheuristic search-based optimisation* – the tool with which SBSE tackles software engineering problems. Section 2.2.2 discusses how problems can be repre-

sented effectively for use with metaheuristic search-based optimisation algorithms. Finally, section 2.2.3 discusses how to formulate a software engineering problem as an optimisation problem.

### 2.2.1 *Metaheuristic Search*

*Metaheuristic search algorithms* are optimisation algorithms that iteratively improve upon a solution to discover a (near) optimal solution to a problem [11]. They aim to efficiently explore the *solution space* of a problem – i.e. the (possibly infinite) set of all possible solutions to a problem. Metaheuristic search techniques can be applied to a broad range of problems – as opposed to heuristic techniques which are designed to address specific problems. Each candidate solution examined by a metaheuristic search algorithm is evaluated to assess how “close” they are to solving the problem. Some candidate solutions will be “closer” than others: this information is used to guide the search over the solution space toward an optimal solution.

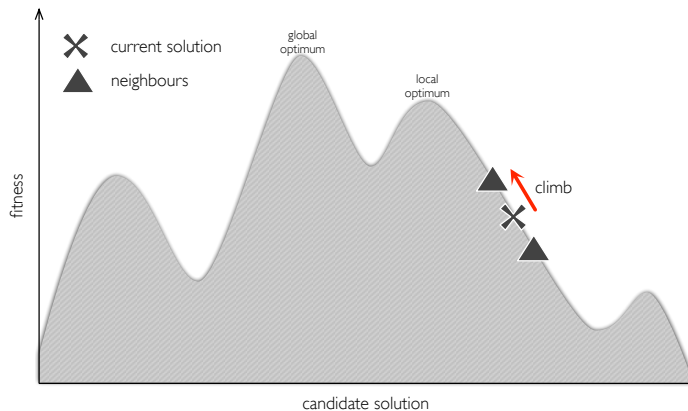
Metaheuristic search algorithms optimise either a single solution or a set of solutions, known as a *population*. Each candidate solution is evaluated to calculate its *fitness* – a measure of how close that solution comes to solving the problem. Algorithm-specific operators are applied to the solution(s) in an attempt to improve the fitness. This process repeats until either a satisfactory solution has been found, or until a pre-defined amount of time passes.

We briefly overview two categories of metaheuristic optimisation techniques: local search algorithms and population-based algorithms.

***Local Search*** Local search, or *single-state*, techniques operate upon a single candidate solution, attempting to improve its quality until an optimal solution has been found, or a maximum number of iterations has been reached. The operator commonly used in local search techniques is the *neighbourhood function*. This function returns a set of candidate solutions that are structurally similar to the current solution. At each iteration of the local search technique, one of the neighbouring solutions is selected to become the current solution, and the process repeats. The most well known example of a local search technique is *hill-climbing*.

Arguably one of the simplest metaheuristic technique, hill-climbing is the process of examining *neighbouring* solutions for a fitter candidate, and once found the the fitter candidate becomes the current solution [103, 28]. The algorithm “climbs” towards a peak in the fitness landscape. Figure 2.4 illustrates this process on a simple fitness landscape.

Hill-climbing works well when the fitness landscape is simple, but it is unable escape from a *local* optimum in the case of a more rugged fitness landscape because it always moves to fitter solu-



**Figure 2.4:** A simple illustration of hill-climbing on a rugged landscape. In this example, only a local optimum will be reached.

tions and won't accept a weakening move. This means that the fittest solution found may not be the global optimum, as can be seen in figure 2.4.

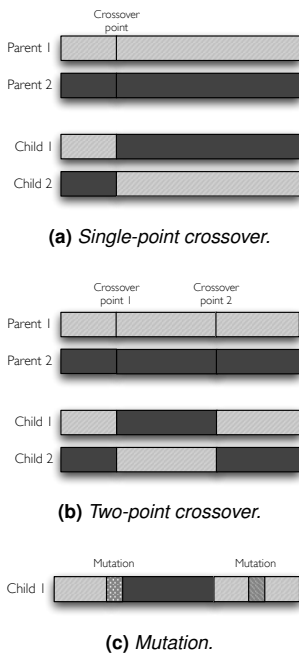
The starting point (*initialisation procedure* [103]) of the hill-climbing algorithm can either be selected at random, or by using some pre-existing knowledge of the solution space. To overcome the issue of local optima, variants of the hill-climbing algorithm have been devised. For example, *hill-climbing with random restarts* runs the hill climbing algorithm  $n$  times from different starting points and returns the best solution found [103]. An alternative approach is taken by the *simulated annealing* technique which allows less fit solutions to be selected based on some gradually reducing probability [28, 103]. This means that initially, worsening moves are accepted but as time increases it becomes less likely to select a worsening move. This process enables the algorithm to escape local optima.

**Population-Based Search** An alternative to single-state algorithms, are techniques that optimise a population of solutions. Commonly population-based algorithms take inspiration from evolution in nature. These algorithms iterate through *generations*: the members of the current generation's population are used to produce the population in the subsequent generation.

The most widely known population-based algorithm is the *genetic algorithm* (GA) [56, 57]. GAs select the best individuals from the population to be combined and mutated to develop the population of the next generation. A typical GA takes the following steps:

1. *Initialise* the population.
2. *Evaluate* the fitness of the population
3. Repeat until termination criteria is met:
  - (a) *Select* the fittest individuals from the population.
  - (b) *Breed* the fittest individuals to produce a new population.

(c) Evaluate the fitness of the new population.



**Figure 2.5:** Illustration of the crossover and mutation genetic operators used in genetic algorithms.

There are numerous approaches to selecting individuals from the population. The most simple would be to simply select the fittest  $n$  solutions. This can cause the population to converge quickly, and so other, stochastic, techniques have been proposed. In *Roulette wheel* selection, individuals with higher fitness have a greater probability of being selected. In *tournament* selection, an individual is chosen at random from the population and ‘pit’ against another random opponent. The individual with the highest fitness is determined the ‘winner’. A fixed number of ‘fights’ take place, with the winner being kept after each round. The individual remaining at the end of the tournament is kept. The tournament is run  $n$  times to select the population for reproduction.

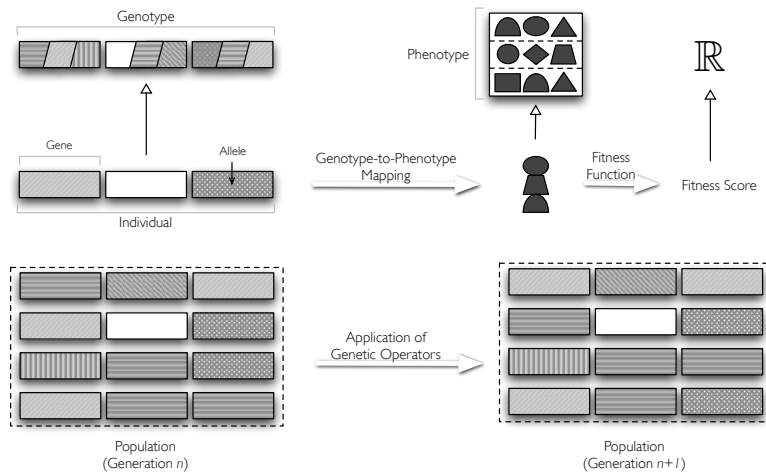
For genetic algorithms, the breeding phase commonly consists of the application of two *genetic operators*. The *crossover* operator takes two *parents* from the population and combines them to produce two children. The most common crossover operators are *single-point crossover* and *two-point crossover*, illustrated in figure 2.5. The *mutation* operator is then applied to the set of children produced through crossover. It stochastically makes slight changes to the children in order to maintain diversity in the population, so that the population doesn’t converge on a single area of the solution space. Depending on the particular encoding that the search algorithm is being applied to, different mutation operators are defined. As its goal is to create variants of an individual, the mutation operator in combination with selection can be viewed as evoking *continual improvement* to the population’s fitness – it is a form of hill-climbing [57]. Selection combined with crossover, however, can be seen as inducing *innovation*, as good solutions are combined into (potentially) better solutions [57]. Goldberg [57] treats selection as a genetic operator. Here, we refer to mutation and crossover as *adaptation operators* and all three as genetic operators. The goal of selection is to pick the best of a generation. Without applying any of the adaptation operators we would instantly reach a local optimum; the goal of the adaptation operators is to help discover new (and possibly better) individuals [57]. Often the fittest solutions in a population, the *elite*, are kept unaltered in the new population. This encourages the elites to become parents and therefore attempt to improve the fitness of the children, though this can cause the population to prematurely converge on a sub-optimal solution [103]. There are a variety of possible termination conditions: the GA may run for a set number of generations, or until a desired fitness is attained, or until solutions converge and no further improvement in fitness is detected.



### 2.2.2 Representations for Search

Each metaheuristic search-based optimisation algorithm is defined with respect to an *encoding* of the problem domain. For instance, genetic algorithms are often applied to binary or integer vectors. This encoding is then translated into the native format of the solution to be evaluated by the fitness function. Particular translations have been defined for specific tasks. For example, *grammatical evolution* [151, 127] uses the binary or integer vectors to instantiate programs from a grammar: each bit/integer is used to select rules to unfold, or terminal values to print, from a given grammar. *Cartesian Genetic Programming* [112, 110] transforms vectors of integers into directed graphs representing program functions and their inputs and outputs.

With respect to evolutionary algorithms, the search-amenable encoding is referred to as the *genotype*, and the *phenotype* is the translated form used for fitness evaluation [147]. An *individual* is an instance of the genotype and is translated into its phenotypic form using a *genotype-to-phenotype mapping*. Constituents of an individual (e.g. bits) are called *genes* and the value assigned to a gene is known as an *allele*. A *representation* consists of the genotype, the genotype-to-phenotype mapping, and the genetic operators (e.g. mutation and crossover) that are applied to the genotype [147]. Figure 2.6 illustrates the terminology of representations that we will be using throughout this thesis.



**Figure 2.6:** An abstract illustration of the terminology of representations and evolutionary algorithms.

Other techniques do not use an encoding and the search operators act directly upon the phenotype. *Genetic Programming* [134], for instance, uses tree structures to evolve programs. The mutation and crossover operators are defined with respect to the tree structures, e.g. for crossover, entire branches of the program are swapped.

Rothlauf [147] is the seminal work on representations for evolutionary algorithms. The choice of representation for a problem can have significant impact on the efficiency of the search

algorithm [147]. Designing a representation badly can result in making simple problems more difficult to solve, and designing a representation well, can make a challenging problem more manageable [147]. We describe two important properties of representations that can affect the efficacy of metaheuristic search algorithms: *locality* and *redundancy*.

**Locality** The term locality refers to the effects that small changes to a genotype have on the phenotype. Rothlauf uses the term *high-locality* to mean that neighbouring genotypes correspond to neighbouring phenotypes, and *low-locality* relates to the opposite – where neighbouring genotypes do not correspond to neighbouring phenotypes [147]. Galván-López et al. [52] prefer the terms *locality* and *non-locality*; a representation is said to ‘have locality’ when the phenotypic divergence is low, and a representation ‘has non-locality’ when the divergence is high. In this thesis we will use Galván-López et al.’s terminology. Locality is particularly important for evolutionary algorithms that make use of a mutation operator for continual improvement, as the search algorithm can be guided smoothly towards the solution through small genotypic mutations. Non-locality can result in effectively randomising the search algorithm, making it challenging to discover the solution.

Rothlauf [147] defines the locality  $d_m$  of a representation as:

$$d_m = \sum_{d_{x,y}^g = d_{min}^g} |d_{x,y}^p - d_{min}^p|$$

where:

$d_{x,y}^g$ : the distance between genotypes  $x^g$  and  $y^g$

$d_{x,y}^p$ : the distance between phenotypes  $x^p$  and  $y^p$

$d_{min}^p$ : the minimum possible distance between genotypes

$d_{min}^g$ : the minimum possible distance between phenotypes.

$d_m$  is calculated by summing the resulting phenotypic distance of all neighbours ( $d_{min}^g$ ) of every genotype. Incidentally, any genotypic neighbours that result in a phenotypic distance of zero – i.e. the same phenotype – are punished with the minimal phenotypic distance. When  $d_m = 0$ , a representation is said to have perfect locality.



Galván-López et al. observe that a locality score by itself is meaningless [52]. The score only gains meaning when it is compared against other representations of the same problem. Furthermore, the score is closely related to the size of the search space [161] – the more neighbours each genotype has, the higher the score is likely to be.

Figure 2.7 illustrates two representations, one with locality and one without. The grid represents the genotypic search space, organised by genotypic distance. Each adjacent cell has the minimum genotypic distance. The shading of the cell represents the

phenotypic distance to the adjacent cells: the greater the change in intensity, the greater the phenotypic distance.

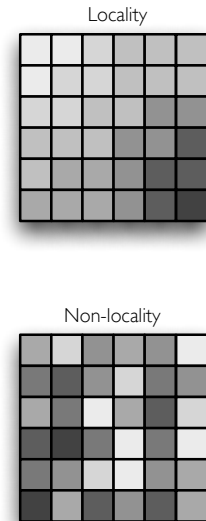
The common belief, supported by empirical studies [150, 149], is that locality is necessary for efficient evolutionary search. However, more recently, Seaton et al. [161] have demonstrated that, for certain classes of problems, non-locality can be beneficial, positing that one reason for this might be that high locality might restrict the diversity of the search landscape. Rothlauf also demonstrates that the difficulty of searching over *deceptive* fitness landscapes – where the fitness increases moving away from the global optimum – can be reduced with non-locality (as the search becomes more random) [147].

**Redundancy** Rothlauf defines redundant representations as encodings that require a larger number of genes than is necessary to encode the phenotype, and therefore result in a larger genotypic space than phenotypic space [147]. Redundancy can increase the connectivity of fitness landscapes, as *neutral* mutations to the genotype – those that have no effect on the resulting phenotype – enabling the search to explore new areas of the landscape [147, 190]. The opinion on whether neutrality is good or bad, however, is divided [147].

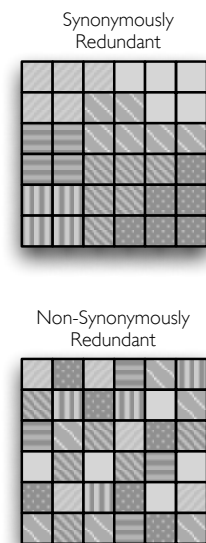
Rothlauf distinguishes between two types of redundancy: *synonymous* and *non-synonymous* redundancy [147]. A representation is synonymously redundant if the individuals that map to the same phenotype are similar to one another, and a representation is non-synonymously redundant if they are distinct. We illustrate these in figure 2.8: the grid represents the genotypic space, each square is an individual and its shading illustrates its phenotypic form. Note the similarity between the illustrations of synonymous redundancy (figure 2.8) and locality (figure 2.7). Synonymous redundancy does not guarantee locality – phenotypes may be grouped at the genotype level, but adjacent phenotypes may be dissimilar. E.g. the phenotypes  and , although genotypically adjacent, may be distinct. Non-synonymously redundant representations can reduce the difficulty of a problem as fit solutions may be distributed about the landscape. However, applying adaptation operators can result in completely distinct phenotypes, therefore resulting in random search [147].

### 2.2.3 Reformulating Software Engineering Problems

In the previous section, we have introduced metaheuristic search-based optimisation algorithms, and touched upon the ways to encode problems for search by defining genetic representations and described some properties of problem representations that affect the performance of metaheuristic search. In order to reformulate a software engineering problem as a search problem, one needs to define two artefacts [65, 28]: a suitable representation



**Figure 2.7:** An illustration of locality and non-locality. Each adjacent cell has the minimum genotypic distance. The colour intensity of each cell illustrates the phenotypic distance to its genotypic neighbours.



**Figure 2.8:** An illustration of the two types of redundancy. Each square is a genotype and its shading represents its phenotype.

of the problem and a fitness function to evaluate candidate solutions. Clark et al. [28] specify four characteristics of software engineering problems that make them suitable for being addressed using SBSE techniques:

1. The problem has a large solution space.
2. There are no known solutions that are efficient to compute.
3. It is reasonably easy to determine the quality of a candidate solution.
4. Generating candidate solutions is cheap.

We discussed representations in the previous section – the definition of both a genotype-to-phenotype mapping, and a set of genetic operators are required. Harman and Clark [64] propose that for many software engineering problems there exist metrics that can be used as the fitness function. In the general case, however, defining a fitness function is challenging and it is important to define it well [148]. A well-defined fitness function is able to distinguish between subtly different solutions in such a way that it substantially increases the chance of guiding the search algorithm through the search space to some optimal solution. In addition to techniques that use formally defined functions, there exist *interactive* evolutionary algorithms which utilise the user’s expertise to assign fitness to candidate solution [167, 166].

## 2.3 Combining MDE and SBSE

The observation that premises SBSE is inherent in MDE. Developing an optimal model of a domain can be extremely difficult, however it can be quite straightforward to determine whether a model accurately captures the domain. It is unrealistic to think that search could solve all problems associated with modelling, however the solutions to many of the challenges and tasks faced in MDE take the form of models or model management operations, indicating that utilising SBSE methods could prove fruitful. In particular, Clark et al’s criteria for the successful application of SBSE is largely met by MDE: solution spaces are large, solutions are difficult to come by, and it is possible to determine the quality of a candidate solution. Furthermore, it seems reasonable to believe that we can cheaply generate candidate solutions: there has been much work on evolving programs, which can be applied to discovering MMOs, and there is existing work that optimises class diagrams (e.g. [135]), so lessons can be learnt there.

In this section we present a detailed survey of existing work that has addressed the idea of combining MDE and SBSE. We focus in particular, on the ways in which MDE problems have

been encoded for use with SBSE techniques. We divide the literature into three categories and address each in turn. Section 2.3.1 deals with *discovering or optimising models*. Section 2.3.2 presents literature on *discovering or optimising model management operations*. Section 2.3.3 outlines work that highlights *research opportunities* for the two fields. Finally, in section 2.3.4, we list the case studies that have been used in the literature. For a new field to progress, it needs common examples for practitioners to compare and contrast their work, and to provide a common lexicon with which to discuss the field.

### 2.3.1 *Discovering or Optimising Models*

Models are the core development artefact in MDE. They are the design, the implementation, the testing, and the documentation of a system. Different kinds of models need to meet different standards. For example, implementation-related models should be optimal with respect to the implementation platform, documentation models need to be legible and promote understanding, whilst models used for validating model transformations need to satisfy testing-related criteria (e.g. test coverage). SBSE has had great success in optimising programs [67], increasing program comprehension [63], and in generating test data for programs [106, 107]. Furthermore, Rähä presents a detailed survey of the literature of search-based software design [135], dividing the field into four sub-fields: architecture design, clustering (high-level re-design), refactoring (low-level re-design), and software quality (analysis). Much of the literature surveyed by Rähä [135] focuses on utilising object-oriented metrics to optimise existing designs. For instance, O’Keeffe and Ó Cinnéidie [124] automatically restructure class diagrams using simulated annealing. They analyse candidate refactorings using a weighted sum of object-oriented metrics. Similarly, Harman and Tratt [69] automatically extract UML-like models of large Java programs and attempt to discover a sequence of refactorings that decreases coupling. As weighted fitness functions, such as that used in [124], can affect the quality of a fitness function, Harman and Tratt’s approach produces a set of Pareto optimal refactorings that shows the trade offs between different metrics, allowing the user to decide on the most appropriate refactoring.

Much of the search-based software design literature surveyed by Rähä in [135] focuses on class diagrams as opposed to domain-specific models, and use object-oriented metrics to guide the meta-heuristic. These metrics are applicable to metamodels, however they may not be applicable to all domain-specific models. For instance, in our zoo example in figure 2.3, metrics related to coupling and cohesion are irrelevant. Moreover, the discovered/optimal software designs from the literature are rarely used in a MDE context. In this section we describe the literature that fo-

cuses specifically on discovering or optimising models for MDE. Papers that use search to discover models are listed in table A.1 in Appendix A.1. These approaches are divided into two kinds: discovering models that satisfy some properties (e.g. for testing a model transformation), or optimising existing models. We focus here on discovering or optimising models in an MDE context.

**Discovering Models** In 2004, Fleurey et al. [46] proposed, but didn't implement, an evolutionary-inspired approach to discover and optimise the set of models used to test a model transformation. They utilised a *bacteriological* algorithm [9] – an adaptation of a genetic algorithm specifically designed for mutation testing – to mutate, rank, and select models from an existing test set based on the model's coverage of metamodel constructs. They define *metamodel partitions* – equivalence classes for primitive-typed properties in the metamodel. Partitions are defined using *model fragments* – textual specifications of the possible values in the partition for a particular feature. The proposed fitness function evaluates a model based on its coverage of the partitions, and promotes coverage of previously uncovered partitions. They suggest a mutation operator which randomly selects an element in the model and replaces its value from a different partition.

More recently, Cadavid et al. [21] demonstrate a user-driven approach to validating the boundaries (constraints on relationships and multiplicities) of a metamodel to ensure that a metamodel is neither under-constrained (allows invalid models) or over-constrained (disallows valid models) with respect to the domain. Fleurey et al.'s model fragments are used to define the model space, and *Alloy* [78] is used to generate the models within this space. The fitness function encourages the test set of models to be dissimilar whilst covering the set of model fragments. Domain experts are then able to examine the set of models to detect any faults in the specification of the metamodel.

Rose and Poulding [145] present a search-based approach to discovering sets of test models that provide coverage of model transformations. They translate a metamodel into an *input profile*, a stochastic context-free grammar that, when sampled, emits models. They apply a hill climbing algorithm over the probabilities used by the input profile to discover a combination of probabilities that produces a test set that maximises coverage of the transformation under test. Neto et al. [117] also demonstrate how a stochastic model generator can be incorporated into a search-driven framework for testing model-based technologies.

Burton et al. [18, 129] combine MDE with SBSE to address the challenge of *acquisition* problems. In large organisations, it is seen as beneficial to manage acquisition-related projects, not in terms of specific *resources*, but in terms of the *capabilities* they wish to be exhibited by the resources procured. Typically, acquisition problems are extremely expensive and it may not be obvious as

to which set of resources would be the most appropriate to invest in. Furthermore, the desired capabilities may be competing, as different stakeholders may have different requirements. Burton et al. address this issue by defining domain-specific modelling languages for expressing desired system *capabilities* and for modelling the set of *components* that can (partially or fully) satisfy the capabilities. They then use a multi-objective optimisation algorithm to determine a set of solutions which can be examined by the user. A solution is a combination of components that satisfy the desired capabilities, each with an associated cost. No single solution is better than another in all objectives, and so it allows investors to understand the trade-offs between different solutions. They encode solutions using an intermediary modelling language that captures correspondences between components and capabilities. A model-to-model transformation translates these into solution models.

Goldsby and Cheng [59] use a *digital organism*-based evolutionary approach to discover behaviour models of autonomous systems. Building on top of the *Avida* platform – a search framework for performing computational evolutionary biology [123] – they evolve interacting state machines that capture the behaviour of a system. Users provide a class diagram of the state of the autonomous system and any pre-existing state machines for the classes, and their tool, *Avida-MDE*, produces a state machine for each class. Class operations define the transitions and compositions of properties are used for transition guards. Individuals in the search are encoded as instructions that *select* elements from a specified state machine, *construct* transitions in a state machine, and *replicate* the entire state machine. Replicated models are randomly mutated to introduce diversity into the population. The fitness function considers three qualities of a candidate solution. Firstly, the model is syntax checked before being transformed into *Promela* [74] for checking temporal properties using the *SPIN* model checker [74]. Secondly, solutions with fewer transitions and those that exhibit determinism (one outgoing transition per state) are rewarded. Most importantly, however, each candidate solution is compared against a set of user-specified scenarios. Solutions are rewarded based on the percentage of the scenario execution path that they satisfy. Any solutions that satisfy the scenarios (regardless of the other two aspects) are known as *compliant behavioural models*. The set of compliant behavioural models found at the end of the execution can then be examined by the developer to see how each fairs in different environmental conditions, or be used to produce the implementation code of the system. In [58] the authors automatically cluster the population into groups that address different environment conditions using utility functions defined by the user.

Cheng further progresses this work to address the uncertainty found in self-adaptive systems [23]. Ramirez et al. [136, 138, 105]

use a genetic algorithm to discover optimal target system models at runtime. A model of the current system and information from system monitors are used to guide the search, along with a set of domain-independent evaluation functions. The target system models are encoded as graphs of system components and their interconnections; mutation and crossover operators are defined over these graphs. Additionally, Ramirez et al. [137] use a genetic algorithm to evolve goal models with relaxed constraints in order to accommodate uncertainty. An executable specification of the runtime system, a KAOS [32] goal model, and a set of utility functions are input into the search algorithm which produces a set of goal models that still satisfy the functional requirements but have relaxed constraints.

**Optimising Existing Models** Since Rähä's survey [135], Simons et al. [167] have presented an interactive evolutionary algorithm to improve the coupling and cohesion of class diagrams. Bowman et al. [13] utilise class coupling and cohesion metrics to discover the optimal assignment of operations and attributes to classes in a domain model. They represent the problem using a linear genotype where each gene represents a different feature in the class diagram and each allele defines which class that feature should belong to. Using a multi-objective genetic algorithm, they output a set of optimal assignments for the developer to inspect. Ghannem et al. [54] use Genetic Programming to evolve rules that detect refactoring opportunities in models. They encode a set of "IF-THEN" rules that state that if a compound condition applies, then a particular design defect is evident in the model.

Etemaadi et al. [39] argue that architecture optimisation should be performed directly on the architecture model (i.e. without a genotype-phenotype mapping) and propose software architecture-specific search operators with which to perform optimisation.

Optimising product line architectures (PLA) has recently received some attention. Colanzi et al. [30] focus on defining a search-amenable representation for PLAs. They argue that the existing approaches that address evolving software architectures are not suitable for searching for optimal PLAs as they are unable to adequately represent variability and product features. A metamodel-based, search-amenable representation of PLAs is proposed, along with bespoke mutation and crossover operators. The fitness of candidate solutions would be calculated using PLA-related metrics. Karimpour et al. [141] use a multi-objective optimisation algorithm to determine how best to integrate new feature requests into an existing product line – balancing overall product value with product line integrity. The product line is represented as a binary vector where a 1 represents a product is enabled and a 0 means it is disabled. Currently this approach only addressed the addition of new features and not their deletion or structural changes to the product line tree. Sanchez et al.



[156] present an approach to search for optimal system configurations from feature models that can be executed both offline or online, meaning that it can be used in both at the design stage and at run time. They used a direct encoding (feature model), using mutation to enable or disable features, and evaluate candidate solutions using quality attributes. Sayyad et al. [158] encode feature models as binary strings, where each bit represents a feature and its value defined whether it is enabled or not. They illustrate the value of visualising the evolution of each fitness objective in a multi-objective optimisation algorithm, and measure the effect of mutation and crossover rates on a case study.

**Summary** In this section we have provided an overview of the literature that attempts to use SBSE techniques to discover models. Excluding product line architectures, there is little overlap between the problems being addressed by these techniques, making comparison between approaches difficult. Regarding the choice of representation, there was no stand out approach. Whereas the most common problem representation used in the software design literature, surveyed in [135], is an integer encoding, the research presented in this section favours problem-specific encodings. Four papers use an integer string encoding, four use a bespoke encoding, four use a direct encoding, one uses a graph representation, one uses a grammar, and two use external tools to generate the model space. Two of the three papers related to product line architectures use binary encoding, whereas the third opts for a direct encoding. Most commonly, evolutionary algorithms were used to tackle single-objective problems, and NSGA-II used for multi-objective problems, with other techniques including simulated annealing, genetic programming, and particle-swarm optimisation.

### 2.3.2 *Discovering or Optimising Model Management Operations*

Model management operations (MMO) are arguably the heart of MDE: the power gained from a model becomes manifest when it is utilised in some way, and this is commonly done using an MMO. Section 2.1.1 overviews a number of MMOs. SBSE techniques could prove a powerful tool in aiding in the development of MMOs. For example, one of the challenges found in MDE is that of *metamodel evolution* [71]. When a metamodel is updated to a new version (e.g. on the introduction of new requirements), existing models may no longer conform to the metamodel – these models need to be *migrated*. For example, we may make the Animal class in Figure 2.2 abstract, and define two concrete subclasses, Mammal and Reptile. The model in Figure 2.3 would no longer conform to the metamodel due to the fact that abstract classes cannot be instantiated. We migrate models using specialised model

transformations, called a *migration strategies*, but it may not always be obvious what the best migration strategy is. One solution would be to simply delete the *Animal* objects – this would result in a model that conforms to the new version of the metamodel, but the model has lost information. A better solution would be to convert the two *Animal* objects into instances of the *Mammal* class. There may exist numerous valid, semantic-preserving, migration strategies, some of which may be more efficient than others with respect to non-functional properties. Discovering optimal migration strategies, or model transformations in general, is a key area where SBSE could benefit MDE and one where the bulk of existing research has focused.

**Existing Approaches** Table A.2 in Appendix A.2 lists the papers in this area. The first known attempt at using SBSE techniques to discover model transformations was by Kessentini et al. [85] in 2008. Their approach uses examples of mappings between source and target models to guide a *particle swarm optimisation* (PSO) algorithm. The example source and target models, and the mappings between them, are translated into predicates. The genotype is an integer vector where each gene location represents a construct in the source model under test, and each allele references one of the example mapping blocks. That is, the allele selects a mapping block to use as guidance for how to transform the associated source model construct into a target model construct. The predicate representing the source construct is compared for similarity against the left hand side of the predicates in the associated mapping block. The fitness function rewards solutions with the highest similarity score. Similarity matching allows the discovery of mappings for models containing data-related differences (e.g. the value of a string). The result of applying the search algorithm to a source model is a set of predicates that define how to transform that model into its target. This approach relies heavily on the user-defined mapping blocks. The mapping blocks need to cover all constructs defined in the source metamodel, which may be impractical or infeasible. Furthermore, it is unclear why search is needed here. Mutating the value of a gene will select a different mapping block for the associated construct. However, there is no relationship between mapping blocks and so it is unclear how the search algorithm can be guided towards better solutions: it appears that the representation suffers from non-locality. Considering the likelihood that the number of mapping blocks would be fairly small, it may be more practical to exhaustively compare each source model construct predicate against each of the predicates in the example mapping blocks.

This work was extended in [86] to address the issue of requiring an oracle to validate model transformations. The aim here is to utilise the mappings between example source and target models as an oracle function that states whether the transformation is

correct. Instead of an integer vector-based genotype that is used to select predicates from the examples, as in [85], in this paper the genotype is composed of predicates directly, with crossover and mutation operations defined over these predicates. The fitness function is couched in terms of risk: a statement of the probability that the transformation contains errors, calculated by performing a similarity test with the example mapping corpus.

Faunes et al. [43, 44] further extend this work to overcome the issue of defining example transformation mappings. Example source and target models are still required, however it is not required that there are mappings defined between them. They utilise Genetic Programming to generate transformation rules expressed in *JESS* [72], a fact-based rule language. The transformation is discovered by attempting to find a set of rules that is able to transform all source models into their relevant target models. The fitness of a candidate solution (a set of *JESS* rules) is calculated by performing a 'quick and efficient' match of the desired target model and the actual target model produced by the candidate transformation rules. This matching algorithm sacrifices precision with efficiency [84] and performs string similarity matching on the predicate-based representation of the models.

The problem of merging models is addressed using search in [88]. In traditional software development, code is worked on in parallel and the changes made by different programmers may need merging and any conflicts that arise need resolving. This is equally true for modelling. Kessentini et al. [88] utilise existing tools to detect the change operations that have been applied in each model to be merged by performing comparisons between the original model and the changed model(s). Applying these operations sequentially to the original model results in a merged model. However, conflicts can arise if an operation invalidates the preconditions of a subsequent operation. The goal of [88] therefore, is to discover the optimal ordering of operation application that minimises conflicts, whilst maximising the total number of operations that are applied. They represent the problem as a fixed length integer vector where the length is defined by the number of calculated operations (excluding duplicates). The evaluation of the work is weak. The set of models that they attempt to merge are incremental releases of the same model, meaning that the number of conflicts would be relatively small. To address this they asked five graduate students to edit the models to introduce conflicts [88]. It is unclear whether the introduced conflicts would be representative of real situations of parallel merging. To reduce the computational cost of the fitness function, the candidate sequence of operations is not applied to the source model. Instead, each operation is compared pairwise against all subsequent operations to determine if the operations would cause a conflict. This approach, albeit efficient and has shown to achieve good results on a single case study, ignores the possibility of *sequences* of op-

erations introducing conflict. The paper does not mention the amount of time that each evaluation costs, however one might prefer to suffer the cost of performance to increase the quality of the solutions. Additionally, the fitness function aims to maximise the number of operations that are applied to the final model and does not consider the human in the loop. Providing guidance for the user would be a better approach as commonly decisions of this nature are made by a lead developer or domain expert. Even when a change is non-conflicting, it may not be appropriate for the merge to occur due to it breaking some domain constraints. An interactive search algorithm could be used to allow the user guide the search towards appropriate merges, or perhaps return a *set* of potential merge strategies (with diversity between solutions encouraged) and allow the user to select the most appropriate solution.

ben Fadhel et al. [10] enumerate an exhaustive list of possible model refactorings and use simulated annealing to detect changes between two versions of a model. The fitness is based on a similarity score between the original target model and the model that results from applying the refactorings to the original source model. As with [88], they use predicates to represent the models and the refactorings are represented as a vector of string parameters.

Mkaouer et al. [114] use a multi-objective algorithm for discovering transformation rules. The aim to maximise the correctness of the transformation, whilst minimising the complexity of the transformation rules being produced, and maximising the quality of the target model that results from applying the transformation. They encode the transformation as a sequence of binary trees, where each tree represents a single transformation rule. The binary tree representation used only allows for simple, declarative transformations to be produced.

**Summary** One potential criticism of the approaches presented in this section is that they use a non-standard representation of the models, metamodels, and transformation mappings. In [85, 86], models and the mappings between them are transformed into predicates, and in [43], the metamodels and models are defined as JESS fact templates and fact sets. It is unclear whether these encodings are able to represent all possible metamodels and models. To date these approaches have all focused on a standard example transformation (class diagrams to relational database schemas – see section 2.3.4). The metamodels used in this transformation may not use all possible metamodeling concepts, and so more examples are required to show the generality of using predicates or facts. Furthermore, given the abundance of model transformation languages (e.g. ETL [94], ATL [79], QVT [121], VIATRA [8]), it seems surprising that the authors did not attempt to discover rules from one or more of those languages.

### 2.3.3 Position Statements

In [181] we highlight two areas where SBSE can benefit MDE. These areas align with the previous two sections: discovering and validating models, and discovering and validating MMOs. Simons [166] argues that interactive approaches to modelling using SBSE will become increasingly important. Mkaouer et al. [114] argue that most modelling problems are multi-objective and that modelling programs can benefit from *preference-based* multi-objective search algorithms. In particular, they claim that model refactoring, model evolution, and model testing are three problems that would benefit from these algorithms. Neto et al. [117] define a two-dimensional view of evaluating model-based software using search-based model generation. They highlight the trade off between model realism and the type of search being employed: random search on toy problems is the cheapest, and utilising industrial models to guide online search-based model generators is the most expensive.

McKinley et al. [105] propose that evolutionary algorithms can be used to mitigate uncertainty in adaptive systems. In particular, SBSE can provide both run-time and development-time support for adaptive software. For instance, they propose that it can aid in the discovery of novel adaptation algorithms, discovering optimal reconfigurations, and finding adaptation paths [105]. The works by Ramirez et al. [136, 137, 23] discussed earlier go some way to validating these claims.

Kessentini et al. [84] discuss lessons learnt from their attempts at using SBSE techniques to address MDE problems. The common approach they took was to encode the MDE artefacts of interest into a representation that could be used with existing SBSE techniques, but found it challenging to select an appropriate search algorithm. They argue that the availability of well-understood case studies is of great importance. Furthermore, they found that it was often necessary to trade precision with efficiency when defining the fitness function(s) for a given problem. To address these issues, they propose a framework that can be used to encode MDE problems for search. They posit the use of an intermediate language – an *encoding metamodel* – that is designed for use with genetic algorithms. Users define a transformation between their problem metamodel and the encoding metamodel for use with the framework. Standard genetic operators are defined with the framework, but users are able to define their own. Fitness functions are defined either with respect to the encoding metamodel, or through the instantiation of a *fitness evaluator* which requires a user-defined transformation back to the problem domain. This approach requires a large amount of effort on behalf of the user. Furthermore, the encoding metamodel may not elegantly capture all aspects of the problem domain and may not provide a suitable search-amenable representation of all possible

domains. Rothlauf [147] argues that the choice of representation is vital and has the ability to increase or decrease the difficulty of a search problem. Users of the encoding metamodel may need to go through numerous iterations to discover the optimal transformation from their domain.

Burton and Poulding [19] suggest three examples of the synergies between MDE and SBSE. Firstly, they argue that domain-specific modelling languages (DSML) can be used to represent problems. A DSML can capture the problem concisely and be more expressive than standard encodings [19]. Additionally, MDE tools provide generation of useful artefacts such as editors/viewers for the solution space. Secondly, they posit that SBSE can be used to instantiate models for testing purposes. Thirdly, they propose that SBSE can be used to discover model transformations. Furthermore, they postulate that MDE can assist SBSE with the choice of representation for the problem and solution, whilst enabling both interaction and visualisation of solutions. Moreover, defining a representation at a higher level of abstraction would enable the same representation to be used for many problems in the same domain.

#### 2.3.4 Literature Case Studies

Kessentini et al. [84] assert that well-defined case studies are required so that different SBSE techniques can be compared on common problems, to guide users when determining how to best formulate their problem for search and offer insight into which SBSE technique to apply. Table 2.1 lists the case studies that have been used in the literature surveyed in this chapter. In this thesis, we do not aim to provide a set of benchmark problems. The case studies in table 2.1, however, could be adapted and formalised to provide the basis for such a set of case studies.

The most commonly occurring case study is the class to relational database transformation. This has been used for both in the discovery of models and the discovery of model transformations. This is also a common example found in the model transformation literature. The model transformation literature could provide a useful portfolio of case studies that could be adopted by the search-based model-driven community. Furthermore, online transformation repositories, such as the ATL transformation zoo<sup>11</sup> and the AtlanMod Zoo (<sup>12</sup>), could provide useful benchmarks related both to models and MMOs.

<sup>11</sup> <http://www.eclipse.org/atl/atlTransformations/>

<sup>12</sup> <http://www.emn.fr/z-info/atlanmod/index.php/Zoos>

## 2.4 Summary

This chapter has introduced Model-Driven Engineering (MDE), a start-of-the-art approach to engineering software, and Search-Based Software Engineering (SBSE), a state-of-the-art approach to

Paper	Kind of study	Case study
[69]	Model refactoring	Three large Java programs (JHotDraw, Maven, XOM)
[10]	Model refactoring	Graphical Modelling Framework (Graph, Gen, Map metamodels)
[54]	Model defect detection	Class diagrams extracted from two real world projects
[59]	Model discovery (autonomous systems)	Robot navigation system
[58]	Model discovery (autonomous systems)	Adaptive flood warning system
[138]	Model discovery (runtime architecture)	Remote data mirrors
[156]	Model discovery (product line)	Video surveillance processing software
[158]	Model discovery (product line)	e-Shop feature model
[141]	Model discovery (product line)	e-Shop feature model
[18]	Model discovery (acquisition problems)	Next release problem, stock control system
[20]	Model discovery (to test transformations)	Statecharts metamodel, feature diagrams metamodel
[145]	Model discovery (to test transformations)	Robot navigation system
[46]	Model discovery (to test transformations)	Class diagram to relational database schema
[85]	Model transformation rule discovery	Class diagram to relational database schema
[44]	Model transformation rule discovery	Class diagram to relational database schema; Basic sequence diagram to state diagram; Advanced sequence diagram to state diagram
[86]	Model transformation oracle discovery	Class diagram to relational database schema

**Table 2.1:** *Case studies used in the literature.*

optimising software engineering problems. Both MDE and SBSE advocate automation as an enabler of development, and are well suited to work together to further improve software development practices. We have presented the literature that has combined MDE and SBSE, and grouped them into two categories: the discovery and optimisation of models, and the discovery and optimisation of model management operations. Although many of the problems being addressed in the literature aim to discover models, there is no standard way of applying SBSE techniques to MDE problems and so there is much diversity in the representations being used for search. A standardised representation, that is amenable to a wide range of SBSE techniques would lower the entry point to using SBSE to tackle MDE problems, and enable an exploration of which SBSE techniques work well for different classes of MDE problems. In the remainder of this thesis, we develop such a representation, apply it to a set of MDE problems and evaluate its search-related properties.





# *II*

## *Making Models Searchable*



# *Proof of Concept Representation for Models*

# 3

THE ORIGINAL PROTOTYPE of a search-amenable representation of MDE models arose out of work undertaken as part of set of a weekly lab meetings in November 2010. A new cohort of PhD students had recently joined the Enterprise Systems (ES) research group – a group whose focus often relates in some way to MDE. In order to familiarise the new starters with the technologies and tools commonly used in the ES group, we set up a team-based modelling challenge. A further motivation behind this challenge was to produce something that could be demonstrated to prospective students at university open days. The group was split into two teams, each lead by an experienced MDE practitioner, and were given the goal of producing a domain-specific language (DSL) for describing the behaviour of characters in a fighting-style computer game. One constraint was that the language needed to be usable by people with no programming experience (i.e. students and parents on university open days). The game, *Super Awesome Fighter 4000*<sup>1</sup> (SAF), was built over the course of two evenings for this task. The two teams both produced similar looking metamodels for their DSLs, but selected different ways of integrating them with SAF. One team created a graphical modelling language and used a model-to-text transformation to generate executable Java code to be included in the game. The other team implemented a textual modelling language and some Java utility classes that parsed and executed the model without generating any code. Due to not requiring any code generation, the textual DSL was deemed a more elegant solution and crowned the winner.

Discussions with Simon Poulding (then a lecturer in the ES group) lead to the idea of attempting to automatically discover fighters with certain characteristics or of differing skill levels, and from this came the first iteration of a search-amenable representation for models. This representation focuses on encoding textual models, utilising the language’s grammar to produce models from a string of integers.

## Contents

3.1 Super Awesome Fighter . . .	42
3.2 SAF as a Search Problem . .	45
3.3 FDL Search: Grammatical Evolution . . . .	46
3.4 Analysis of GE-Based Representation . . . . .	55
3.5 Summary . . . . .	59

<sup>1</sup> Now hosted as a Google Code project at <http://code.google.com/p/super-awesome-fighter> and playable at <http://super-awesome-fighter.appspot.com>.

**Chapter Contributions** The contributions of this chapter are summarised below.

- A search-amenable representation for textual MDE models, and its implementation using *Grammatical Evolution* [151, 127].
- An empirical analysis of the locality of Grammatical Evolution with respect to the prototype representation.

**Chapter Structure** This chapter describes how we made it possible to search for fighters with desirable characteristics, and therefore how we devised the first generic search-amenable representation of MDE models. Section 3.1 presents SAF in more detail, focusing on the textual DSL used to describe fighters. Section 3.2 describes the search goals related to SAF and includes the definition of the fitness metric with which to evaluate candidate fighters. Section 3.3 presents our *Grammatical Evolution*-based solution for searching for fighters with desirable characteristics. Finally, section 3.4 provides a critique of the approach, highlighting flaws in both the representation and the search algorithm used.

**Note:** The work described in this chapter is based on joint work with Simon Poulding (then a lecturer in this department), Louis Rose (then a research associate in this department), and other members of the Enterprise Systems (ES) research group, and has been published in [186]. Rose and other members of the ES group were responsible for defining FDL using EMFText [70], and creating the supporting Java classes that integrate FDL with SAF. Poulding implemented the metaheuristic search framework used in the experimentation, and aided in configuring and running the experiments. Section 3.3.1 draws extensively from [186] and was largely written by Poulding; it is included for completeness. My contributions are: the design and implementation of SAF, the GE mapping, and the fitness functions used to evaluate candidate fighters.



## 3.1 Super Awesome Fighter

*Super Awesome Fighter 4000* (SAF) is a fighting game written in Java to demonstrate MDE concepts at university open days. SAF was specifically designed to develop a DSL that describes its characters' behaviours. The core game does not use any MDE techniques or components; in fact, very little care went into the engineering of the game. The development goal was to create an artefact that resembles a real legacy application that developers wish to extend using MDE techniques<sup>2</sup>.

A fight in SAF takes place between two opponents, in the style of classic fighting games such as *Street Fighter*<sup>3</sup> or *Mortal Kombat*<sup>4</sup>. Players can be defined using a textual DSL (created by Rose), the *Fighter Description Language* (FDL), or can be implemented in Java. SAF is a time-step based game; at each time step the game engine asks the two players for an action to perform based on the current state of the game. Players can perform two actions simultaneously: one movement action (e.g. run towards the opponent), and one fighting action (e.g. punch high). Some actions take longer

<sup>2</sup> The SAF Google Code project aims to redesign SAF to be fully model-driven, and to integrate metaheuristic search at its heart.

<sup>3</sup> [http://en.wikipedia.org/wiki/Street\\_fighter](http://en.wikipedia.org/wiki/Street_fighter)

<sup>4</sup> [http://en.wikipedia.org/wiki/Mortal\\_Kombat](http://en.wikipedia.org/wiki/Mortal_Kombat)

than one time step; if the player selects such an action, they are blocked from performing another action until the previous action has completed. Players also have a set of customisable characteristics, such as strength and reach. These customisable characteristics are used to derive other characteristics, such as weight and speed; a powerful, long reaching player will be heavy and therefore slow, where as a less powerful player will be lighter and therefore faster.

### 3.1.1 Fighter Description Language

The *Fighter Description Language* (FDL) is a DSL, implemented using EMFText<sup>5</sup> [70], for specifying fighters in SAF. EMFText is a language and tool for defining textual DSLs. The EMFText library parses a FDL character (represented as a string) into a model in memory which is then used by SAF. Figure 3.1 shows the metamodel for FDL. A fighter (Bot) has two features: a Personality and a Behaviour. A fighter's personality is defined by a set of Characteristics – defining the power and reach of the fighter (values range between 0 and 10). If one of the characteristics is not specified by the user, its value defaults to 5. The behaviour of a fighter is made up of a set of BehaviourRules. These rules specify how the fighter should behave in certain Conditions. A rule is composed of a Condition, a MoveAction and a FightAction. FDL offers the ability to specify a *choice* of move and fight actions using the keyword *choose*. For example, a rule can define that it wants to either block high or block low, and the game will pick one of these at random. FDL provides a special condition, *always*, which is executed if no other condition is applicable. Listing 3.1 shows an example fighter in FDL and illustrates the choice mechanism.

<sup>5</sup> [www.emftext.org](http://www.emftext.org)

```
1 JackieChan {
2   kickPower = 7
3   punchPower = 5
4   kickReach = 3
5   punchReach = 9
6   far [run_towards punch_high]
7   near [choose (stand crouch walk_towards) kick_high]
8   much_stronger [walk_towards punch_low]
9   weaker [run_away choose (block_high block_low)]
10  always [walk_towards block_high]
11 }
```

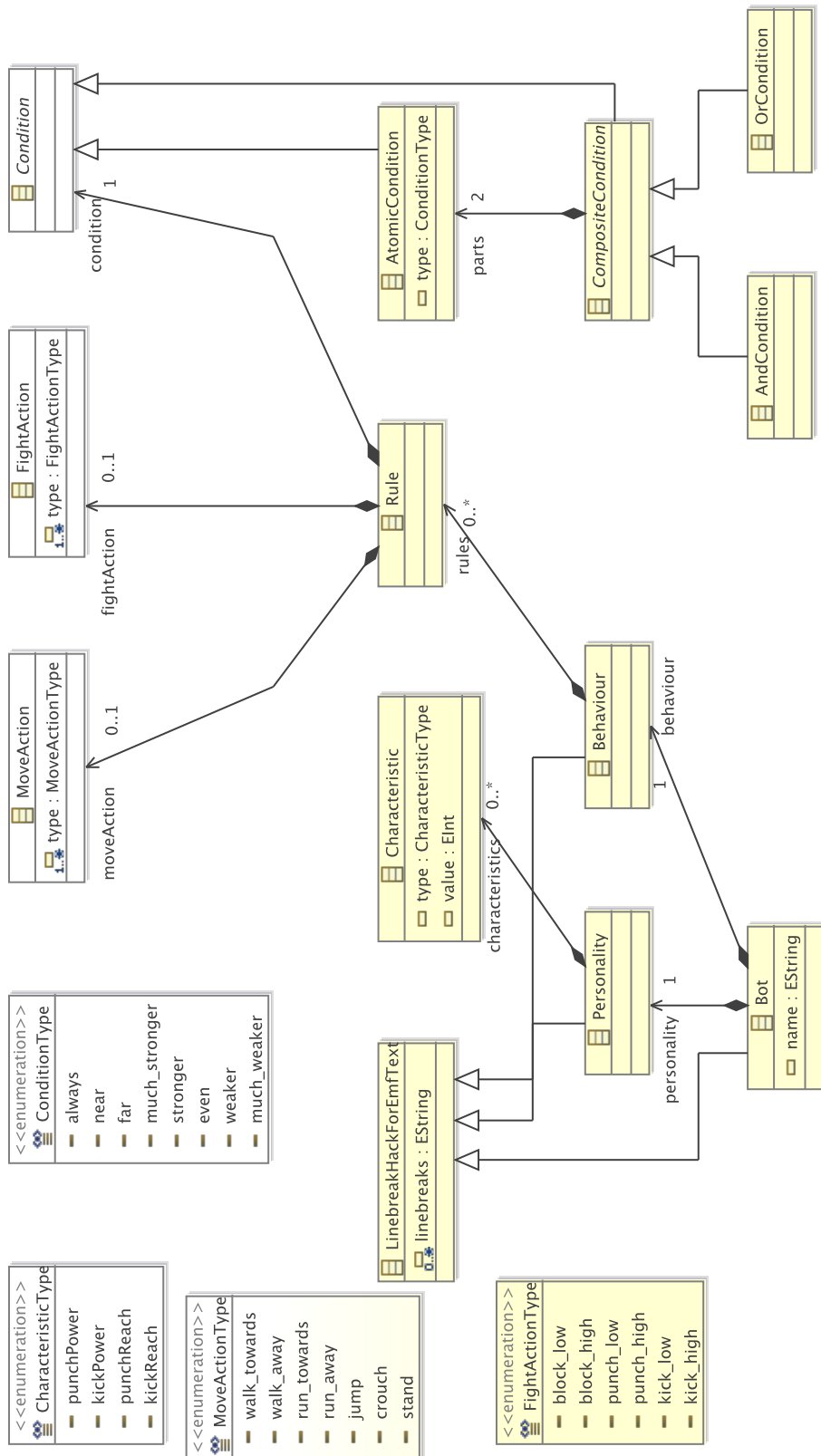
**Listing 3.1:** An example character defined using FDL.

EMFText was chosen for the implementation was due the fact that the ES team who developed FDL were familiar with it. The approach can be implemented for any equivalent grammar definition language that has a metamodel, such as Xtext<sup>6</sup> [37].

<sup>6</sup> [www.eclipse.org/Xtext](http://www.eclipse.org/Xtext)

## Summary

This section has presented an overview of the SAF game and described the DSL used to specify the rules that define the behaviour



**Figure 3.1:** The Ecore meta-model for the Fighter Description Language used in SAF.

of players in SAF. The next section sets out the search goals related to SAF.

## 3.2 *SAF as a Search Problem*

When defining a character in Rose’s FDL, it is not always apparent how good that character is until it has fought a number of matches. Games such as *Street Fighter* and *Mortal Kombat* pit the human player against fighters of increasing difficulty so that the challenge of the game increases the further a player progresses. Instead of manually iterating the definition of a fighter to develop a fighter with certain characteristics, we can use search. We set out to answer two experimental questions:

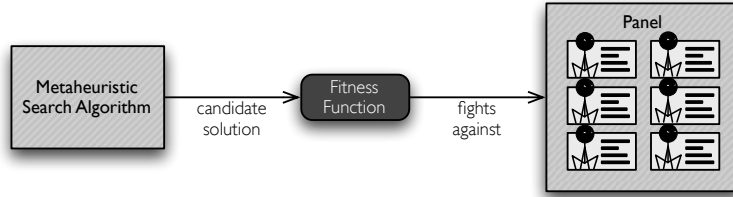
**EX1** Is it possible to specify unbeatable fighters? If a fighter can be unbeatable, it may be necessary to either amend the game play or restrict the Fighter Description Language to limit the possibility of a human player specifying such a fighter.

**EX2** Is it possible to derive a fighter that wins 80% of its fights against a range of other fighters? Such a fighter could be used as the pre-defined non-player opponent since it would provide an interesting, but not impossible, challenge for human players. The figure of 80% is an arbitrary choice that we believe represents a reasonably challenging opponent for human players. It acts as an illustration of discovering a fighter of a specified quality.

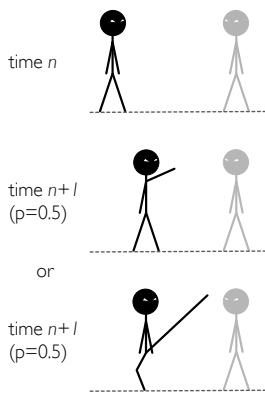
### 3.2.1 *Fitness Metric*

In order to assess the properties of “unbeatable” (EX1) and “wins 80% of its fights” (EX2), a set of opponents needs to be defined. It is not practical to test how each candidate fighter performs against *all* possible opponents, and so we created a ‘panel’ of representative opponents by asking members of the ES research group to specify what they believed would be winning fighters, illustrated in figure 3.2. (Note that our colleagues are acting simply as examples of typical human game players: they are *not* attempting to perform manually the equivalent of our proposed automated search-based approach in exploring the capabilities of the FDL.) The fitness of a candidate fighter is assessed by having it play the SAF game against each opponent in the panel. The game play is stochastic owing to the *choose* construct in the DSL (see figure 3.3), and so each candidate fighter fights each opponent a number of times so that a more accurate fitness can be estimated.

**Figure 3.2:** The experimental process to address the two SAF experimental goals.



```
near[
  walk_towards
  choose (kick_high punch_high)
]
```



**Figure 3.3:** Illustration of the choose operator.

The fitness of a candidate fighter ( $f$ ) is based on the difference between the number of fights won by the candidate fighter against the panel, and a target number of winning fights (e.g. 80%). It is calculated as:

$$f = \left| \rho n_{\text{opps}} n_{\text{fights}} - \sum_{o=1}^{n_{\text{opps}}} \sum_{i=1}^{n_{\text{fights}}} w_{o,i} \right| \quad (3.1)$$

where  $n_{\text{opps}}$  is the number of opponents in the panel;  $n_{\text{fights}}$  the number of fights with each opponent;  $\rho$  the proportion of fights that the fighter should win, and  $w_{o,i}$  an indicator variable set to 1 if the fighter wins the  $i^{\text{th}}$  fight against the  $o^{\text{th}}$  opponent, and 0 otherwise. The proportion of fights to win,  $\rho$ , is set to 1 for experiments on EX1, indicating an optimal fighter must win all fights against all opponents in the panel, and is set to 0.8 for experiments on EX2. Fighters with lower fitnesses are therefore better since they are closer to winning the desired proportion of fights.

### Summary

This section has laid out the search goals that we wish to investigate. The next section describes our first attempt at applying search to FDL models using *grammatical evolution* and answering the experimental questions.

## 3.3 FDL Search: Grammatical Evolution

FDL is a textual DSL. *Grammatical Evolution* [151, 127] is a metaheuristic technique designed for searching over grammars and is thus used to address our experimental goals. The genotype (strings of integers captured in a model) are mapped to the phenotype (FDL models) by a model-to-model transformation that takes into consideration the grammar of FDL. Section 3.3.1 gives an overview of grammatical evolution. Sections 3.3.2 and 3.3.3 describe the genotype and the process of mapping it to the phenotype. Section 3.3.4 then applies the approach to the experimental goals laid out in section 3.2.

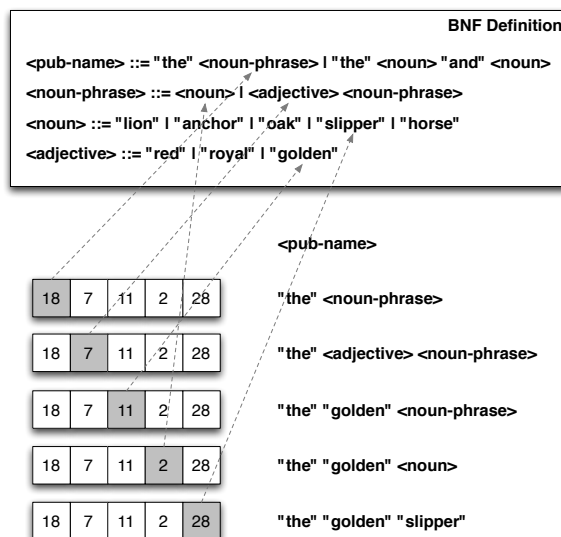


### 3.3.1 Background

*Note: This section is based closely on [186] and draws extensively on Poulding's contribution to the background for the paper. It is included for completeness.*

The technique of Grammatical Evolution (GE) was first described by Ryan and O'Neill [151, 127] as a mechanism for automatically deriving 'programs' in languages defined by a context-free grammar where the definition is expressed using Backus-Naur form (BNF). Applications of GE include symbolic regression [151], deriving rules for foreign exchange trading [15], and the interactive composition of music [165].

The central process in GE is the mapping from a linear genotype, such as a bit or integer string, to a phenotype that is an instance of a valid program in the language according to the BNF definition. Figure 3.4 illustrates the process using a simple grammar for naming pubs (bars).



**Figure 3.4:** An example of genotype-to-phenotype mapping in Grammatical Evolution. Simon Poulding is credited for this image and example.

The pub naming grammar is defined in BNF at the top of figure 3.4 and consists a series of production rules that specify how non-terminal symbols (the left-hand sides of the rule, such as <noun-phrase>) may be constructed from other non-terminals symbols and from terminal symbols (constant values that have no production rule, such as "red"). Vertical bars separate a series of choices as to how to construct the non-terminal symbols.

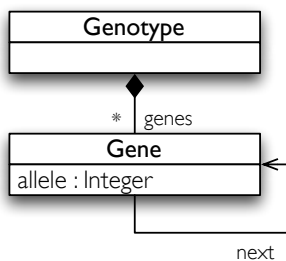
At the left of figure is the genotype that will be mapped to the phenotype, in this case, a pub name valid according to the naming grammar. The mapping process starts with the first production rule in the grammar, that for the non-terminal <pub-name>. There are three options as to how to produce this non-terminal, and the allele of the first gene determines which choice to use by taking the allele modulo the number of choices. In this case the allele is 18, there are 2 choices,  $18 \bmod 2 = 0$ , and so the first choice of the two, "the" <noun-phrase>, is used. The next construction

decision is for the non-terminal <noun-phrase> and it uses the value of the second gene. The gene has a value of 7, there are 2 choices,  $7 \bmod 2 = 1$ , and so the second choice is used. Production continues in this way until there are no more non-terminals to produce.

Should the mapping process require more genes than are present in the genotype, genes are re-used starting at the first gene. This process is known as *wrapping*. It is possible for the mapping process to enter an endless loop as a result of wrapping. Therefore, a sensible upper limit is placed on the number of wrappings that may occur, and any genotype which causes this limit to be reached is assigned the worst possible fitness.

Ryan and O’Neill’s original work on Grammatical Evolution used a specific genetic algorithm, with a variable length genotype and specialist genetic operators. More recent work makes a distinction between the genotype-to-phenotype mapping process, and the underlying search algorithm, using, for example, differential evolution [125] and particle swarm optimisation [126], in place of the genetic algorithm. We take a similar approach in the work described in this paper by designing a genotype-to-phenotype mapping process that is independent of the underlying search algorithm.

### 3.3.2 Defining the Genotype: the encoding of a fighter



**Figure 3.5:** *The genotype: a metamodel encoding individuals from the metaheuristic search framework.*

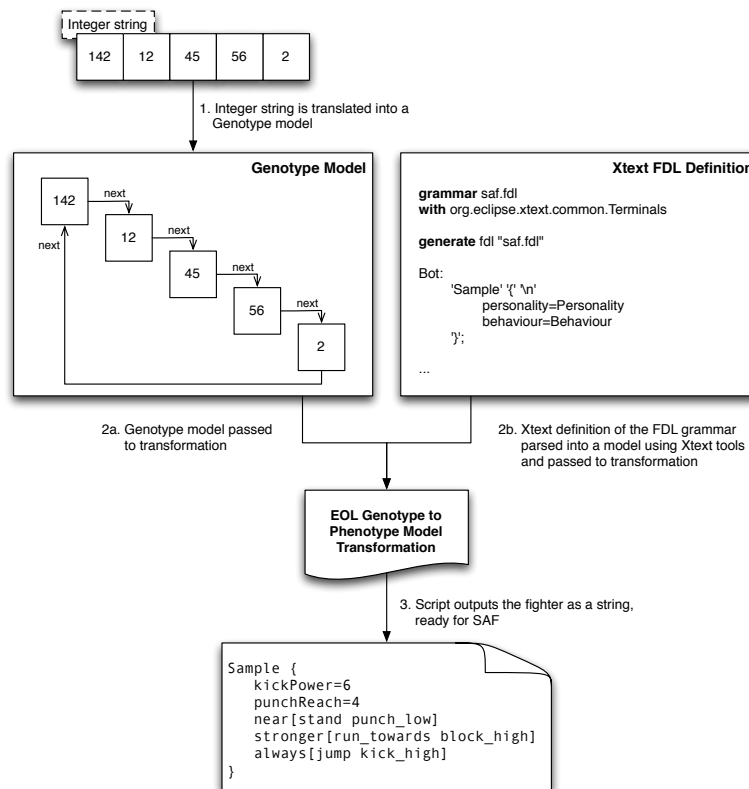
The first step in the process of applying Grammatical Evolution to SAF is to turn the genotype (a string of integers) into a model representation in order to perform the model transformation into the phenotype – an FDL fighter. Figure 3.5 illustrates the metamodel of our genotype. A Genotype is composed on a number of Genes. A Gene has one attribute: its allele; and one reference: a pointer to the next gene in the genotype.

The integer string produced by the metaheuristic search framework is used to create a model that conforms to this metamodel. A Gene object is created for each integer in the string, and its value attribute is set to the value of the integer. Each Gene’s next reference is set to the successive gene, with the final gene in the genotype pointing back to the first. This creates a cycle, meaning that the wrapping process that occurs in GE is handled automatically by traversing the references. An example model that conforms to this metamodel is shown as part of figure 3.6.

### 3.3.3 Transliterating the Phenotype: the FDL fighter

The next step is to transform this model of the genotype into a model of the phenotype (the Fighter Description Language). The transformation is written in the Epsilon Object Language (EOL) [92], a general purpose model management language that is part of the Epsilon model management platform [97]. Figure 3.6 is an

overview of this transformation process, a process which can be used with any metamodel defined by an *Xtext* grammar [37].



**Figure 3.6:** The process of transforming the genotype (an integer string used by the search framework) into the phenotype (a specific fighter description in FDL, for use in SAF).

Rose’s FDL was defined using EMFText. EMFText produces a metamodel for the DSL plus supporting tooling, such as a text-to-model parser and text editor, from the grammar definition. We could define our genotype-to-phenotype mapping using the metamodel produced by EMFText, and therefore tightly coupling the approach with FDL. However, the FDL grammar definition is itself a model: the EMFText grammar definition language has a metamodel and any grammars defined in the language are models that conform to this metamodel. In order to make our approach generic to any DSL defined by EMFText, we can write our genotype to phenotype model transformation with respect the EMFText metamodel rather than specific languages.

Unfortunately, the EMFText metamodel is verbose and writing the transformation became non-trivial. *Xtext*<sup>7</sup> [37] is another textual DSL generation framework built on top of EMF, that is similar to EMFText; languages defined using Xtext are models that conform to the Xtext metamodel. However, the Xtext metamodel is cleaner and more amenable to traversing with a model transformation. As such, we reimplemented the grammar of FDL in Xtext (see appendix B.1). Note however, we only did this in order to create an Xtext-conforming metamodel for FDL that enabled us to easily generate FDL characters; no code was generated from

<sup>7</sup> [www.eclipse.org/Xtext](http://www.eclipse.org/Xtext)

the Xtext definition and the existing EMFText-based code for FDL was untouched. The Xtext FDL model is used to produce a character as a string, which is then parsed into SAF using the EMFText tooling as normal. As was the idea with EMFText, our model transformation is applicable to any language defined in Xtext.

The Xtext metamodel contains metaclasses for all aspects of the grammar, including production rules, choices, and terminals. Each production rule in a grammar model is represented as an object conforming to a class in the Xtext metamodel and contains references to other objects in the model that represent its non-terminals and terminals. This representation facilitates the mapping process: where there is a choice in a production rule, genes from the genotype model are used to select which of the target objects to use and therefore which path to travel through the model. When the path reaches a terminal string, it is added to the output string.

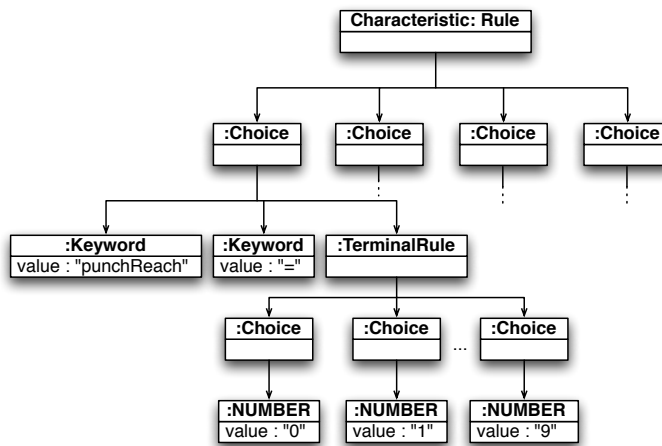
```

1 Characteristic :
2   'punchReach' '=' value=NUMBER '\n' | 'punchPower' '='
   value=NUMBER '\n' |
3   'kickReach' '=' value=NUMBER '\n' | 'kickPower' '=' value=
   NUMBER '\n' ;

```

**Listing 3.2:** The Xtext grammar rule defining characteristics of a fighter. For the full specification see Appendix B.1.

**Figure 3.7:** The Characteristic grammar rule shown in listing 3.2 from the FDL Xtext grammar, represented as an object diagram. The diagram visualises how the model of the grammar expresses rule choice. We expand the first choice of the grammar rule: to specify the punchReach characteristic. Leaves are terminals in the grammar.



To illustrate this approach, consider the rule in the fragment of the FDL grammar shown in listing 3.2. This rule defines the characteristics of a fighter, and contains four choices – each assigning a value to the different fighter characteristics (punchPower, punchReach, kickPower, kickReach). When the FDL language specification is parsed into a model conforming to the Xtext metamodel, the rule in listing 3.2 takes the shape shown in figure 3.7 (using object diagram syntax).

When this rule is reached during the transformation, the current gene’s allele identifies which alternative to execute by taking the allele modulo the number of choices. If the first alternative is chosen, the keywords (terminals) punchReach and = will be added

to the output string, and the next gene in the genotype model will select the NUMBER to assign to the selected characteristic. The execution can then traverse back up the reference chain (figure 3.7) and execute the next production rule in sequence, or terminate if no production rules in the grammar are left to expand. If the user-defined number of genotype wrappings is reached during the execution, the transformation aborts and the candidate is assigned the worst possible fitness. Otherwise, the transformation results in a string that conforms to the grammar of interest – in our case, a fighter in FDL.

### 3.3.4 Evaluation and Results

The previous section described a generic process for mapping an integer string genotype to a phenotype using EMF model transformation technologies. To illustrate and evaluate the approach, we apply it to SAF in order to address the experimental questions from section 3.2. The experiments use a genetic algorithm as the search algorithm, and in addition we perform secondary experiments using random search in order to assess whether the problem is sufficiently trivial that solutions can be found by random sampling of the search space. Both sets of experiments use our GE mapping to evaluate the fitness of candidate solutions.

The objective of this evaluation is to understand the feasibility of the proposed search-based approach in the context of the original motivating problem: finding interesting and challenging opponents in SAF. Furthermore, we wish to discover any issues with the approach that might affect its application to other domains, and understand where there are gaps that need addressing.

**Algorithm Settings and Implementation** The algorithm settings, including the parameters used in the genotype-phenotype mapping and in the fitness calculation, are listed in table 3.1. Since the efficiency of the algorithms is not being explicitly evaluated in this work, no substantial effort was made to tune the parameters to this particular problem, and the choice of some parameter settings (for example the use of integer mutation) was made with reference to existing GE studies, such as [75].

The genetic algorithm is a bespoke implementation in Java since this is the language used in the interface to the genotype-and-phenotype mapping. A bespoke implementation was written because it was initially thought to provide a more flexible basis for proposed future work on co-evolutionary strategies. However, other evolutionary computation libraries written in Java, such as *ECJ*<sup>8</sup>, could have been used in conjunction with our genotype-to-phenotype mapping process.

For random search, the bespoke genetic algorithm implementation was used but with the mutation probability set to 1.0. This has the effect of selecting a new random population at each gen-

<sup>8</sup> ECJ website: <http://cs.gmu.edu/~lijeclab/projects/ecj/>

Parameter	Setting
GENOTYPE-TO-PHENOTYPE MAPPING	
Number of genes	20
Gene value range	0–32767
Population size	20
Maximum number of generations	50
Initialisation method	Random gene values
Selection method (for reproduction)	Tournament, size 2
Reproduction method	Single point crossover
Mutation method	Integer mutation (random value)
Mutation probability (per gene)	0.1
Number of elite individuals	2
GRAMMATICAL EVOLUTION	
Maximum wrappings (during mapping)	10
FITNESS METRIC	
Number of opponents ( $n_{\text{opps}}$ )	7
Number of fights ( $n_{\text{fights}}$ )	5

**Table 3.1:** *Parameter settings for the genetic algorithm, genotype-to-phenotype mapping, and fitness metric.*

eration. The elite individuals are kept in the population to keep track of the best solutions found to date.

**Response** Four experiments were performed: one for each combination of question (EX<sub>1</sub> or EX<sub>2</sub>) and algorithm (genetic algorithm or random search). For each experiment, the algorithm was run 30 times, each run with a different seed to the pseudo-random number generator. Our chosen response metric is a measure of the effectiveness of the approach: the proportion of runs resulting in an ‘optimal’ (as defined by EX<sub>1</sub> or EX<sub>2</sub>) fighter. The fitness metric is noisy as a result of the stochastic choose construct in the FDL, so the condition for optimality is slightly relaxed to allow candidate fighters with a fitness of 1.0 or less. In other words, an optimal fighter may differ by at most one from the desired number of winning fights, rather than requiring an exact match. This condition also accommodates the situation where the choice of  $\rho$  causes the term  $\rho n_{\text{opps}} n_{\text{fights}}$  in the fitness function to be non-integer.

**Table 3.2:** *The proportion of successful runs (those that find an ‘optimal’ fighter) for the four experiments. The ranges in parentheses are the 95% confidence intervals. Values are rounded to 2 significant figures.*

Question	Search Algorithm	Proportion Successful
EX <sub>1</sub> ( $\rho = 1.0$ )	Genetic algorithm	0.67 (0.50 – 0.83)
EX <sub>1</sub> ( $\rho = 1.0$ )	Random search	0 (0 – 0.12)
EX <sub>2</sub> ( $\rho = 0.8$ )	Genetic algorithm	0.97 (0.88 – 1.0)
EX <sub>2</sub> ( $\rho = 0.8$ )	Random search	0.47 (0.31 – 0.66)

**Results and Analysis** Table 3.2 summarises the results of the four experiments. The ‘proportion successful’ column is the frac-

```

1 fighter {
2   punchReach=9
3   even[choose(crouch walk_towards) choose(block_high
4     punch_low)]
5   always[crouch block_low]
6 }

```

**Listing 3.3:** Example of an ‘unbeatable’ fighter description found by the genetic algorithm.

tion of algorithm runs in which an ‘optimal’ fighter was found. The 95% confidence intervals are shown in parentheses after the observed value, and are calculated using the Clopper-Pearson method [29] (chosen since it is typically a conservative estimate of the interval).

For EX1, the objective was to derive unbeatable fighters. The results show that unbeatable fighters can be derived: the genetic algorithm found such examples in approximately 67% of the algorithm runs. Listing 3.3 shows a particularly simple example of an optimal ‘unbeatable’ fighter derived during one algorithm run. The ease of derivation using a genetic algorithm is not necessarily an indication of the ease with which a human player may construct an unbeatable fighter. Nevertheless, it is plausible that a human player could derive unbeatable fighters with descriptions as simple as that in listing 3.3, and therefore the game play or the FDL may need to be re-engineered to avoid such fighters.

For EX2, the objective was to derive challenging fighters that won approximately 80% of the fights against the panel of opponents. The results show that it was easy for the genetic algorithm and possible, but not as easy, for random search to derive descriptions for such fighters, such as the example shown in listing 3.4. This allows us to conclude that our metaheuristic technique is effective, as random search is the lowest benchmark to compare metaheuristic techniques against [65].

```

1 fighter {
2   kickReach=9
3   stronger[choose(jump run_away) choose(kick_low block_low)]
4   far or much_weaker[choose(crouch run_towards) choose(
5     punch_low punch_high)]
6   always[crouch kick_low]
7 }

```

**Listing 3.4:** Example of an ‘challenging’ fighter description found by the genetic algorithm.

An unintended, but very useful, outcome of these experiments was that the search process exposed some shortcomings in the Fighter Description Language that were not discovered by human players. These shortcomings have exposed new requirements for FDL. One example was that the game engine requires that the fighter description specify a behaviour for every situation (whether weaker or stronger than the opponent fighter, or near or

far from it), but the language grammar does not enforce this requirement. If the fighter could not return a behaviour rule for a particular condition, the game would crash. This was resolved by ensuring that all descriptions contained an `always` clause.

```
1 fighter {  
2   punchPower=9  
3   punchPower=7  
4   punchPower=2  
5   kickPower=7  
6   punchPower=2  
7   kickPower=2  
8   near[crouch punch_low]  
9   stronger or far[choose(run_towards run_towards) kick_high]  
10  much_weaker and weaker[walk_away block_low]  
11  always[crouch kick_high]  
12 }
```

**Listing 3.5:** Example of an ‘unbeatable’ fighter description that illustrates language shortcomings.

Further examples of language shortcomings are illustrated in the description shown in listing 3.5: characteristics of `punchPower` and `kickPower` are specified multiple times (lines 2 to 7); the condition `much_weaker` and `weaker` can never be satisfied (line 10); and both choices in the `choose` clause are the same (line 9). Although none of these issues prevent game play – only one of the repeated characteristics is used; the condition is never considered; and the `choose` clause is equivalent to simply `run_towards` – they are not intended (and, moreover, unnecessarily increase the size of the search space). The language might be modified to avoid these shortcomings.

Finally, we compare the efficacy of the genetic algorithm and random search on the two experimental questions. The results for EX1 in table 3.2 suggest that random search cannot find an ‘unbeatable’ fighter (at least in the same upper limit on the number of fitness evaluations as the genetic algorithm), and that the problem is non-trivial. For EX2, random search does succeed in the easier problem of finding ‘challenging’ fighters, but with less consistency than the genetic algorithm. The non-overlapping confidence intervals indicate that the differences between random search and the genetic algorithm are statistically significant for both questions.

### **Summary**

This section has presented our first attempt at searching for models with desirable characteristics. We presented an approach that utilised a grammatical evolution-based model-to-model transformation for transforming a string of integers into a model and applied it to the Super Awesome Fighter case study. The results showed that we were able to discover fighters with certain characteristics and, by comparison with random search, demonstrated that metaheuristic search is possible over a space of models that



conform to a given metamodel.

The next section examines the drawbacks of this approach and further motivates the need for a generic representation of MDE models.

### 3.4 *Analysis of GE-Based Representation*

The previous section describes an approach that satisfied our requirements: we were able to search for models with desirable characteristics. The approach is generic to any textual DSL defined in any textual modelling language. Our implementation was based on the Xtext metamodel. However, the approach can be tailored to textual DSLs developed using other, similar, frameworks. No code needed to be generated from the Xtext grammar definition – our approach simply uses the grammar to produce a string which can then be parsed by existing DSL tooling (e.g. EMFText for FDL). This enables MDE practitioners to adopt the approach with minimal initial costs. Their existing Xtext metamodels can be used directly, only a fitness function needs to be defined.

There are, however, a number of other issues with the approach:

*Textual models only* The approach is heavily tied to textual modelling languages. It is unclear as to the proportion of textual modelling languages compared to graphical modelling languages, but it is clear that a generic approach to searching for models must support both textual and graphical models. One solution would be to utilise existing work that transforms graphical models into textual models and vice versa, e.g. HUTN [144], as has recently been done by Rose and Poulding [145].

*Cross-referencing not supported* The FDL language used for the case study is possibly too simple to draw conclusions that can be generalised to all textual DSLs. The language does not have any constructs that *cross-reference* another: if an FDL instance was viewed as an object diagram it would be shaped like a containment tree as opposed to a graph. Many languages provide support for cross-referencing objects defined elsewhere in the program. Xtext supports defining cross-references at the grammar level, by allowing names to be assigned to objects<sup>9</sup>. For our GE-based representation to be generic, support for cross-referencing is a necessity.

*Data not supported* FDL has minimal data support. It has only integer values representing personality characteristics, however,

<sup>9</sup> <http://www.eclipse.org/Xtext/documentation.html#DomainModelWalkThrough>, <http://blogs.itemis.de/stundzig/archives/773>

as there are so few, the valid integer values are encoded into the Xtext grammar. The original EMFText-based grammar for FDL allows any integer value to be assigned to a characteristic and the SAF game validates whether this value is acceptable. We simplified the grammar when converting it to Xtext to directly encode all possibilities as a set of terminals. This solution is obviously impractical in the general case, and so an improved representation would need to support primitive data types in a generic manner.

*GE ≈ random search* One criticism of GE is that the phenotypic value that each gene represents is too dependent on those genes that precede it. As each gene is used to select specific grammar rules, a change in the gene's value may change the selected grammar rule and each subsequent gene is used to select different grammar rules than previously. A small change in the genotype, therefore, can result in a large change in the phenotype, meaning that GE has low locality [150]. This means that neighbouring genotypes do not correspond to neighbouring phenotypes, and therefore applying metaheuristic search to GE-based representations can effectively result in random search.

To further understand how the locality of GE affects FDL, we performed some simple experiments which are presented in the next section. Locality is necessary for efficient evolutionary search. Therefore, by showing that GE suffers from non-locality, we can rule out using it as the basis for the generic representation. Solving the other issues mentioned above would be inconsequential if the representation suffers from non-locality. We address the issues of cross-referencing, data, and textual-only models in the next chapter where we present our generic, search-amenable representation of models.

### 3.4.1 FDL Locality

Locality has been shown to be important for efficient evolutionary search [150, 147]. In section 2.2.2 we defined locality in terms of the relationship between genotypic distances and associated phenotypic distances. A representation is said to have locality if a small change to the genotype also results in a small change to the phenotype. In order to investigate the locality of GE with respect to FDL, we have performed a set of experiments that apply small mutations to the genotype and analyse the effects this has on the phenotype.

The locality of a representation depends on the distance *metrics* defined on the genotypic and phenotypic spaces. For FDL, we have chosen to use an existing model comparison tool to calculate the phenotypic distance. We selected *EMF Compare*<sup>10</sup> for this due to it being the most robust EMF-based comparison frame-

<sup>10</sup> [www.eclipse.org/emf/compare/](http://www.eclipse.org/emf/compare/)

Parameter	Setting
Number of individuals analysed	100
Number of genes	20
Number of mutations per gene	100
Allele range	0-32767
Maximum wrappings (during mapping)	10
Number of repetitions	10

**Table 3.3:** *Parameter settings for locality analysis of the GE-based representation of FDL.*

work at the time of writing. EMF Compare works by first matching all objects in one model to the objects in the other. Matching is performed based on the object identifiers (either an XMI identifier, an attribute, or the fragment URI). Once the matches have been produced EMF determines the quantity and type (e.g. add, delete, move) of differences between the matched objects. We define the phenotypic distance between two models as the number of differences calculated by EMF Compare.

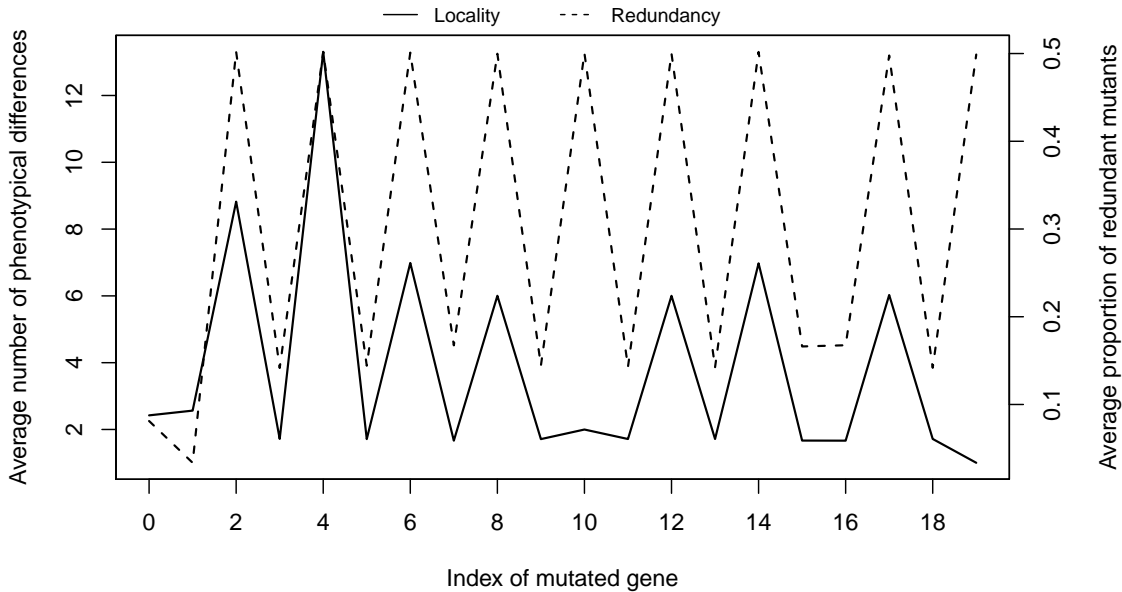
The distance metric defined on the genotypic space is that of the mutation operator used for evolutionary search. The mutation operator used in the previous experiments was random integer mutation, therefore if two genotypes differ by just one gene they have a genotypic distance measure of one – the minimum distance between genotypes.

**Algorithm Settings and Implementation** The parameters selected for the locality analysis are shown in table 3.3. We generated 100 different individuals with a maximum length of 20 genes (as in the earlier SAF experiments). For each individual, each gene is then randomly mutated 100 times independently of the others. This creates a sample set of neighbours whose genotypic distance is one. Each neighbour is then transformed to its phenotype and compared with the original, unmutated phenotype using EMF Compare to calculate the phenotypic distance. We averaged the number of differences between the original and the mutant for each gene. This whole process was repeated 10 times, each with a different random seed, meaning that 2,000,000 models were evaluated.

**Results and Analysis** Figure 3.8 shows the results found in this experiment. As expected, mutations to genes at the start of the genotype resulted in a larger number of differences in the mutant phenotype than those appearing later in the genotype.

Some gene mutations do not cause a change in the phenotype. As mentioned in section 2.2 this is called genotypic redundancy. In the case of locality, we are only interested in cases where the number of differences is greater than zero. For completeness, figure 3.8 also shows the average redundancy per gene (based on the proportion of mutations that resulted in zero phenotypical changes). Interestingly, you can see that there is a relationship

### Locality and Redundancy Analysis of the GE Representation for SAF Models



**Figure 3.8:** The results of analysing the locality and redundancy of GE with respect to the FDL.

between high redundancy and low locality. These spikes likely appear at the Condition and Action grammar rules (from appendix B.1). Both of these have binary choices – hence the 50% redundancy – and selecting the alternative will result in many phenotypical differences. The subsequent genes have lower locality as they are used to select terminal values, e.g. the condition types, and therefore result in fewer phenotypical changes.

On average, nearly 30% of genotypical neighbours are phenotypically identical. Furthermore, the average number of phenotypical differences that a random genotypic mutation will induce is 3.8. This score is only meaningful in context. An average of 3.8 model differences per mutation may be too much to provide efficient evolutionary search. If this technique was used with more complex grammars, it is likely to be even higher – the number of phenotypic differences increase where there is a choice in a grammar rule. More choices and more rules will result in more phenotypic differences. A further investigation of GE being applied to other languages is required to draw strong conclusions here.

One problem with this experiment is that we only considered structural differences in the analysis. For languages like FDL, two syntactically distinct models may be *semantically* equivalent. Furthermore, in the case of FDL (and possibly other languages) there are two ways in which can define the semantics of an FDL model. Firstly, we could define the semantics based on the results of fights. If two syntactically distinct fighters win the same proportion of fights against a common set of opponents, they might

P1	<pre>fighter{   even[walk_towards block_high]   always[walk_towards block_high] }</pre>
P2	<pre>fighter{   even[walk_towards block_high]   weaker[walk_towards block_high]   always[walk_towards block_high] }</pre>
P3	<pre>fighter{   even or weaker[walk_towards block_high]   always[walk_towards block_high] }</pre>
P4	<pre>fighter{   weaker[walk_towards block_high]   stronger[walk_towards block_high]   near[walk_towards block_high]   far[walk_towards block_high]   much_stronger[walk_towards block_high]   much_weaker[walk_towards block_high]   even[walk_towards block_high]   always[walk_towards block_high] }</pre>

**Figure 3.9:** Four syntactically different, but semantically equivalent, FDL fighters.

be seen as semantically equivalent (consider the case where we are searching for suitable opponents for a player). Secondly, the same behaviour can be expressed in FDL in different ways. For instance, the fighters in figure 3.9 are all semantically equivalent, but syntactically quite different – as shown in table 3.4. This may be a side effect of the domain, but it is something we need to consider when evaluating candidate solutions in any domain: structural comparison may not always be adequate.

### Summary

This section has enumerated a number of flaws with the current GE-based representation of (textual) models, and focused on the issue of locality. The representation’s flaws mean that it is not currently possible to represent all possible models (as only a subset of textual metamodels are supported) and evolutionary search is inefficient. It is true that the locality- and redundancy-related results presented in this chapter cannot be generalised to all Xtext-defined languages, but they do illustrate the issue with using a GE-based mapping for representing models.

## 3.5 Summary

In this chapter we have presented the original case study that lead to the development of a prototype search-amenable representation for MDE models. We have presented a video game,

	p1	p2	p3	p4
p1	0	5	3	26
p2	5	0	8	23
p3	3	8	0	27
p4	26	23	27	0

**Table 3.4:** Illustration of how semantically equivalent models may be syntactically distinct.

Super Awesome Fighter, which includes a textual DSL for specifying character behaviour. We demonstrated that it is possible to use evolutionary search to discover fighters with particular qualities, and provided a short analysis of the grammatical evolution-based representation with which we used to achieve this. This work has provided a useful platform of knowledge on which to build a generic, search-amenable representation of MDE models. The next chapter presents our solution.

### **3.5.1 *Future Work for SAF***

Beyond the realm of searchable representations, the SAF work could be extended in a number of directions. Firstly, the opponents against which the fighter's fitness metric is assessed could be derived using co-evolutionary methods rather than a human-derived panel. We speculate that currently the fighter properties of 'unbeatable' and 'challenging' may not be applicable beyond the panel of human-derived opponents, and that by co-evolving a larger, diverse panel of opponents, fighters with more robust properties may be derived. Secondly, non-player fighters could be dynamically evolved during the game play: each time a human player finds a winning fighter, a more challenging non-player opponent could be evolved, thus ensuring the human player's continued interest. We address the latter point in terms of self-adaptive systems in section 5.2.

# A Generic Representation for Models

# 4

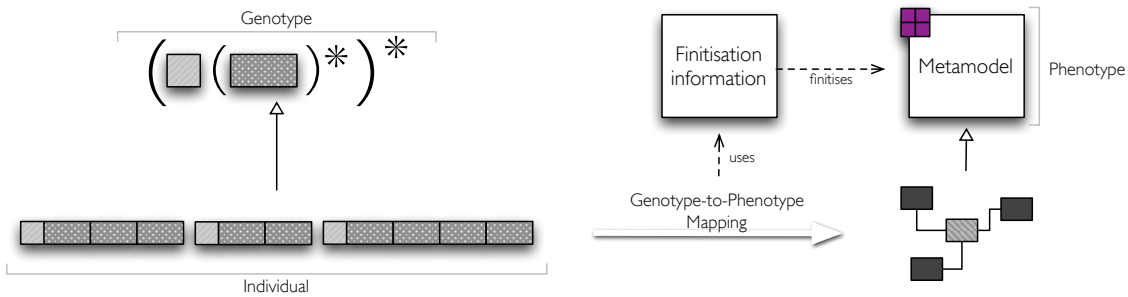
IN CHAPTER 2 WE DESCRIBED previous research that combined MDE and SBSE to address MDE problems, and argued the need for a simple way to apply SBSE techniques to common MDE problems. The previous chapter took the first step to producing such an approach: an integer-based representation using Grammatical Evolution to discover optimal textual models. This prototype demonstrated the feasibility of defining a general-purpose search-amenable encoding of models that, but suffered from a number of issues. Firstly, the approach was only applicable for textual models. Recent work by Rose and Poulding [145] propose generating *HUTN* [144] – a human-readable textual notation for models – which, although textual, can represent any kind of model. Secondly, the prototype representation was incomplete as it was unable to produce non-containment references between two model elements. Finally, as demonstrated in [150] and emphasized in section 3.4, Grammatical Evolution suffers from non-locality which can result in inefficient evolutionary search.

This chapter presents a generic representation of models that conform to MOF [120] metamodels, which is amenable to many well-studied metaheuristic optimisation techniques. Introduced in section 2.1.1, MOF is a standardised language for defining metamodels. Our representation addresses the issues related to the prototype implementation: it is independent of the concrete syntax of the modelling language; it can encode all features of MOF models (e.g. containment references) and all possible models that can conform to a given metamodel; and it is applicable to many metaheuristic search-based optimisation algorithms. We present *Crepe*: an implementation of our representation for the metamodelling language of the Eclipse Modeling Framework (EMF), *Ecore* (see section 2.1.2). Furthermore, we introduce a model-based metaheuristic search framework, *MBMS*, built using state-of-the-art MDE technologies, that uses *Crepe* at its core. Figure 4.1 illustrates the core components of the representation. We define a *genotype*: a description of the structure of an individual; and a mapping from the genotype to the *phenotype*: a

## Contents

4.1 Metamodel Finitisation . . . . .	63
4.2 The Genotype . . . . .	67
4.3 Genotype to Phenotype Mapping . . . . .	70
4.4 Search Properties and Adaptation Operators . . . . .	74
4.5 Crepe and MBMS: The Realisation for Ecore . . . . .	77
4.6 Worked Example . . . . .	87
4.7 Discussion . . . . .	92

model conforming to a user-provided metamodel. In order to accomplish the mapping, the user provides some *finitisation information*: structural and data-related constraints that control the models that can be encoded by the representation. Not shown in the diagram are the genetic operators that are applied to individuals using search.



**Figure 4.1:** A conceptual overview of the components of our generic model representation.

**Chapter Contributions** The contributions of this chapter are summarised below.

- The definition of a novel generic representation of models that is amenable to a wide range of existing metaheuristic optimisation techniques.
- The implementation of the model representation using MDE technologies.
- The implementation of a metaheuristic search framework, built using state-of-the-art MDE technologies, where all aspects of the framework are expressed as models, and the core algorithms are defined as model management operations.
- An interactive, web-based visualisation of the search algorithm, made possible through the use of MDE technologies and by capturing the progress of the search algorithm in a model.

**Chapter Structure** Section 4.1 presents metamodel finitisation: the process of defining extra structural constraints and specifying acceptable data values, thereby reducing the size of the search space and enabling the representation to produce models of interest. Section 4.2 briefly discusses the genotypes used in other representations and presents the genotype that we have designed to represent models. Section 4.3 details the mapping from our genotype to a model conforming to a given metamodel. We present our implementation of the representation, Crepe, and a model-driven metaheuristic search framework, MBMS, in section 4.5. Section 4.6 illustrates the use of Crepe and MBMS on a simple example. Finally, we discuss the limitations of the representation in section 4.7.



## 4.1 Metamodel Finitisation

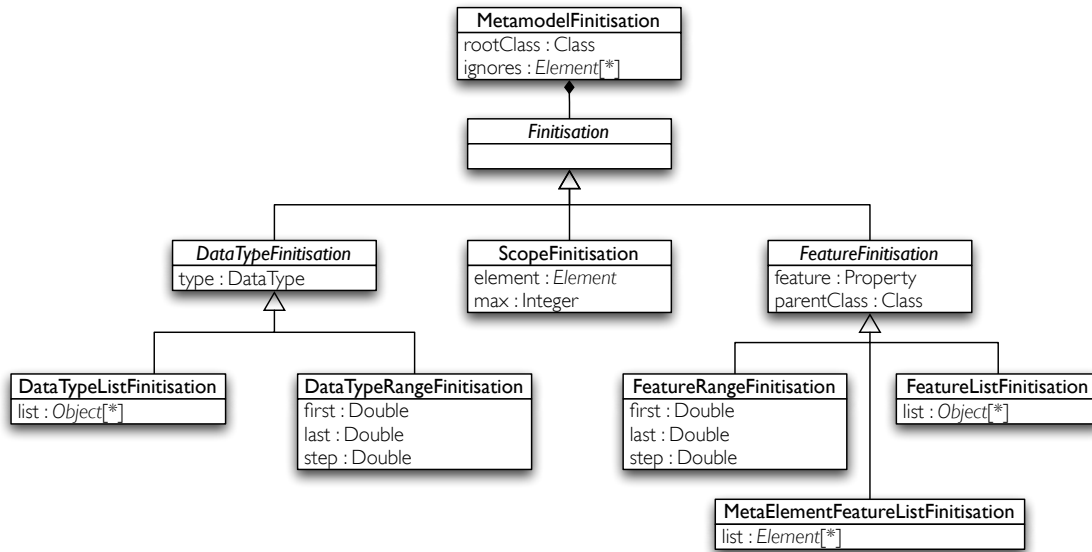
When designing metamodels, it is common to use unbounded data types for attributes (e.g. integers, strings, etc.), or to assign one-to-many multiplicities to associations instead of enforcing an upper bound (we provide supporting evidence for this claim in section 7.2 where we shows that nearly 50% of the references in a large corpus of metamodels do not have an explicit upper bound). This results in an infinitely large *model space* – i.e. there are an infinite number of models that can conform to the metamodel. In reality it is uncommon for the domain being modelled to require the infinite number of possibilities for every attribute. For instance, it is unlikely that an integer-typed attribute in the domain would really take any possible integer, and so can be bounded to reduce the size the model space<sup>1</sup>. Moreover, regarding strings, it is likely that the value of a string-typed attribute has little effect on the model's use. If the value of a string-typed attribute is important for an associated MMO, then the developers will have in mind a fixed set of values that the MMO is hard-coded to use. Alternatively, a string-typed attribute could be refactored into an enumeration class, or a new meta-class. Therefore, although the existence of a string-typed attribute will result in an infinite model space (ignoring physical limitations of hardware), it is likely that this is artificially large and the model space can be made finite by specifying a set of possible values.

In order to transform the genotype into the phenotype we need to finitely describe what we would like our models to look like. The information present in the metamodel that we want the models to conform to is not simply enough by itself to create realistic models. More information is needed to describe the data that can appear in instances of the metamodel. This is analogous to the *terminal set* used in Genetic Programming [134] which specifies the terminals (variables and constants) that can appear in the programs. Additionally, we may wish to enforce an upper bound on certain associations in order to limit the the search space to a computationally reasonable size.

Incidentally, it is not the case that a metamodel has just one finitisation. Different applications of the metamodel may require different finitisations. Furthermore, as already alluded to, the finitisation information provided can have a significant impact on the effectiveness of the search algorithm: the more information specified, the greater the size of the search space due to the combinatorial explosion of the possible value assignments.

We capture this extra domain information in a model called the *finitisation model*. The initial idea was to annotate the metamodel (a common practice for metamodels defined in Ecore [170]), however it is not guaranteed that users have access to the metamodel directly in order to add the annotations (e.g. if the metamodel is

<sup>1</sup> In fact, it is possibly more likely that the attribute has a probability distribution of values, but we leave this as future work.

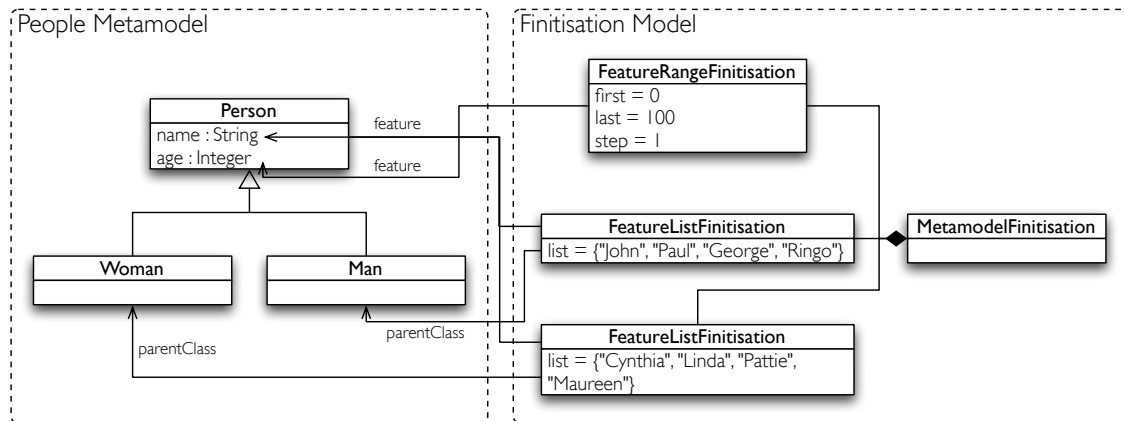


**Figure 4.2:** The meta-model for the finitisation model, written in MOF. The types *Object*, *Element*, *Class*, and *Property* refer to MOF classes, and represent references to the metamodel being finitised.

propriety), and as previously mentioned, the same metamodels can be finitised differently to address different problems. Creating a model to capture finitisation information is non-intrusive and is more powerful than annotations as it allows more complex finitisations to be defined. The metamodel for the finitisation model is shown in figure 4.2. We currently support three different forms of finitisation: feature restriction, data type restriction, and meta-element scoping. Additionally, we allow parts of the metamodel to be *ignored*, and require the user to specify a *root* object type. The metamodel is defined in a way that would allow for new forms of finitisations to be added as they are encountered through the application of many case studies. This section describes the finitisation metamodel and process of finitisation in detail.

#### 4.1.1 Data and Structural Finitisation

Both specific features and entire data types can be restricted to a list or range of values. Numerical ranges are specified as doubles, and can be cast to the appropriate numeric type based on the particular feature to which the value is being assigned. FeatureFinitisations specify the feature that is being finitised and also allows an optional *parentClass* to be specified. This reference is useful to manage different subclasses who inherited features from the same superclass. For instance, figure 4.3 shows a very simple metamodel for people. When creating the finitisation model for the people metamodel, the user may wish to define the age finitisation on the abstract *Person* class; this finitisation is inherited by the children of the person class. The user may, however, want to provide different finitisations for the name attribute inherited by



**Figure 4.3:** A simple meta-model illustrating the use of the optional `parentClass` field in the `FeatureFinitisation` class of our finitisation model.

Male and Female and use the `parentClass` feature to specify this.

*Data type restriction* makes the finitisation process more tractable; the alternative would be to define finitisations for every feature of every type. Data type finitisations do not override specific finitisations of features of the same data type: if a particular feature has a `FeatureRangeFinitisation` assigned to it and that feature’s data type also has been finitised, only the feature-specific values will be used to populate the models being produced. However, if a feature has multiple finitisations (e.g. both a range and a list), the entire set of values is considered as that feature’s finitisation.

In addition to finitising the values of features and data types, we also support *scoping* for both meta-classes and references. The finitisation metamodel allows developers to specify upper limits (a *scope*) on both the number of instances of a particular meta-class that can appear in a model, and on the number of objects that a particular reference can point to. This addresses the issue of unbounded references, and reduces the size of the model space. Whereas the finitisation of infinite data types is required, scoping is not. Scoping, however, reduces the size of the search space, making the problem more tractable<sup>2</sup>. Depending on the particular application of the search, specifying a small scope for many classes may be sufficient for a successful search. For instance, if the goal of the search is to find models that expose bugs in a model transformation, we may be able to make use of the *small-scope hypothesis* [78] which states that the counterexamples for most bugs will arise within a small scope.

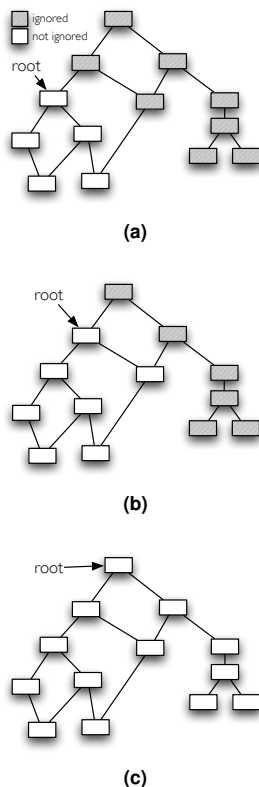
To support inter-model referencing, i.e. cases where an object in one model references an element in a different model, the `MetaElementFeatureListFinitisation` class allows users to specify possible external targets for that reference. This could be used to provide partial solutions to a problem. For example, consider the zoo metamodel in figure 2.2. Animals can be assigned to cages and can also eat other animals. A considerate zookeeper would not want to put animals in cages with their predators. Therefore, the zookeeper could search for the optimal assignment of ani-

<sup>2</sup> Care is needed to ensure you don’t make the solution model out of scope.

mals to cages by building a model of the animals and cages and specify them as `MetaElementFeatureListFinitisations` for a secondary metamodel that manages cage assignments.

### 4.1.2 Ignoring Elements

Some parts of a metamodel may not be relevant to all its instances. For example, when creating UML class diagrams, the modeller is not interested in parts of the UML metamodel that relate to sequence diagrams or activity diagrams. It may also be the case that certain features of a particular metaclass are not relevant to the search goal and would simply increase the size of the search space unnecessarily. The finitisation model, therefore, allows users to specify that certain meta-elements should be ignored during the mapping process (the `ignores` feature in the `MetamodelFinitisation` class in figure 4.2). If a meta-class is ignored, then all of its subclasses are also ignored. Furthermore, if a feature in a superclass is ignored, then that feature is ignored in all subclasses.



**Figure 4.4:** An illustration of specifying the root meta-class to focus the search space on a particular area of a metamodel.

### 4.1.3 The Need for Root

The only compulsory element in the finitisation model is the `rootClass` feature in the `MetamodelFinitisation` class (figure 4.2). This is a reference to a particular meta-class in the metamodel whose model space we plan to explore. For each individual analysed during the search, this root class is instantiated and acts as the container for the model objects creating during the mapping from genotype to phenotype. All models naturally have a container object. For instance, in UML a class diagram is contained by a `Package` object, and a `Package` object is contained by a `Model` object. It may not, however, be possible to infer this root class directly from the metamodel and so the user is required to explicitly specify a root.

We previously mentioned that metamodel elements can be explicitly ignored. Specifying a root class can have the same effect. As illustrated in figure 4.4, any meta-classes that are not directly or indirectly contained by the root class are implicitly ignored. For instance, to use our technique to search for a particular UML class diagram, it would be possible to set the UML `Package` meta-class as the root class, thus automatically ignoring all non-class diagram related meta-classes.

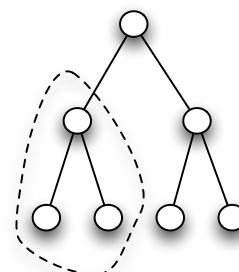
We now describe the structure of individuals in our representation, before explaining how an individual is transformed from its genotypical representation into a model conforming to the given metamodel.

## 4.2 The Genotype

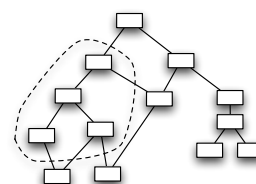
Concrete syntaxes are often defined to provide a more meaningful and user-friendly representation of the model, however the underlying elements of MOF-like models can be visualised as *object graphs*. A *direct* genotypic representation, where the search space is the solution space (or equivalently using GA terminology, where the genotype is also the phenotype), would ensure an *efficient* encoding: there would be no genotypic redundancy. However, significant work would be required to define the genetic operators that can be used to efficiently explore the solution space. We have chosen to utilise a linear, integer-based genotype, structured in a way that allows the complete expression of models. Before we introduce our genotype, we briefly describe other existing genotypes that were discarded or gave inspiration to our design.

### 4.2.1 Existing Representations

Inspiration could be taken from the *Genetic Programming* field [134], which does not distinguish between the genotype and phenotype and uses *trees* to represent programs. However the step up in complexity from trees to graphs is not trivial. GP's mutation and crossover operators are defined to act directly upon these trees. Altering these operators to work on object graphs is non-trivial due to the complexity of object graphs. Firstly, the crossover operator in GP is *subtree crossover*. A node is selected at random in both parents to be the crossover point. The subtree in the first parent is then replaced by the subtree in the second, and so care is needed to ensure that the crossover does not break any syntactic constraints. Defining a crossover operator for graphs is more challenging. It would not be possible to select a single crossover *point*. Instead, a crossover *graph* would need selecting (perhaps a heuristic could be used to select the subgraph based on elements attached to a randomly chosen element). Figure 4.5b illustrates this. Ensuring that the subgraphs are able to replace one another (without invalidating structural constraints) is costly; selecting subgraphs that are able to replace each other (analogous to strongly-typed GP) falls under the *subgraph-matching* problem which is known to be NP-complete [168]. Whereas the 'interface' to trees in GP is a single node, making crossover possible (figure 4.5a), the 'interface' of an object graph is potentially many nodes (figure 4.5b). Determining an appropriate solution to managing the assignment of these references after crossover would be non-trivial: many references would need to be destroyed or recreated randomly – as found by Burton et al. in their crossover operator for models [18]. As the goal of crossover is to exchange key genetic information ("building blocks") which is passed to



(a) Genetic programming: subtree crossover.



(b) Object graph representation: subgraph crossover

**Figure 4.5:** Selecting parts of the phenotype for crossover.

children, this may prove too destructive and therefore reduce the algorithm to random search.

GP uses two mutation operators – *subtree mutation* and *point mutation* [134]. Subtree mutation suffers the same issues as the crossover operator, as it works by replacing randomly selected subtrees with randomly generated subtrees. Point mutation is more plausible – it mutates a single node (with certain constraints so as to not introduce syntactic errors). Mutating a “point” in an object graph is more difficult as points can be entire objects, features of an object, or associations between objects. Each of these points would need great care to ensure that mutating them doesn’t make the model invalid. Data mutation (i.e. mutating an object’s attribute values) would be possible, but careful consideration is needed for how to resolve the mutation of the meta-class of an object.

There are variants of GP that produce object oriented code, which is conceptually similar to models (both can be visualised as object graphs). These approaches however, move away from the tree-based genotype used in GP and use instead a linear genotype to represent programs (i.e.. an integer/bit string-based genotype with a fixed mapping to the phenotype). Basic OOGP [178] uses integer triples to select and combine objects from a resource pool. Basic OOGP requires all objects to be instantiated in advance in the resource pool. Given the size that models can reach and the combinatorial explosion of feature assignments, this would not be practical for models. Another variant of GP is Grammatical Evolution (GE) [151, 127], used by our prototype representation in section 3.3 and [186]). In GE, the phenotypical impact of every gene is dependent on the genes that appear previous in the genotype, particularly if wrapping is used during instantiation (see section 3.4.1). Therefore, if the value of a single gene is altered, it can have dramatic effects on the phenotypical impact of the genes that follow, making the locality of the representation very low. In section 3.3, we showed how destructive genotypic mutation can be. Rothlauf and Oetzel [150] also demonstrate the negative impact that GE’s low locality causes on performance. Furthermore, the representation we used in [186] is unable to express references between objects – a crucial requirement for any representation of MDE models.

*Cartesian Genetic Programming* (CGP) [110] uses a linear, integer-based representation for building graphs of program functions and their inputs and outputs. CGP assigns an identifier to each function and terminal and splits the genotype into segments. The first gene in a segment identifies the function that that segment represents. Subsequent genes in the segment are used to define the inputs and output(s) of the function. This idea can be mapped to MDE models. CGP uses only point mutation during evolution.

One of the requirements of our model representation is to ensure the representation is generic enough to express any model

conforming to any given metamodel. Developers could potentially define their own metamodel-specific evolutionary operators, enabling a direct representation. This would be costly to define, but may overcome some of the issues outlined above. Burton et al. [18] use a direct representation within a multi-objective genetic algorithm used to address acquisition-related problems. They define a crossover operator to swap sets of objects between models to produce children. This operator can result in producing invalid children and the algorithm is required to automatically fix this.

A further drawback of using a direct representation for models is that they can get very large (hundreds of megabytes). Search techniques may become impractical due to the vast memory usage and computation costs needed in cases of large populations.

#### 4.2.2 The Genotype to Represent Models

A *linear* representation of models has the advantage that the many existing linear genotype-based search algorithms, such as genetic algorithms, evolutionary strategies, simulated annealing and hill climbing, could be used ‘out-of-the-box’. This would allow MDE practitioners to easily determine whether different algorithms or different combinations of genetic operators perform better for different problems, without defining new operators for each technique or devising new metaheuristic techniques. Additionally, MDE practitioners can make use of the wealth of existing research into linear genotype-based algorithms [56, 57, 103].

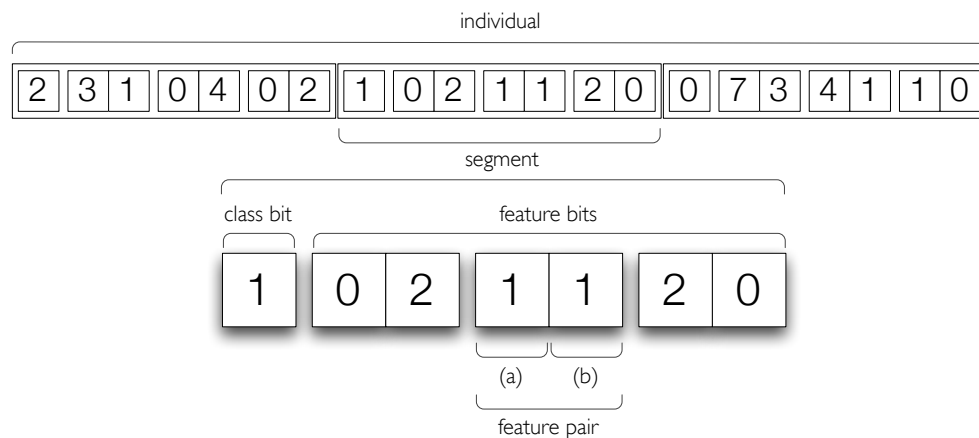


Figure 4.6 shows the genotype that we have defined to encode models. As with CGP, the genotype is divided into a number of *segments*, with each segment representing a single object in the encoded model. The first gene in a segment, the *class bit*, identifies which metaclass is instantiated by the object being represented. Successive genes, the *feature bits*, define the values of features from that metaclass. Feature bits are grouped into pairs: the first member of the pair, the *feature selector bit*, identifies a particular feature of the meta-class; and the second member of the pair,

**Figure 4.6:** The genotype of our representation. Segments are the building block for representing models as integers. (a) is the feature selector bit, and (b) is the feature value bit.

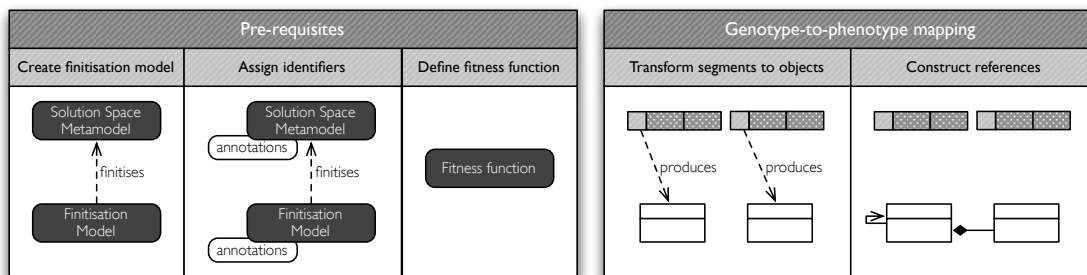
the *feature value bit*, specifies the value that should be assigned to the feature in the object being represented.

The next section describes how this structure, when united with the finitisation model, can represent any MDE model conforming to any given metamodel. Moreover, we will show how this mapping is reversible: i.e. existing models can be re-encoded.

## 4.3 Genotype to Phenotype Mapping

The previous section presented the structure of an individual in our generic representation of MDE models. Each of the integers in the individual is responsible for identifying part of the model being represented – the class bit and feature selector bit represent structural elements to be instantiated, and the feature value bit represents the assignment of values to a feature. To define the mapping from genotype to phenotype, we need a way to identify not only elements in the metamodel being instantiated, but also the data finitisations defined in the finitisation model. Once all relevant elements have been assigned identifiers, we can start transforming an individual into a model.

The steps taken to transform an individual into a model (the mapping from genotype to phenotype) are as follows. Steps one and two occur only once, before a search algorithm begins. Steps three and four take place for every individual evaluated by the search algorithm.



**Figure 4.7:** An illustration of the processes involved in transforming from our genotype to a model conforming to a given metamodel.

1. Create the finitisation model for the metamodel and assign identifiers to each of the data values assigned to features.
2. Assign identifiers to meta-classes and features defined in the metamodel.
3. Transform each segment in the individual into a model object.
4. Assign references between the model objects.

These steps are illustrated in figure 4.7. A further prerequisite shown in figure 4.7 is the need to define a fitness function; this,

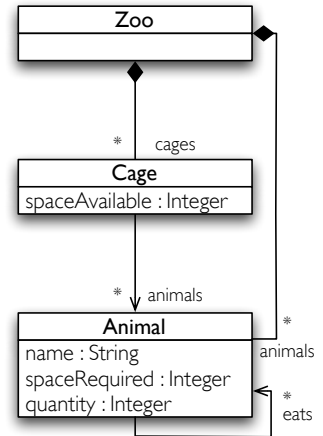


however, is independent of the encoding process. We now describe the genotype-to-phenotype mapping in more detail, using a simple example of instantiating a model of a zoo for illustration. Figure 4.8 shows the metamodel for the zoo, which consists of cages and animals. As previously described, one example usage of this metamodel would be a language that allows zookeepers to assign animals to cages, taking into consideration the fact that certain animals need to be kept apart from others for food chain related reasons.

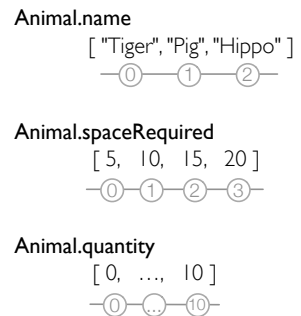
**Step 1: Finitisation** In order to represent the value(s) of a feature, we first define a finitisation model for the zoo metamodel (see figure 4.9, which is simplified to make annotation legible) as described in section 4.1. Each of the FeatureFinitisations have identifiers assigned to each of their data values so that it is possible to reference them during the mapping stage. Values declared as a list finitisation are assigned an identifier based on their index in the list; values specified as a range finitisation are assigned an identifier based on their index in the range (based on the stated step).

**Step 2: Metamodel identifiers** In order to reference meta-elements (i.e. classes and features in the metamodel) from the genotype representation, we automatically assign identifiers to them (see figure 4.10). Each meta-class is given an identifier, and each of its meta-features are also assigned identifiers, unique to that meta-class only. Meta-elements that cannot be instantiated, such as enumeration types, are not assigned a class-level identifier, however the enumeration literals are assigned feature identifiers. Abstract classes are not assigned a class-level identifier, however the subclasses of abstract classes assign identifiers to their inherited features. Furthermore, we label the root class as defined in the finitisation model. In this example, the root class is not assigned an identifier. In some cases the root class may be assigned an identifier – for instance in UML class diagrams where packages can contain sub-packages.

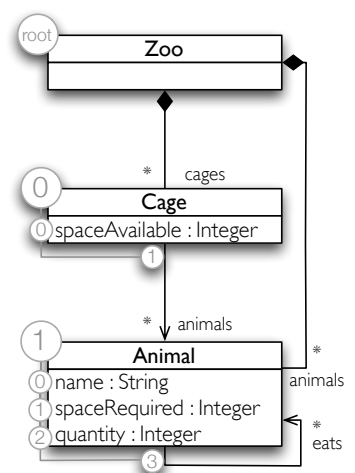
**Step 3: Segment mapping** A model object is created by translating a segment using the definition in figure 4.6, and the identifiers defined in figures 4.9 and 4.10. Figure 4.11 shows how we resolve the segment’s class bit (1) to the Animal class; so we create an Animal object. Next, we resolve each of the feature pairs. In this example, we assign the Animal object the name “Tiger”, set its spaceRequired feature to 5, and its quantity feature to 2. Each of the values in the segment is interpreted modulo the number of valid values (i.e. the number of meta-classes for a class bit, or the number of features in a certain meta-class for a feature bit). When the feature selector bit maps to a reference feature (i.e. not an attribute), we take note of the feature pair and move on to the next.



**Figure 4.8:** A simple metamodel for describing the layout of zoos. (Repeated from figure 2.2.)



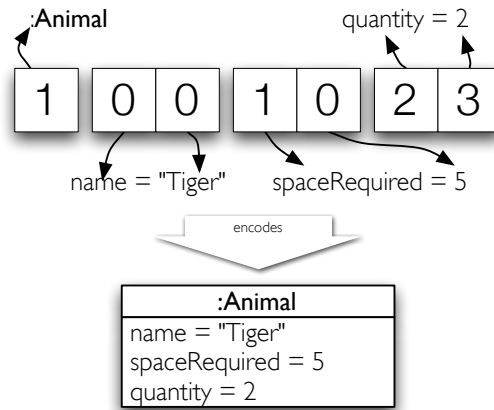
**Figure 4.9:** Step 1: Finitisation information for the zoo metamodel.



**Figure 4.10:** Step 2: Assigning identifiers to the zoo metamodel.

If we chose to assign the target of the reference at this point, we could only select from objects that have previously been created during the mapping process. This would bias against any objects produced later in the mapping, and so we wait until all objects have been instantiated before assigning references.

**Figure 4.11:** Step 3: Mapping a segment to an Animal object.



A segment may contain multiple feature pairs that select the same attribute. To deal with this, we could do one of three things. Firstly, we could ignore any subsequent attempts to assign an attribute value. Secondly, we could allow reassignment, keeping the last-assigned value. Or finally, we could adjust the feature selector bit so that it assigns a value to a different feature. The latter solution might introduce more clashes with other feature pairs in the segment, and may affect the metaheuristic algorithm by implicitly mutating an individual. The choice between the first and second solution should not cause any problems with the metaheuristic, providing the choice is consistent. Any feature pairs that are ignored are said to be *non-coding* feature pairs (discussed in detail in section 7.4). Our current implementation of the representation (section 4.5) uses the second solution.

Once all segments have been translated into model objects, we instantiate one instance of the root class to act as the container for the other objects.

**Step 4: Constructing references** Once all segments (ignoring references) have been mapped to objects, and those objects instantiated, we can then assign the references. This is analogous to the linking phase often used during code compilation. The first phase of constructing these references is to assign all possible objects to be direct children of the root object (see figure 4.12a). In the zoo example we assign all animals and cages to the appropriate references in the zoo root object. This aims to increase the likely hood of as many objects appearing in the final model as possible. Any objects which are not directly contained by the root at this time will be left floating until another object takes ownership of them.

The second phase then iterates through all of the unassigned

feature pairs which were saved during step 3. To assign a reference, we select the set of instantiated objects whose type (metaclass) is the same as the reference target's type, and then take the feature value modulo the size of the set to select one of those objects (see figure 4.12c). Objects can only be contained by one other object (i.e. have one containment reference directed at them). If object  $c$  is already contained by object  $p1$  and a feature pair assigns  $c$  to  $p2$ , then  $p2$  becomes the container of  $c$  (see figure 4.12d). This allows objects to take ownership of objects that had been previously assigned to the root object in the first phase (e.g. in the case of subpackages taking classes).

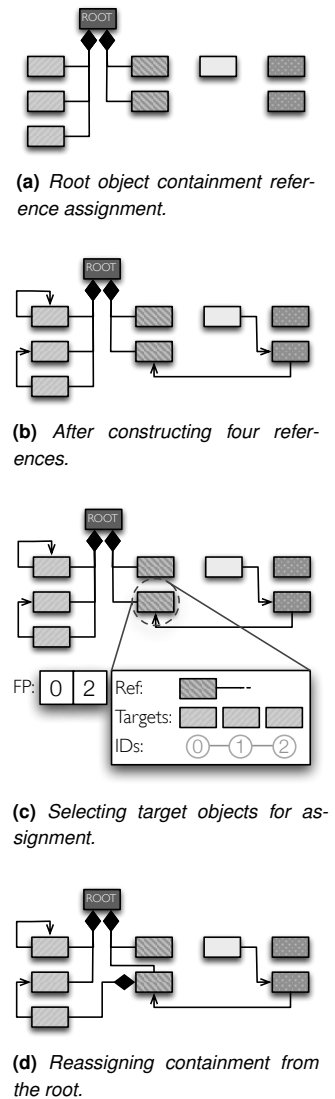
If the finitisation model finitises a reference with objects in another model then we would instead select from those as we would for attributes. If, however, the finitisation model defines some inter-model objects and those objects also conform to the same metamodel as the model being generated, we select the target object for assignment from the combined set of all possible target objects. If the reference has already reached its upper bound, we ignore that feature pair. Once all feature pairs have been resolved, there may be instantiated objects that remain floating, i.e. those that were not assigned to any container references and so are not directly or indirectly contained by the root object. These objects are deleted, and therefore introduce redundancy in the genotype. (This is discussed further in our evaluation of the representation in section 7.4 and figure 7.15.)

The result at the end of this process is a model that conforms to the input metamodel and consists only of data defined by the user in the finitisation model. This genotype and its mapping is capable of encoding all models that conform to a given metamodel. To add new objects, one simply adds more segments to an individual; to assign more features to an object, simply add more feature pairs.

### 4.3.1 And Back Again

The genotype-phenotype mapping can easily be reversed in order to transform an existing model into its genotypic representation. This can be useful for a number of reasons. For instance, if we wanted to optimise an existing model we could transform it to its genotypic form and, for example, apply a hill-climbing algorithm. Or perhaps we might like to seed the initial population of an evolutionary algorithm with existing models that have high fitness in order to kick start the search algorithm.

The phenotype-genotype mapping shares the first two steps of the genotype-phenotype mapping, i.e. it still requires a finitisation model and meta-element identification. It is crucial that the meta-element identification process is deterministic, otherwise mapping a model to and from its genotype may result in a



**Figure 4.12:** An illustration of the reference construction phase.

different phenotype. Steps three and four differ to the genotype-phenotype mapping and are described now.

**Step 3: Reverse object mapping** The process of creating a segment from an object is the inverse of the process of creating an object from a segment. A segment is created for each object in the model. We then iterate through the objects. The identifier of an object's meta-class is assigned to the class bit of the segment. Next, each structural feature of the object is examined in turn. The identifier of the feature becomes the feature selector bit, and the finitisation model is queried for the feature's value to get the feature value bit. It is up to the implementation of this mapping to decide how to deal with cases where the finitisation model does not specify the value being looked up, e.g. throw an error or ignore that feature entirely. As with the genotype-phenotype mapping, if we encounter a reference feature, we wait until all objects have been mapped to (partial) segments.

**Step 4: Constructing references** Once all objects have been mapped to segments we can assign the references. Each unassigned reference is sequentially assigned by selecting the collection of segments whose equivalent model objects can legally be assigned to that reference. The feature value bit is assigned the value of the index of the object in that set.

In the next section, we discuss various search-related properties of the representation and describe the set of genetic operators that have been defined for this representation.

## 4.4 *Search Properties and Adaptation Operators*

This section describes the search-specific characteristics of the representation – the adaptation operators (see section 2.2.1) and method of initialising the search algorithm (by constructing one or more initial solutions). As mentioned earlier, because it is based on a linear genotype, our representation can be used by any linear genotype-based search algorithm. Each search technique would use an appropriate initialisation method (population or single-state) and select suitable adaptation operators (e.g. selectorecombinative GAs use only crossover, not mutation).

Section 4.4.1 describes how to initialise individuals in the representation. Section 4.4.2 describes how the standard crossover and mutation operators are used with the representation, and section 4.4.3 introduces some custom, representation-specific operators.

#### 4.4.1 Initialisation

In the general case, the initial search population of a population-based algorithm (such as a genetic algorithm) is generated by assigning random values to each gene. This would spread the initial population randomly about the search space, allowing a wide area to be explored. The selection of genes could be performed in a more constructive way, so as to distribute the population evenly. Certain applications, however, may require a different method. For example, it may be beneficial to *seed* the population with variants of an existing model (using the phenotype-to-genotype transformation) if the goal is to optimise a specific model. Similarly, for a single-solution-based search algorithm, a random starting point can be chosen or an existing model could be mapped down to its genotype and used.

The length of an individual ( $|I|$ ) plays an important role in determining the model space that can be represented. It is defined as follows:

$$|I| = \sum_{i=1}^n |S_i| \quad (4.1)$$

where  $|S_i|$  is the length of the  $i$ th segment and segment length is defined as:

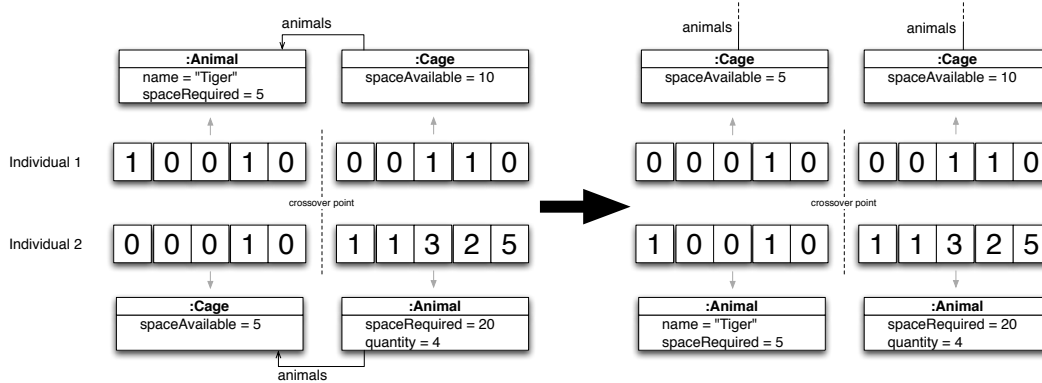
$$|S| = 1 + (2 * \#fp) \quad (4.2)$$

where  $\#fp$  is the number of feature pairs in that segment.

The segment length limits the number of structural features an object can instantiate, and the number of segments determines the overall size of the model. To allow for a diverse population, the number of feature pairs can be allowed to differ between segments. This reduces redundancy in the segments which represent meta-classes with few features, potentially inhibit some features from being assigned values in meta-classes with a large number of features. We analyse the redundancy of the representation in section 7.4 and find that the metamodel being represented heavily influences the redundancy of the representation.

#### 4.4.2 Standard Adaptation Operators, Adapted

The linear genotype described in section 4.2 permits standard genetic operators, such as a single-point crossover, to be used. However, the structure of the genotype in our representation means that crossover can be particularly destructive if it is allowed to occur at any point in the individual. If crossover is too destructive, it can be detrimental to the search as it can introduce too much variation into the population as potentially useful genetic traits are not carried forward to the children. To reduce the amount of destruction, we only allow crossover to occur at the points between segments. This allows entire objects to be copied over to

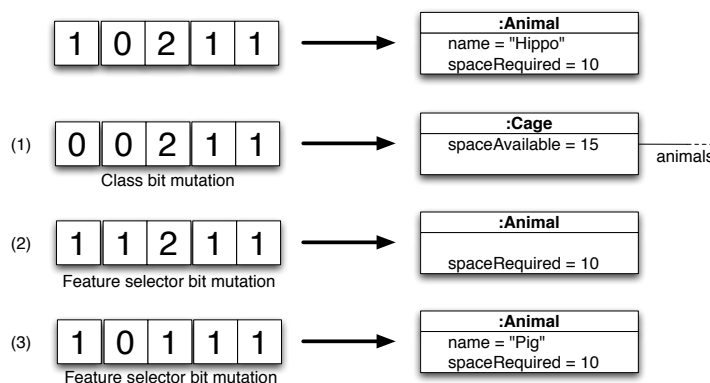


**Figure 4.13:** Illustration of the crossover operator.

the child being created – only the feature pairs that represent references will be affected. Figure 4.13 illustrates the crossover operator. The animals references in the Cage objects are reassigned to other Animal objects in the model (not shown). If there were no Animal objects in the model, then those feature pairs would become redundant.

The mutation operator also differs slightly from the usual mutation operator used with linear genotypes. Figure 4.14 demonstrates how mutating different types of gene causes different amounts of variation in the phenotype. Mutating the class bit has the most significant effect on the phenotype, whereas mutating either the feature selector bit or feature value bit has a lesser effect on the phenotype. Therefore each gene type has a different probability of being mutated (specified by the user). This is known as *positional dependence* [25].

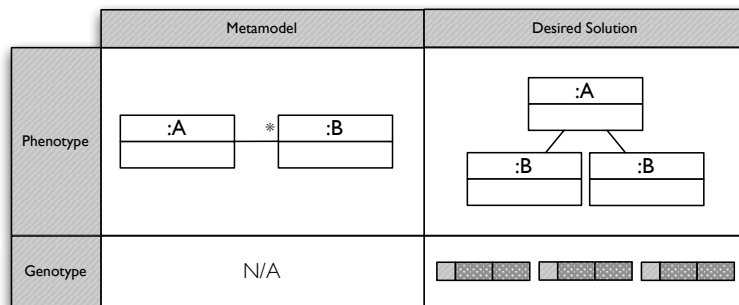
**Figure 4.14:** Illustration of the mutation operator.



#### 4.4.3 Custom Adaptation Operators

Initialising a population with variable-length individuals using variable-length segments aims to produce a diverse, representative population. However, it does not guarantee that the individuals in the population will be able to represent the desired solution. If individuals are much longer than the solutions, the search algorithm would need to find an individual which has a

large amount of redundancy, which may be challenging or even impossible. For example, if the user has not defined scopes for meta-classes or references, a situation may arise where it is not possible to make a segment redundant because it is always able to be contained by another object. Figure 4.15 illustrates this. The encoding of the metamodel shown cannot be redundant as all Bs can always be contained by an A object. If the solution requires just two B objects to be referenced by a single A object, then any individual who has more than three segments is unable to encode the solution. Obviously, this example here is contrived, but these cases may arise without the awareness of the user and so care is needed when defining the finitisation model.



**Figure 4.15:** An illustration of a metamodel which, without proper finitisation, will result in a representation with no redundancy. An individual with more than three segments will be unable to encode the solution.

As a result of this phenomenon, we need to provide the ability to increase and decrease the lengths of individuals. Therefore, we define two new genetic operators for our representation:

*Segment Creator* inserts a randomly created segment into a random position in an individual.

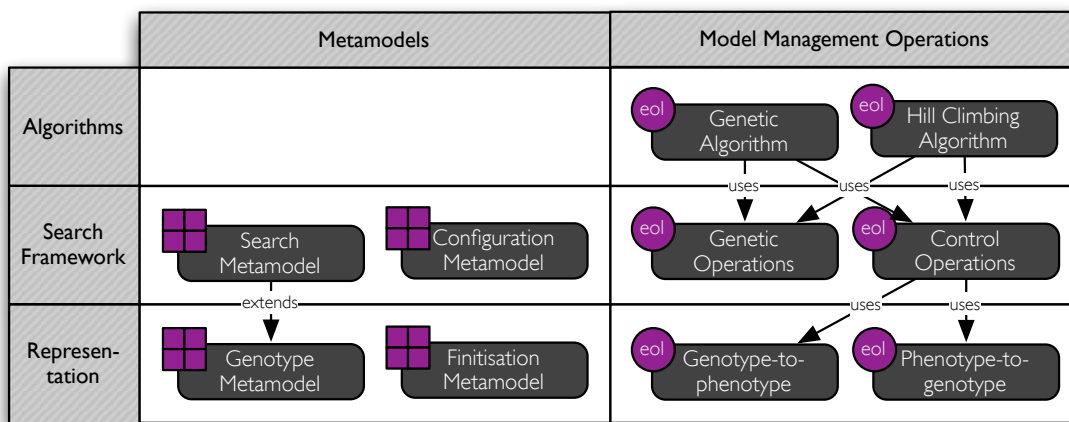
*Segment Destroyer* deletes a randomly selected segment from an individual.

The adaptation operators presented in this section aim to be beneficial to the search by not being too destructive. As such, our crossover operator transfers entire objects between models, and we split the mutation operator into gene-specific operators, allowing the user to have fine-grained control over the operators being applied. We have also defined some representation-specific operators which allow the size of the models to be altered. The selection of these operators will entirely depend on the search algorithm being used and problem being addressed.

## 4.5 *Crepe and MBMS:* *The Realisation for Ecore*

This section describes our implementation of the model representation presented in the previous sections. We target the Eclipse

Modeling Framework (EMF) due to it being the most widely used modelling platform at present. This means that our implementation can represent any model whose metamodel is defined using *Ecore* – the metamodeling language of EMF. As such, our implementation is called *Crepe*: the Canonical *RE*presentation for *Ecore*. Furthermore, we use *Crepe* as the representation for a bespoke model-based metaheuristic search framework that targets MDE problems. Every key component in the search framework is expressed as a model and all operations are defined as model management operations (MMO) written in the Epsilon Object Language (EOL) [92]. EOL was chosen because it is a robust, general-purpose model management language that supports the manipulation of multiple models simultaneously.



**Figure 4.16:** The building blocks of the model-based metaheuristic framework that uses our representation at its core.

Figure 4.16 depicts the structure of our model-based metaheuristic search framework, highlighting the models and MMOs used in each layer. Each important aspect of the framework is decoupled, making it easy to define new algorithms and tailor existing functionality. In this section we describe each layer of the framework from bottom to top. Section 4.5.1 describes the implementation of the genotype metamodel and genotype-phenotype mappings. Section 4.5.2 presents *MBMS*, an extensible *Model-Based Metaheuristic Search* framework that uses *Crepe* as its representation. Section 4.5.3 describes how metaheuristic algorithms are implemented in *MBMS* and overviews the two currently implements: a genetic algorithm and a hill climbing algorithm. Finally, section 4.5.4 illustrates one of the benefits of a model-based metaheuristic search framework by presenting a tool that provides an interactive visualisation of a search space.

#### 4.5.1 *Crepe: Implementing the Representation*

There are three main components of our representation: the structure of an individual, the finitisation information, and the map-

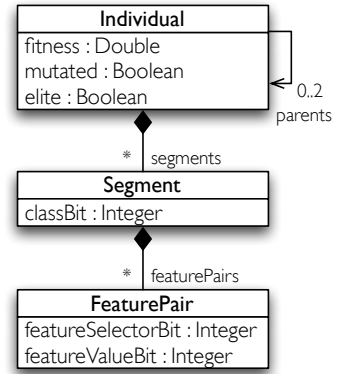


pings between genotype and phenotype. We address each in turn.

**The Genotype Metamodel** Figure 4.6 in section 4.2 illustrated the structure of an individual in our representation. In Crepe, we have captured this structure in the metamodel shown in figure 4.17. Individuals possess three search-related attributes: the fitness of the individual (assigned once the candidate has been evaluated); a flag saying whether the individual has been mutated; and a flag expressing whether or not the individual is an elite that has been carried forward from a previous generation. Individuals also have references to their parents – the individual(s) in the previous generation that bred to produce them. This provides some traceability through the search algorithm, allowing the user to see the workings of the selection and breeding operations (e.g. for validation purposes). If the representation is being used for something other than search (see section 8.2), the three attributes and parents reference can be ignored. Individuals are composed of Segments, which have a single integer attribute to express the class bit. Segments contain an unbounded number of FeaturePairs, which define the feature value and feature selector bits.

**The Finitisation Metamodel** To express the data and structures that can be encoded by the representation, the user must define a finitisation model. The metamodel for such models was shown in figure 4.2. We have implemented this metamodel in Ecore: where figure 4.2 shows a type as Element, Class, or Property, our implementation uses Ecore’s EObject, EClass, and EStructuralFeature respectively.

**Genotype-Phenotype Mappings** As both the genotype and phenotype are models, the genotype-phenotype and phenotype-genotype mappings are implemented as model-to-model transformations written in EOL. Initially, these transformations were written in Java. Rewriting them in EOL reduced the number of lines of code by ~90% and made the code much easier to maintain. This came at a cost of performance, however, as EOL is language that is interpreted by Java. At this stage, finely tuning the performance of the representation and MBMS is not critical: we are using it to demonstrate the feasibility and practicality of a generic model representation. The genotype-phenotype mappings described in section 4.3 rely heavily on the identifiers assigned to meta-elements. Ecore does in fact automatically assign identifiers to each meta-class and meta-feature. However, Ecore assigns identifiers to non-instantiable classes, such as abstract classes, enumerations and data types. To use Ecore’s identification scheme would introduce a lot of redundancy into the representation as all segments whose class bits reference non-instantiable classes would have to be ignored. To overcome this, we compute the set of instantiable classes at the start of the execu-

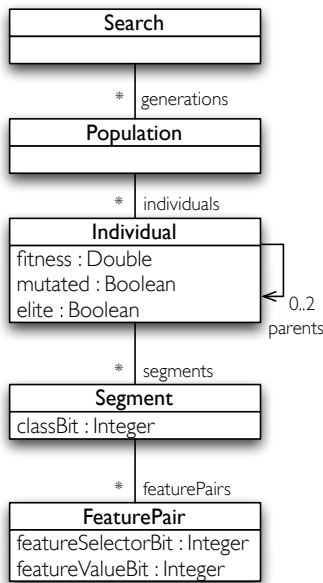


**Figure 4.17:** The metamodel that defines the structure of individuals in Crepe.

tion, order them by their Ecore identifier, and assign them a new identifier for use with the mapping based on their position in the ordered list. This also means that we don't assign identifiers to classes that are ignored in the finitisation model. An analogous process occurs when selecting meta-features to instantiate – only those that *can* be instantiated are assigned identifiers.

#### 4.5.2 MBMS: Implementing the Search Framework

To use Crepe to perform metaheuristic search, we have implemented *MBMS*, a model-based metaheuristic search framework. As shown in figure 4.16, the core of MBMS defines two meta-models and two MMOs. We address each of these in turn before bringing all aspects together to describe the usage of MBMS.



**Figure 4.18:** The meta-model used to capture the execution of a search algorithm.

**The Search Metamodel** The metamodel for the models that capture a single execution of a search algorithm is shown in figure 4.18. A single Search contains a sequence of Populations, which represent the generations of the search. A Population is made up of a number of Individuals – as defined in the genotype metamodel. For population-based algorithms, the Population object will contain many individuals. For local search algorithms, it will just contain one.

**Configuring the Search** A common practice for Java-related projects is to define configurable parameters in properties files (with the file extension ‘.properties’). Properties files provide a compact way to capture key-value pairs. An example properties file for an evolutionary algorithm might look like as follows:

```

1 population.parents.size = 8
2 population.children.size = 6
3 population.elites.size = 2
  
```

**Listing 4.1:** Example property definitions for configuring an evolutionary algorithm.

We considered defining a metamodel to capture the properties for configuring our search algorithms, but the overhead of this would be too costly. In particular, the current concrete syntax of properties files is familiar to Java users, and easy to pick up for non-Java users. A graphical syntax for the configuration would be too cumbersome for simply defining pairs of strings. Furthermore, Java provides a well-defined API for managing properties files – something which we would have to implement ourselves were we to define a custom metamodel. Instead, we decided to use properties files directly, but *treat them as models*.

As presented in section 2.1.2, Epsilon is agnostic to modelling technologies. Through its *model connectivity layer* (EMC), Epsilon supports many kinds of models: e.g. EMF, XML, MDR, BibTeX, Z, and spreadsheets. Models from different technologies can be input into Epsilon programs and seamlessly interact. Therefore,

Parameter	Possible Values
CONTROL PARAMETERS	
termination.iterationsThreshold	INT
termination.fitnessThreshold	DOUBLE
REPRESENTATION PARAMETERS	
individuals.segments.quantity.max	INT
individuals.segments.quantity.min	INT
individuals.segments.featurepairs.max	INT
individuals.segments.featurepairs.min	INT
individuals.maxallele	INT
POPULATION-BASED ALGORITHM PARAMETERS	
population.size	INT
population.initialisation.type	random, given, last
population.parent.size	INT
population.parent.selectiontype	random, tournament, copy, fittest
population.elite.size	INT
population.elite.selectiontype	random, tournament, copy, fittest
population.children.size	INT
population.children.mutation.type	uniform, weighted
population.children.mutation.uniform	[0.0 ... 1.0]
population.children.mutation.weighted.cb	[0.0 ... 1.0]
population.children.mutation.weighted.fsb	[0.0 ... 1.0]
population.children.mutation.weighted.fvb	[0.0 ... 1.0]
population.children.construct.segments.probability	[0.0 ... 1.0]
population.children.destruct.segments.probability	[0.0 ... 1.0]
population.selection.tournamentsize	INT

**Table 4.1:** The set of default parameters for use in MBMS.

we have defined an EMC *driver* for properties files, allowing them to be treated like models and be input to any Epsilon module and manipulated using EOL. This means that properties are able to be configured at runtime and are also able to be persisted to disk. Listing 4.2 shows a simple example of using EOL to read from a properties file.

```

1 var parentPopulationSize = Property.all.selectOne(p |
2   p.key == "population.parents.size");

```

**Listing 4.2:** An illustration of reading from a properties file with EOL.

Table 4.1 lists the properties that we have defined in MBMS to control the search algorithms (new implementations of search algorithms can either use existing properties or define their own). The properties are divided into three categories:

*Control parameters:* These parameters control the execution of the metaheuristic search algorithm. We have defined two parameters that both control termination of the algorithm: when the fitness of an individual reaches a certain threshold, and when the maximum number of iterations of the algorithm has been reached.

*Representation parameters:* These parameters configure the size of individuals. In particular these parameters are used when creating individuals, such as during the initialisation of a search algorithm.

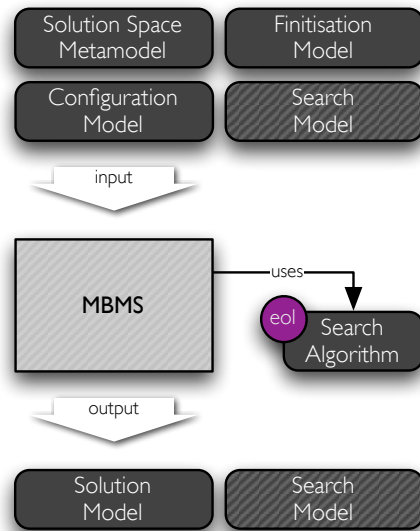
*Population-based algorithm parameters:* These are the parameters common to population-based search algorithms, and in particular by the genetic algorithm introduced in the next section. Of particular note is the `population.initialisation.type` property which determines how the population of the GA should be initialised. Due to the fact that we store the search history as a model, we're able to start an execution from an existing population, either by providing an explicit `Population` (in its own input model) or by continuing from the last population in the input search model. Alternatively, the initial population can be generated randomly.

**Control Operations** This MMO defines three important functions: initialising a search algorithm, checking if the termination criteria has been met, and delegating the fitness evaluation of an individual. The initialisation function is defined with respect to population-based algorithms; non-population-based algorithms should define their own initialisation function. The function examines the specified initialisation parameter (`population.initialisation.type`) and acts accordingly. If the initial population is given, then it looks for a model called 'POP' and copies the last `Population` object into the search model. If the initialisation type is specified as `random`, the representation parameters (table 4.1) are used to generate random individuals. Otherwise the last generation in the search model is used.

The termination function reads from the configuration model (properties file) to determine whether the termination criteria has been met. If so, the search algorithm will cease. The evaluation function transforms an individual into its phenotypic form (using the genotype-to-phenotype module described above), and delegates the evaluation to a user-defined function. This is described further in section 4.5.3.

**Genetic Operations** This MMO defines the adaptation operations described in section 4.4: mutation, crossover, segment creation and segment destruction. Furthermore, it implements a selection function that is used to pick out individuals from a set with respect to specified criteria. For example, it is used to select the fittest individuals from a population in order to produce the next generation of individuals, and to select the elite individuals from the current generation.

**Using MBMS** To use the MBMS framework, the user needs to specify the *metamodel* of the solution space, the *finitisation model*



**Figure 4.19:** A conceptual view of the inputs to and outputs of MBMS. The diagonally shaded box represents an optional input.

for the solution space metamodel, the *configuration model* (properties file) and, optionally, and existing *search model*. This is illustrated in figure 4.19. As a result of executing the search algorithm, MBMS produces the fittest solution discovered – a model conforming to the solution space metamodel – and optionally a model that captures the entire history of the search algorithm. The search model can then be visualised to gain insight into the search space, or to perform a manual validation of the search algorithm, as described in section 4.5.4.

The genetic operations, control operations, and search algorithm have been implemented separately to allow users to tailor MBMS to their needs. New search algorithms can be defined, or existing algorithms refined, by importing these modules and overriding one or more genetic or control operations.

### 4.5.3 Implementing Search Algorithms

As with the genotype-phenotype mappings, metaheuristic search algorithms are written in EOL. Each algorithm is defined in its own EOL module, and can import any required modules, such as the pre-defined genetic operators. Currently, we have defined a genetic algorithm and a hill climbing algorithm as proof of concept.

To use one of the algorithms, one needs to create a custom EOL module and import the particular search algorithm module. After any custom set up code, the user simply invokes the `commence` operation and the algorithm begins (assuming all input models have been specified by the launch configuration). For example:

```

1 import "platform:/plugin/jw.research.crepe/mbms/algs/
   geneticAlgorithm.eol";
2
3 /* custom setup code */
4
5 /* start the GA */
6 commence();

```

**Listing 4.3:** *Executing a genetic algorithm.*

Once the current generation's population has been produced (either initialised or bred), the evaluate control operation is invoked. As mentioned earlier, this automatically transforms each member of the population into their phenotypic form, by utilising the genotype-to-phenotype module. Once transformed, the evaluation of each candidate solution's fitness is delegated to a user defined operation named `evaluateFitness`:

```

7 operation evaluateFitness(candidate :MM!EObject) : Real {
8   var fitness = ...
9   return fitness;
10 }

```

**Listing 4.4:** *Defining a custom fitness function.*

Defining the fitness function in EOL gives the user full access to a powerful model management language. However, this may not be enough to evaluate a model in every scenario. Fortunately, EOL has the ability to invoke native Java code<sup>3</sup>, meaning that users can also define fitness functions in Java and delegate the evaluation. This allows users to utilise other components, such as simulators, to calculate the fitness of candidate solutions. An example fitness function defined in Java is shown below:

```

1 package com.example.fitness;
2 import org.eclipse.emf.ecore.EObject;
3 public class Evaluator {
4   public double evaluate(EObject candidate) {
5     double fitness = ...
6     return fitness;
7   }
8 }

```

**Listing 4.5:** *Defining a fitness function in Java.*

The evaluate method takes an EObject as its parameter (as all EMF objects inherit from EObject) and the user would need to cast it to the root class type. Alternatively the evaluate method's candidate parameter can be specified to be the type of the root object in order to avoid this casting. Example EOL code used to invoke this Java method is:

```

9 operation evaluateFitness(candidate :MM!EObject) : Real {
10   var evaluator = new Native("com.example.fitness.Evaluator
   ");
11   return evaluator.evaluate(candidate);
12 }

```

**Listing 4.6:** *Delegating the evaluation to a Java method.*

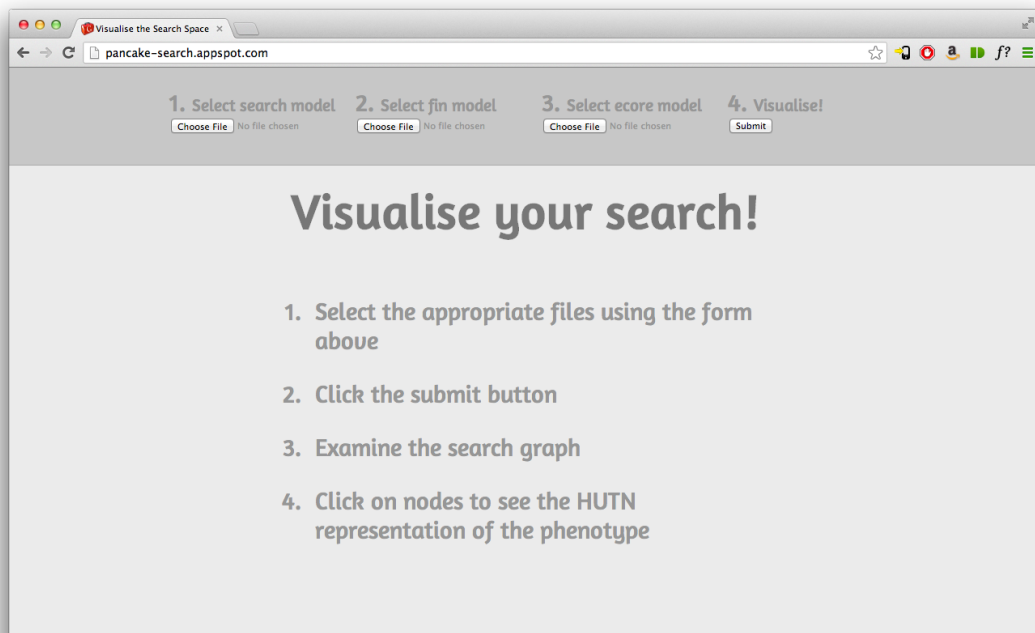
With the metaheuristic algorithm module imported, the fitness evaluation operation defined, and the appropriate input and out-

<sup>3</sup> See [www.eclipse.org/epsilon/doc/articles/call-java-from-epsilon/](http://www.eclipse.org/epsilon/doc/articles/call-java-from-epsilon/)

put models configured, the user can then execute their EOL module.

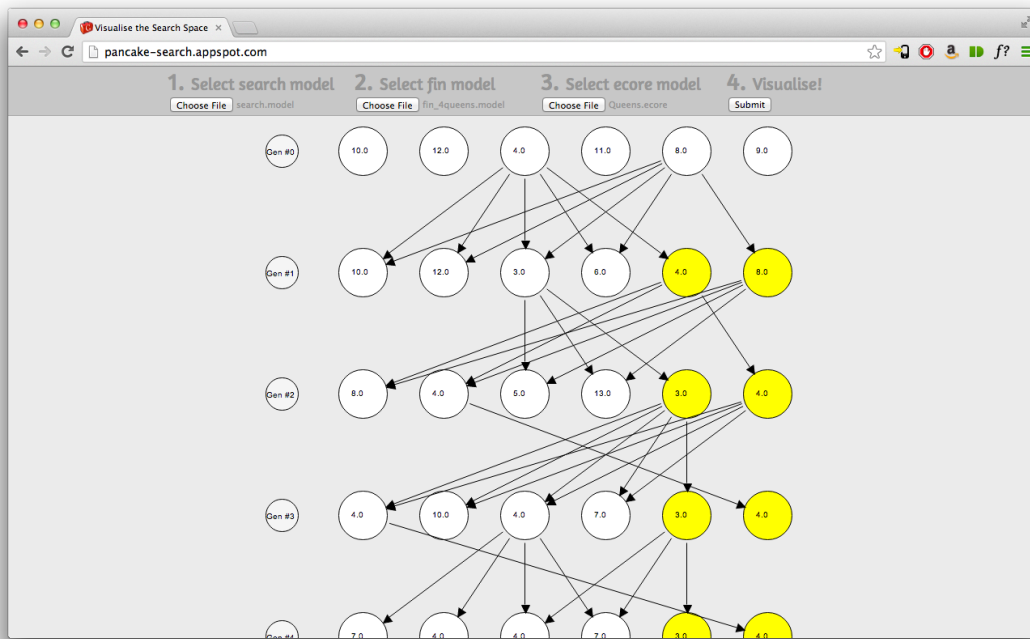
#### 4.5.4 *Visualising the Search*

In section 4.5.2 we stated that the user could optionally provide an existing search model from which to commence the search algorithm, and optionally save the search model once the algorithm has completed. One benefit of saving the search model is that we have an entire trace of the search algorithm which we can inspect in order to gain insight into the algorithm. Developers of new search algorithms can use this to validate that their algorithm is performing correctly, and users can inspect the search history to understand how the search progressed through the solution space. Due to the fact that the search history is captured as a model, we can apply state-of-the-art MDE operations to support its analysis.



**Figure 4.20:** *The initial screen presented to the user.*

As a starting point we have implemented a web-based search model visualiser. Figure 4.20 shows the initial screen which the user is presented with. The user uploads their search model, the metamodel of the solution space, and the finitisation model. The search model only contains genotypic information and so the other two models are required to produce the phenotypes. The web application inputs the search model into a model-to-text transformation written in the Epsilon Generation Language (EGL) [143] which produces some HTML that displays a graph-like representation of the search – illustrated in figure 4.21. Each



**Figure 4.21:** *The search model visualised as a graph.*

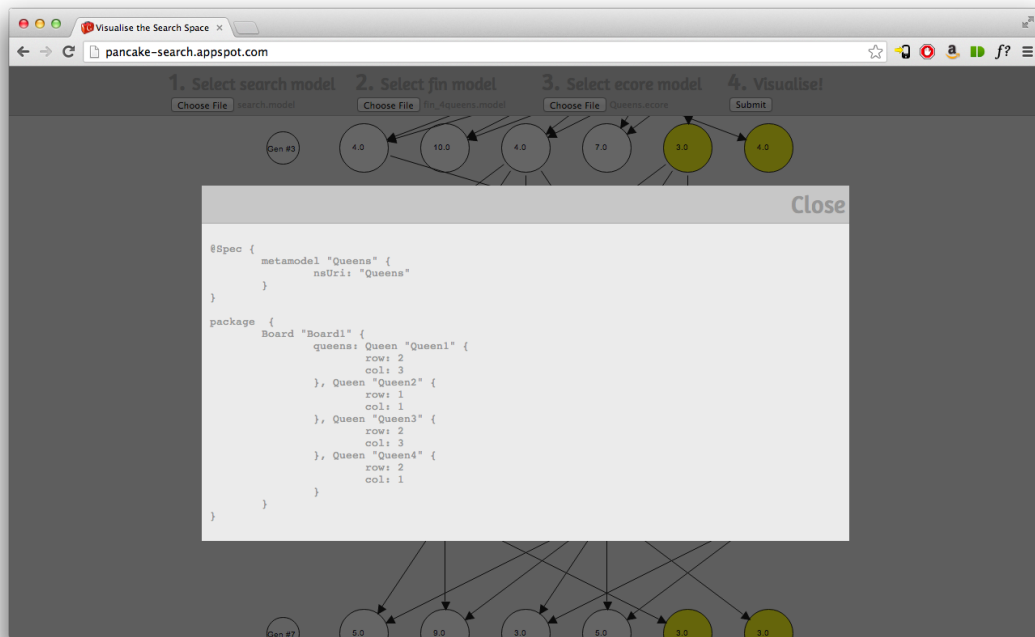
row in the graph represents a single generation, and each node represents an individual in that generation, annotated with their fitness score. To highlight the breeding process, parents are connected to the children they produced. Finally, the elites in each generation are highlighted in yellow.

To inspect a candidate solution, one simply clicks on a node. This sends a request to the server, causing the selected individual to be transformed to its phenotypic form and is input to a second model-to-text transformation. As we know nothing about the concrete syntax of the model, we transform it into the *Human-Usable Textual Notation* (HUTN) [144] – a generic, textual concrete syntax capable of expressing any model conforming to a MOF metamodel. The HUTN form of the model is presented to the user in a popover, illustrated in figure 4.22. An alternative would be to present the user with a downloadable model which can then be viewed using the appropriate editor. This would provide instant feedback to the user and is left to be addressed in future work.

### Summary

This section has presented our EMF-based implementation of our model representation, along with the implementation of an extensible model-based metaheuristic search framework. Each component in the representation and search framework is either a model or a model management operation. The next section presents a





**Figure 4.22:** An individual transformed into its HUTN format and presented to the user for inspection.

simple worked example to illustrate Crepe and MBMS further.

## 4.6 Worked Example

This section demonstrates how someone would go about modelling their software engineering problem and using our representation to search for a solution. To this end, we model a variant of the famous Travelling Salesman problem (TSP), define a finitisation model and illustrate the mapping from genotype to phenotype. TSP is selected as it is a well known combinatorial optimisation problem and so is likely familiar to be with the reader.

### 4.6.1 The Problem: Delivery Man

To demonstrate how one would use the representation for their own problem, we present an MDE implementation of a variant of the Travelling Salesman Problem. TSP is a standard combinatorial optimisation problem in which the goal is to find the shortest path through a set of cities in which each city is visited only once. Our variant contextualises the problem for a delivery company. In each city that a delivery driver visits they need to deliver some goods, and the delivery company want to ensure that they use as little fuel as possible (i.e. take the shortest route).

## 4.6.2 Modelling the Domain

<sup>4</sup>There are a number of ways in which we could model TSP, in this chapter we model the problem in a way that illustrates the concepts of our representation. To efficiently solve this problem, the metamodel used here could easily be transformed into a more optimal encoding.

To model the delivery driver problem<sup>4</sup>, we separate concerns between the cities being visited and the route taken and therefore define two metamodels: one that captures the map, and one that expresses routes over that map. This separates the mapping system from the routing system, allowing the delivery company to produce maps independent of delivery jobs. Furthermore, this also allows us to search for routes for any input map.

Listing 4.7 shows the metamodel for our simple mapping system. A Map consists of a collection of Citys and definitions of Distances between those cities. This metamodel would allow the delivery company to create a model of the cities that they supply. Listing 4.8 shows a metamodel for defining jobs to be allocated to drivers. Each driver is given a Route which consists of an ordered number of Stops at certain cities where the driver is required to deliver some Products.

```
1 package map;
2
3 class Map {
4     val City[*] cities;
5     val Distance[*] distances;
6 }
7
8 class City {
9     attr String name;
10 }
11
12 class Distance {
13     attr int distance;
14     ref City city1;
15     ref City city2;
16 }
```

**Listing 4.7:** *The Emfatic description of the metamodel for defining simple maps.*

```
1 package route;
2
3 import "map.core";
4
5 class Route {
6     val Stop[*] stops;
7     val Product[*] goods;
8 }
9
10 class Stop {
11     ref map.City city;
12     ref Product[*] goods;
13 }
14
15 class Product {
16     attr String name;
17 }
```

**Listing 4.8:** *The Emfatic description of the metamodel for defining routes over map models.*

### 4.6.3 Making the Domain Finite

The finitisation model for the Route metamodel needs to be created with respect to an existing map model. This is because we want to finitise the Stop.city reference with actual city objects from a given map model. We could, therefore, manually create our finitisation model for each map model that we wish to route. However, a more generic approach is to create a simple script that will automatically create this model for us. Listing 4.9 shows such a script, written in EOL. The createEObjectFeatureListFinitisation utility method (not shown) is used to simplify the creation of the finitisation model objects. Similarly the getClassByName method returns a class from the metamodel with the given name.

There are three models that are used in this script: FIN is the finitisation model that is being created by the script; MM is the map metamodel (listing 4.7); and MAP is the particular map model that we wish to route. First of all, the script defines that the Route class is the root object. Secondly, all city objects in the MAP model are added to an EObjectFeatureListFinitisation object for the two Stop features. Finally, as we are currently only interested in the routes, we *ignore* the Product class, removing it from the search. The delivery company could add the product information once the routing has been performed.

```
1 /*
2  * Input models:
3  *   FIN: Finitisation model being created
4  *   MM : The Route metamodel
5  *   MAP: The map model that we select cities from
6  */
7 var fin = new FIN!MetamodelFinitisation;
8
9 fin.rootClass = getClassByName("Route");
10
11 fin.finitisations.add(createEObjectFeatureListFinitisation("
12   Stop", "city", MAP!City.all));
13
14 fin.ignores.add(getClassByName("Product"));
15 fin.ignores.add(getFeatureByName("Stop", "goods"));
```

**Listing 4.9:** An EOL script to automatically produce the finitisation model for the Route metamodel.

This script can be incorporated into the route search script and executed before starting the search meaning that we never need to actually store the finitisation model and making it easier to search in a map-independent manner (as in section 4.6.5).

**Finitising the Map Metamodel** Instead of using pre-defined maps, one could use our representation to generate random map models (e.g. for testing purposes). The finitisation model is fairly simple – City.name could be finitised with a FeatureListFinitisation of random strings, and Distance.distance with a FeatureRangeFinitisation. Although search could be used to discover maps with interesting features, an alternative would be to simply generate

random individuals and transform them to map models.

#### 4.6.4 Defining the Fitness Function

The goal is to find the shortest continuous path through all cities that returns to the first city visited. The search algorithm will be creating N Stop objects, each of which reference a single (not necessarily unique) City object. These objects are assigned to the Route.stops reference. The fitness is calculated by summing the distances between adjacent cities, including the final city with the first (to complete the route). In our problem specification we have not defined any restrictions that would stop the search producing invalid routes – i.e. those which miss some cities, or visit the same city more than once. Therefore, the search algorithm needs to account for these cases and assign the fitness accordingly.

```
1 operation evaluateFitness(candidate : MM!EObject) : Real {
2   // Punish invalid solutions
3   var missingCities = 0;
4   for (city in MAP!City.all) {
5     if (candidate.stops.select(s |
6       s.city == city).size() == 0) {
7       missingCities = missingCities + 1;
8     }
9   }
10
11  if (missingCities <> 0) {
12    return missingCities;
13  }
14
15  // Now calculate the distance score for valid solutions
16  var distance = 0.0;
17  var isComplete = false;
18  for (i in Sequence{0..candidate.stops.size()}) {
19    var stop = candidate.stops.at(i).city;
20    var nextStop;
21    // Need to get back to the start
22    if (i + 1 >= candidate.stops.size()) {
23      nextStop = candidate.stops.at(0).city;
24      isComplete = true;
25    } else {
26      nextStop = candidate.stops.at(i+1).city;
27    }
28
29    distance = distance + MAP!Distance.all.selectOne(d |
30      (d.city1 == stop and d.city2 == nextStop) or
31      (d.city1 == nextStop and d.city2 == stop)).
32      distance;
33
34    if (isComplete) break;
35  }
36  return 1.asDouble() - (1.asDouble()/distance.asDouble());
37 }
```

**Listing 4.10:** The fitness function used to evaluate candidate routes.

As mentioned in the previous section, we define fitness functions in EOL. We have defined the fitness function for this problem in listing 4.10.

Lines 3–12 consider invalid candidate solutions: those whose routes do not include all cities defined in the MAP model. As we

are couching this as a minimisation problem, the fitness for invalid models is defined as the number of cities that do not appear in the route (line 11). The remaining lines calculate the fitness for valid models in the way previously described. To keep all fitnesses for valid models between one and zero (and therefore distinguish them from invalid ones), the fitness is calculated as:

$$f = 1 - \frac{1}{totaldistance} \quad (4.3)$$

#### 4.6.5 Searching for the Optimal Route

For this example, the only genetic operator that we will use is mutation. As the path size is proportional to the number of cities, we need to have a fixed number of segments and so do not need the creation or destruction operators. The simplicity of the route metamodel also means that there will be no segment redundancy (each segment will represent a Stop object).

As we are only using mutation, we illustrate the search with the hill-climbing algorithm in MBMS. Appendix B.2.1 lists the complete EOL code used to set up and define the search problem. The set of neighbours is generated by duplicating the current solution  $N$  times and mutating each gene with a given probability.

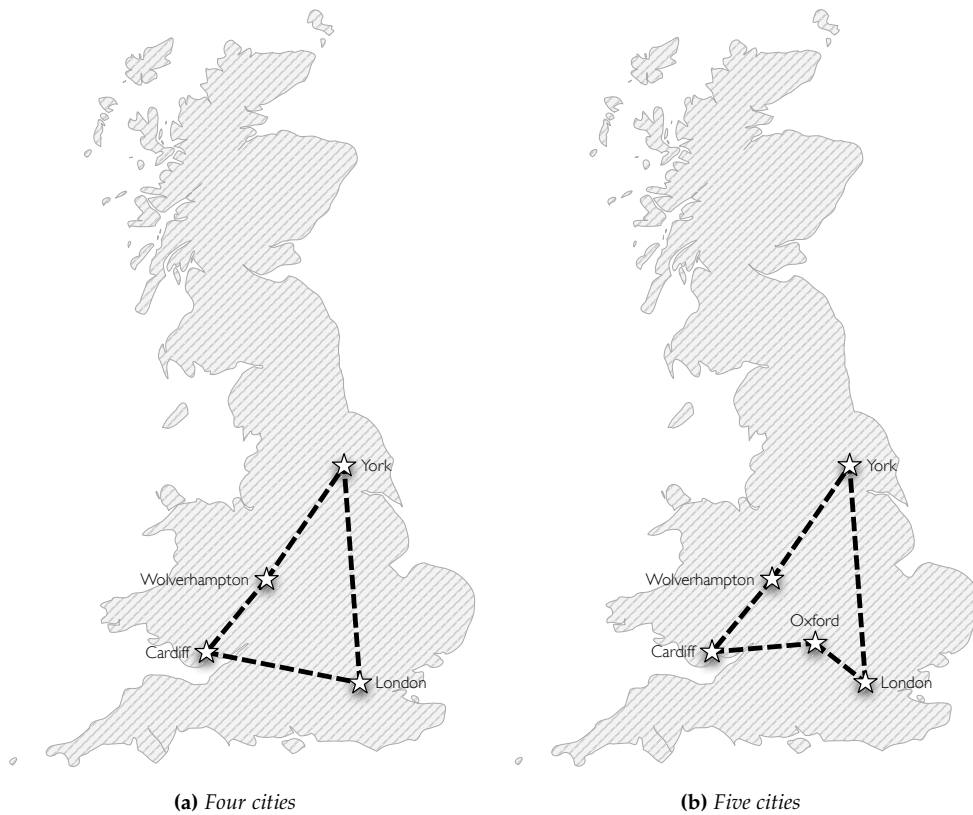
Parameter	Setting
GENOTYPE TO PHENOTYPE MAPPING	
number of segments	4 or 5
number of feature pairs	1
HILL CLIMBING ALGORITHM	
maximum generations	50
number of neighbours	4
reproduction method	single point crossover
mutation method	integer mutation (random value)
mutation probability	0.5

**Table 4.2:** *Parameter settings for the delivery man problem.*

#### 4.6.6 Example: UK Cities

To briefly demonstrate the search, we have defined two simple map models: one with four cities and one with five (see figure 4.23). Table 4.2 shows the parameters we used and figure 4.24 presents the results. The plots show the average results over ten runs, each with a different random seed. On average the solution converges after 40 generations for four cities, and 47 generations for five cities. For completeness we compare the hill climbing algorithm with random search. Random search was implemented by defining the neighbourhood function as generating one totally random individual (see Appendix B.2.2). Figure 4.24 shows that random search does much worse on average. In particular, in

the five cities case, random search is unable to converge on the optimal solution.



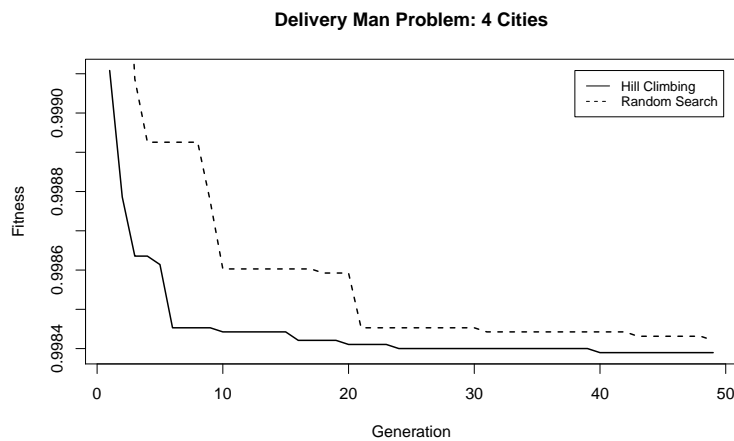
**Figure 4.23:** Visualisation of map models.

### Summary

This section has shown step-by-step how it is possible to model a problem, define a finitisation model and fitness function, and apply search to discovering optimal models. The example used is trivial, and does not utilise every aspect of the representation presented in this chapter (such as the different genetic operators), but the aim was to illustrate rather than explicate.

## 4.7 Discussion

This chapter has presented our generic, canonical, representation for MDE models. We have defined a two-way mapping between structured strings of integers and models that conform to a MOF metamodel. This mapping utilises a user-defined set of data and structural constraints, called *finitisation information*, which is expressed in a model. Commonly the set of models that conform to a metamodel is huge, if not infinite, and so we limit its size using the finitisation model, focusing search algorithms on areas of the model space that we are interested in. Finitisation models



(a) Four cities



(b) Five cities

**Figure 4.24:** Converging on the optimal solution for the delivery man problem.

are problem-specific and different finitisation models may be developed for the same metamodel. Some metamodels will define meta-classes which aren't relevant to all problems. Specifying different root classes in the finitisation model allows users to focus on generating a subset of the elements defined by the metamodel. Furthermore, users can explicitly ignore particular meta-classes or meta-features.

In our representation, every model has a unique, canonical form (with respect to its metamodel and finitisation model). However, models can actually have many genotypic equivalents: different genotypic individuals can map to the same model. This is a result of the representation being *redundant*. We examine this redundancy in chapter 7.

In gaining the ability to transform structured strings of integers to models we are able to make use of the wealth of research into SBSE practices. This means that it is now easy to utilise SBSE techniques to tackle numerous problems found in the MDE domain. As described in chapter 2, there have been some efforts in ap-

plying SBSE techniques to MDE problems, but to date these have used problem-specific representations, or very simple representations. Using our representation means that the user can focus on developing the fitness function to evaluate candidate solutions, and need not be concerned with any other aspects of metaheuristic search.

We have developed a model-based metaheuristic search framework (MBMS) which employs our Ecore-based implementation of our representation, *Crepe*, at its core. The search history, configuration parameters, finitisation information, and resulting solution are all expressed as models in MBMS, and the search algorithms and operations are defined as model management operations. The MMOs have all been written in EOL – a generic model management language. One can, however, view many of the operations as different kinds of model transformation. For example, the genotype-to-phenotype mapping is a model transformation where the source model conforms to the genotype metamodel and the target model conforms to the given solution space metamodel. Furthermore, the adaptation operations used by the search algorithms are model transformations: crossover is a model merge transformation, and mutation is a refactoring transformation. Each of these could have been implemented using the Epsilon Transformation Language (ETL) [94], a language designed specifically for model-to-model transformation. Doing so would have reduced the amount of code written to define these operations, arguably making them more maintainable. Unfortunately, however, Epsilon doesn't provide adequate support to combine separate transformation languages together in an efficient manner. Even though at this stage, we are not interested in fine-tuning the performance of the MBMS framework, the overhead of separating each operation into ETL modules and combining them with other EOL modules would be too great.

One benefit of a model-based metaheuristic search framework is that the search history can be further inspected and even reused. Using MMOs, the search space can be visualised, enabling algorithm developers to validate their implementation, and allowing users to gain insight into the landscape of their solution space. Furthermore, the search model can be reused as a starting point to subsequent searches.

*Crepe* has been implemented separately from MBMS to allow the representation to be used for other, non-metaheuristic search-related, tasks. In chapter 6 we used the representation to analyse uncertainty in models using sensitivity analysis, and in section 8.2 we propose other uses of the representation.

In section 2.3 of the literature review chapter, we described a recent proposal by Kessentini et al. for a framework to encode MDE models for search [84]. *Crepe* and MBMS have a number of advantages over the proposed framework, in particular with respect to usability. One of our goals was to produce a framework



with a low entry barrier for MDE practitioners. Kessentini et al.'s framework requires the user to define a transformation from the metamodel of interest to a generic *encoding metamodel* (which is arguably analogous to our genotype model), or define their own problem-specific encoding metamodel (and the transformation to it). This requires the user to have fundamental knowledge of representations for search. Fitness functions are complex to implement in their framework as the user needs to either define them with respect to the encoding metamodel, or define a second transformation back to the problem's native form. In our framework the user simply needs to define a finitisation model and a fitness function, which can be either in Java or EOL; all transformations are managed automatically and so the user can focus their effort on defining a quality fitness function.

We now discuss some of the limitations with our representation and its implementation.

#### 4.7.1 *Limitations*

There are three main limitations of our representation and its implementation. We address each in turn.

***Minimum scoping*** The finitisation model allows users to specify the restrictions on the data values that can be encoded and also on certain structural aspects of the models that can be encoded. One of the structural finitisations allows a maximum scope to be defined for particular meta-classes or features. This stops the model growing too large, and therefore reduces the size of the search space. We do not, however, currently support *minimum* scoping. These are cases where a particular number of objects conforming to a specific meta-class *must* exist in the model, or where an association must reference at least one object.

There are three ways to address this in the current implementation. Firstly, increase the size of individuals in the search, therefore increasing the probability that meta-classes will be instantiated multiple times. Secondly, the fitness function could heavily punish those solutions who do not meet the minimum scoping rules. Thirdly, one could resolve the issue in the fitness function. Where objects are missing, they could be created by the fitness function following some user-defined systematic procedure. This would actually mean that some individuals 'accidentally' encode more information than expected, but as all candidate solutions would be subjected to this procedure, it should not affect the performance of the search algorithm (though this claim would need to be thoroughly tested).

***Custom datatypes*** Although the finitisation model supports limiting primitive data types such as strings, integers and doubles, it does not currently support custom data types. Metamod-

els may need to define attributes that are not primitively typed or references to objects that are not defined in the metamodel. For example, one might wish to define a reference to a class in a custom Java library. To support this, Ecore allows external Java classes to be captured in the model as *custom data types*, which can then be assigned as meta-class attribute's type. As this enables users to specify any Java class, our representation doesn't support these custom data types and any attributes referencing them are automatically ignored. However, a potential solution to this issue would be to allow the user to provide a *resource pool* of custom data type objects and reference elements in the pool – similar to how Basic OOGP works (see section 4.2.1).

**Model validation** A metamodel is only able to constrain the structure of the model space it represents. In practice more constraints are required in order to properly represent the domain being modelled. Commonly, metamodels are augmented with constraints written in a language such as OCL [61] or EVL [93]. Such constraints include specifying invariants on meta-classes, or defining relations between features. Take for instance, the zoo example from section 2.1 (figures 2.2 and 2.3), we could define two invariants on the Cage meta-class using OCL:

```

1 context Cage inv :
2   self.spaceAvailable >= 3 and self.spaceAvailable <= 30
3
4 context Cage inv :
5   self.animals->collect(a | a.spaceRequired).sum() <= self.
   spaceAvailable

```

**Listing 4.11:** Example OCL constraints for the zoo metamodel from figure 2.2.

The first invariant provides some domain information about the sizes of cages that can be purchase by the zoo. The second invariants states that all animals in every cage need to have the appropriate amount of space. Crepe does not currently consider these extra constraints and so may generate models that would not pass this kind of domain validation. Some constraints can be encapsulated in the finitisation model. For instance, the first constraint above can be defined as a feature range finitisation. We cannot, however, capture more complex constraints, such as the second constraint above, in the finitisation model. To address this, we considered incorporating validation into the genotype-to-phenotype transformation. EMF allows metamodel developers to create a *model validator*<sup>5</sup>, which could be used by the transformation to ensure it doesn't take an invalid action. However, including validation in the transformation is not a trivial process. For instance, we would need to determine the point at which we validate the model. Do we validate once the entire mapping process has completed? What do we then do with models that failed validation? Do we throw the model away, somehow punish the

<sup>5</sup> Custom validators are created using the EMF Validation Framework: <http://www.eclipse.org/modeling/emf/?project=validation>

fitness of that model, or even attempt to *fix* the models so that it passes validation? Rather than validating once the model has been produced, do we instead validate every time we set a feature or create an object? This may inhibit any interesting models from being created as it is likely that the process would need to go through a few “invalid” assignments to produce an instance that is valid. The complex interconnections between model objects make this approach impractical. Meyer’s book, *Object Oriented Software Construction* [109, chapters 11.8 and 11.9], discusses this non-trivial issue in detail for class invariants of object-oriented programs.

Due to these problems, we leave this as future work and propose that these extra constraints be captured in the fitness function in order to provide more accurate guidance for the search algorithm. This also side-steps the issue of tying the implementation to a specific validation framework – it is unreasonable to ask users to implement custom EMF validators if they’ve already defined their constraints in OCL or EVL.



# *III*

## *Applying Search to Models*



# Searching for Optimal Models

# 5

THE LITERATURE SURVEYED in section 2.3 uncovered a number of existing approaches that attempt to discover models using SBSE techniques. The problems addressed covered test model selection [145, 46, 21], discovering behaviour models [59, 58], finding optimal system models [138, 136, 105], and analysing the trade offs between competing requirements [18]. Furthermore, there were a selection of papers that attempted to optimise existing models [30, 141, 156, 158].

There is much scope for applying SBSE techniques to search for optimal models. The challenge lies in defining an adequate fitness function to tailor MBMS to the problem. In this chapter, we use Crepe and MBMS to address the video game problem proposed in section 3.2, discussing how Crepe compares to the prototype representation described in chapter 3 and [186]. Furthermore, we utilise Crepe and MBMS to extract a model of runtime system behaviour from a corpus of system logs. This behaviour model can then enable other systems to adapt their own behaviour in order to optimally fulfil their goals.

**Chapter Contributions** The contributions of this chapter are summarised below.

- An application of Crepe and MBMS to discover optimal models in the context of a video game.
- A model- and search-based approach to self-adaptation at the component level. In particular, we perform a principled analysis of the efficacy of using the representation to extract a model of system behaviour using different strategies.

**Chapter Structure** Section 5.1 uses Crepe and MBMS to discover optimal opponents in the Super Awesome Fighter video game. We show that a genetic algorithm outperforms random search, demonstrating that the problem isn't trivial and that our approach is effective. Section 5.2 defines a model- and search-based approach to applying component-level self-adaptation. We investigate the feasibility of, and different tactics for, using meta-heuristic search to extract a model of a SAF character, based on the log files produced from a set of fights. The model extraction

## Contents

5.1 FDL Search: Crepe . . . . .	102
5.2 Runtime Adaptation . . . . .	109
5.3 Summary . . . . .	133

task proves particularly challenging, and we perform an investigation into whether tuning the parameters of MBMS has any effect. We find that the results improve only slightly and posit why this might be the case: the difficulty of the problem. We show that a genetic algorithm outperforms random search, and investigate three factors that affect the results.

## 5.1 *FDL Search: Crepe*

To provide a comparison between the original Grammatical Evolution-based implementation and our new generic representation of models, we apply Crepe and MBMS to the SAF problem set out in section 3.2. In the GE implementation, the phenotype was a string conforming to the Fighter Description Language. As previously discussed, although textual, FDL has a metamodel and it is this metamodel to which we can apply our search framework.

To remind the reader of the experimental questions for SAF that we set out in section 3.2, they are listed below:

**EX1** Is it possible to specify unbeatable fighters? If a fighter can be unbeatable, it may be necessary to either amend the game play or restrict the Fighter Description Language to limit the possibility of a human player specifying such a fighter.

**EX2** Is it possible to derive a fighter that wins 80% of its fights against a range of other fighters? Such a fighter could be used as the pre-defined non-player opponent since it would provide an interesting, but not impossible, challenge for human players. The figure of 80% is an arbitrary choice that we believe represents a reasonably challenging opponent for human players. It acts as an illustration of discovering a fighter of a specified quality.

Our aim in repeating these experiments is to, firstly, determine that the search framework that we have built on top of our MDE-model representation is effective, and secondly to highlight the strengths of weaknesses between the original GE implementation and our new implementation. A direct comparison of the algorithms is not possible as they are distinct implementations of distinct algorithms, but we hope to draw some interesting insights into both. As with the original GE experiments, we compare a GA against random search in the expectation that the GA outperforms the random search algorithm and thus illustrating the evolvability of the representation.

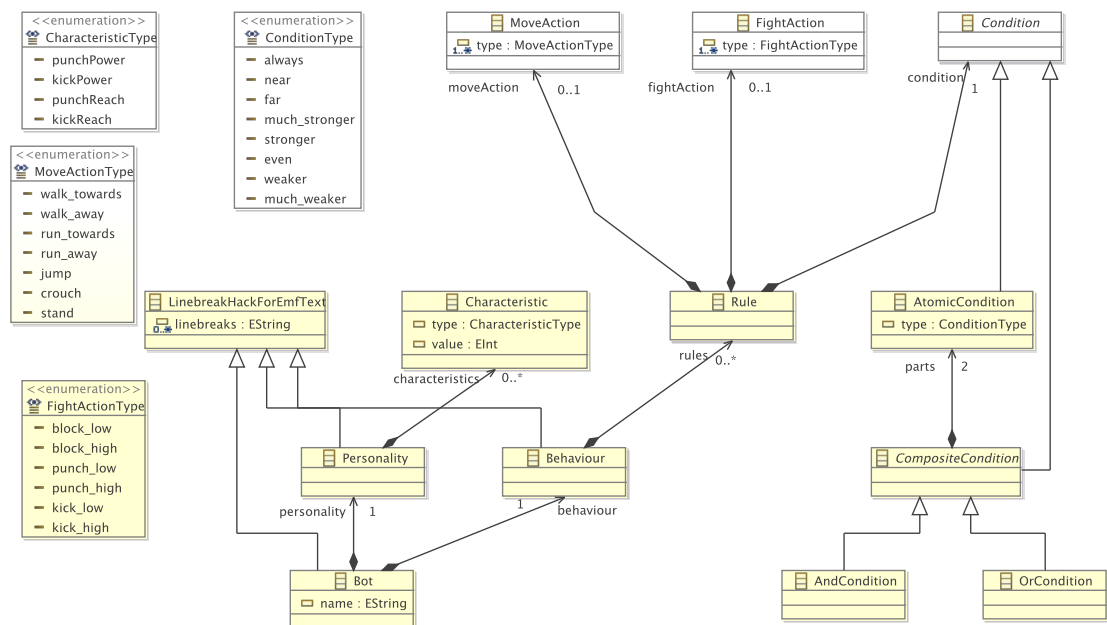
This section shows the process of and the results of using our search framework to address the SAF search goals. Section 5.1.1 shows how the current metamodel of FDL creates a vastly complex solution space, causing our framework to struggle to find valid solutions. Section 5.1.2 then shows how we can develop



a simplified, but equivalent, metamodel for FDL that is more amenable to search. Section 5.1.3 discusses the results of these experiments and finally section 5.1.4 compares them with the results from the GE experiments.

### 5.1.1 Searching FDL

The metamodel produced by EmfText for the FDL language is shown in figure 5.1. You can see that a Bot contains a reference to a Personality object and a Behaviour object, each of which have references to sets of Characteristics and Rules, respectively. Rule Conditions are expressed using subclassing to distinguish atomic conditions from the two composite conditions. Choice of action is captured using arrays of action types (MoveActionType and FightActionType).



**Figure 5.1:** The Ecore meta-model for the Fighter Description Language; repeat of figure 3.1.

The objective of this evaluation is to provide a comparison between the GE representation of models and our generic representation of models. Therefore in order to apply our search framework to the FDL metamodel and thereby repeat our earlier SAF experiments, we need to define a finitisation model. The root class is the Bot class, and the values of Characteristics needs to be finitised to the range 1..10. Furthermore, we need to define a maximum scope of 1 for the Personality and Behaviour classes. This is because Bot objects can only reference one of each, and if we allow the framework to produce many, we have to delete all but one. This may produce a large amount of genotypic redundancy as any Characteristics or Rules assigned to the deleted objects will also be deleted.

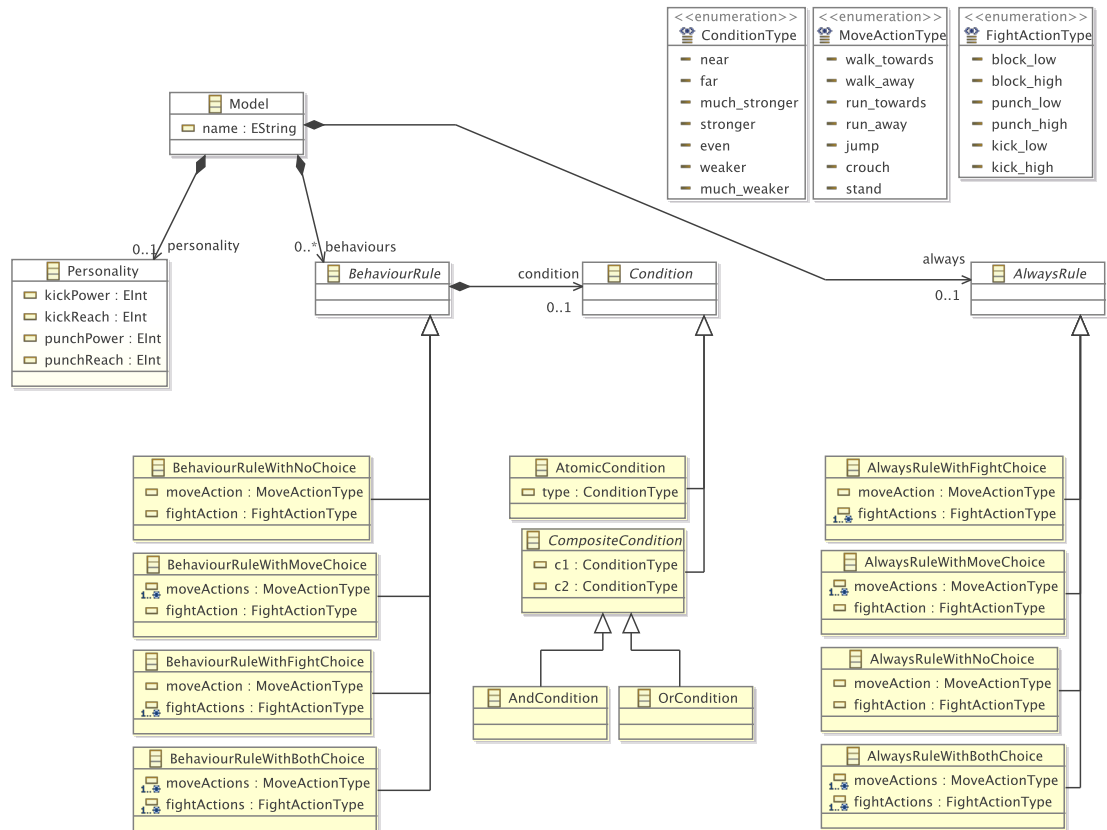
When attempting to apply search over the FDL metamodel,

it quickly became apparent that the metamodel is not conducive to our representation. The number of indirect references make it very difficult to produce models that are valid with respect to constraints that the metamodel does not express. For example, to create a rule with an ‘and’ condition, we need to produce an `AndCondition` object, which references two `AtomicCondition` objects and is referenced by a particular `Rule` object. One complete rule requires up to six objects to be created and have their references correctly assigned. This means that for every rule, six segments in an individual need to be configured correctly. This proved a challenge for the search framework, as it struggled to produce valid rules – even trying cases where individuals consisted of 200 segments were unable to easily produce valid rules, and the time taken to produce such large models made it impractical to perform a complete search. Instead, we decided to define a simplified, yet equivalent, metamodel that is more amenable to the representation and search framework, and also include some domain validation in the fitness function. We also defined a model transformation from this metamodel to the original.

### 5.1.2 *Remodelling SAF for Search*

Figure 5.2 shows our ‘simplified’ metamodel for FDL. Although containing more classes than the original metamodel, it has been defined so as to minimise redundancy and maximise the chances of producing valid models. Firstly, the `Personality` explicitly captures the characteristics’ values. Secondly, `BehaviourRules` are now contained directly by the root class, `Model`. This removes the need for any class scoping. Conditions are very similar to the original, with a small change in the way composite conditions are expressed. Separating the two conditions into their own features means that the search algorithm can assign values to them directly; other abstractions (such as an array) would lead to more complicated code. Another constraint that is not captured in the vanilla FDL metamodel is the requirement that there is a rule with the `always` condition (see section 3.3.4). We enforce this by defining a special `AlwaysRule` class that is contained directly by the `Model`. Both `BehaviourRule` and `AlwaysRule` are abstract and contain four concrete implementations, each of which has a different configuration of action choice. We did this because whilst attempting to generate vanilla FDL models, we found that, due to the large number of feature pairs and actions being represented as an array, the mapping rarely produced rules which didn’t contain choices. Separating action choice the way we have in figure 5.2 gives all four combinations equal chance of occurring in the generated rule set.

We found in initial tests that individuals with between 30 and 50 segments were adequate to produce valid and interesting fighters without impacting performance of the search algorithm.



**Figure 5.2:** A search-amenable Ecore metamodel for the Fighter Description Language.

It may seem unreasonable for a user of our framework to produce a new metamodel for their problem, however we speculate that we could automatically produce the search-amenable metamodel by detecting “bad smells” in the metamodels and defining patterns for how to deal with them. We leave this as future work. Furthermore, it is not uncommon to resorting to solving abstractions when the cost or complexity of solving a problem is high.

### 5.1.3 Implementation and Results

The finitisation model for this metamodel is very straightforward. We simply restrict each of the features in the Personality class to be between one and ten. Although we have defined this metamodel to reduce the number of invalid models that the search framework encounters, it is still possible to produce invalid models. To account for this, our fitness function provides a destructive validation process: if a rule is not valid, it is deleted from the model. If the resulting model has no rules, then that model is assigned the worst possible fitness. The rules are deleted from the phenotype only: the associated genotype segments become redundant, and may be reactivated in a future generation via mutation.

```

1 operation doEvaluateFitness(candidate:MM!EObject) : Real {
2   if (not candidate.validate()) {
3     return MAX_FITNESS;
4   }
5
6   var victories = fightTool.fight(candidate.toFdIString(),
7     numFightsPerOpponent, gameLimit);
8
9   return ((targetPercentage * numOpponents *
    numFightsPerOpponent).floor() - victories).abs();
}

```

**Listing 5.1:** *The fitness function used to evaluate candidate fighters.*

Listing 5.1 shows an excerpt from the fitness function. In order to analyse the fitness of a candidate solution, we fight it against the same human-defined panel of opponents as in the GE experiments. To do this, we defined an Epsilon tool (`fightTool` in listing 5.1) which takes the candidate fighter, fights it against the panel, and returns the number of victorious fights. To avoid performing a model-to-model transformation to the vanilla FDL metamodel, we defined a simple model-to-text transformation that produces a valid FDL string. The custom Epsilon tool then uses the FDL parser to read the string and use it with SAF. The fitness of each candidate solution is then calculated as in equation 3.1.

Parameter	Setting
GENOTYPE TO PHENOTYPE MAPPING	
Number of segments	30..50
Number of feature pairs	10..20
GENETIC ALGORITHM	
Maximum generations	50
Population size	20
Selection method (for reproduction)	Tournament, size 2
Number of elite individuals	2
Reproduction method	Single point crossover
Mutation method	Integer mutation (random value)
Mutation probability	0.1 (uniform)
Segment destruction probability	0.15
Segment creation probability	0.15
FITNESS METRIC	
Number of opponents ( $n_{\text{opps}}$ )	7
Number of fights ( $n_{\text{fights}}$ )	5

**Table 5.1:** *Parameter settings for the genetic algorithm, genotype-to-phenotype mapping, and fitness metric.*

In order to provide a fair comparison with the GE implementation, we kept as many algorithm settings as possible the same – shown in table 5.1. We used uniform mutation, and selected arbitrary values for the creation and destruction operator probabilities. As with the original implementation, no substantial effort was made to tune the parameter settings. Once again, random search was implemented by assigning the mutation probability to

1.0 and keeping all other parameters the same.

**Response** As with the GE case study, we performed four experiments: one for each combination of question (EX<sub>1</sub> or EX<sub>2</sub>) and algorithm (GA or random search). Each experiment was repeated 30 times, each with a different seed to the pseudo-random number generator.

**Results and Analysis** The results of the four experiments are shown in table 5.2. As you can see, the GA met its goal 100% of the time, in both experiments. Furthermore, random search was able to find an optimal solution in all 30 repetitions of EX<sub>1</sub>, but struggled with EX<sub>2</sub> – only achieving a 67% success rate. For EX<sub>2</sub>, the GA found the optimal solution in an average of 10.9 generations. For EX<sub>1</sub>, both the GA and random search always found an optimal solution, therefore we need to compare them further to ensure that the GA is effective and that the problem isn't so simple that random search finds optimal solutions easily. The GA took on average 5.4 generations to converge on an optimal solution for EX<sub>1</sub>, whereas random search averaged 13.2 generations. The results of an unpaired *t*-test showed that this result is statistically significant ( $p = 0.003$ ), meaning that the GA was able to navigate the search space effectively. As illustrated in figure 5.3, the GA found the optimal solution in less than ten generations 80% of the time, and always in less than 15 generations. Random search managed to find an optimal solution in ten generations 60% of the time, 70% in less than 15 generations, and 100% in at most 43 generations.

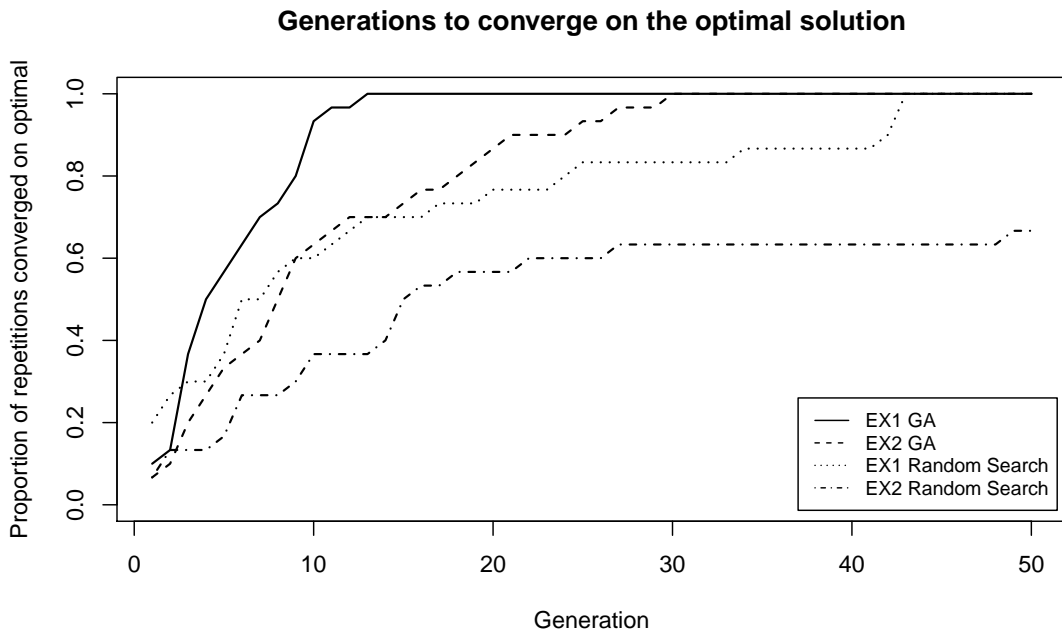
Experimental Question	Genetic algorithm		Random search	
	% successful	Avg gens	% successful	Avg gens
EX <sub>1</sub> ( $\rho = 1.0$ )	100	5.4	100	13.2
EX <sub>2</sub> ( $\rho = 0.8$ )	100	10.9	67	-

What is surprising with these results (especially when compared to the original GE results) is how well random search performs. This, combined with the efficiency of the GA, suggests that the SAF problem isn't too difficult to solve by our representation and search framework. However, as the results show, random search takes on average 2.4 times longer to find an optimal solution than the GA. One question to ask, therefore, is whether these results extend to more complex problems: if this relationship holds then the more difficult the problem, the more impractical random search becomes.

**Table 5.2:** The results for the four experiments. Where an algorithm was 100% successful at finding an optimal solution, we also list the average number of generations that it took to converge.

#### 5.1.4 Comparison of GE versus Crepe

Previous works that have looked at comparing metaheuristics on the same problem, e.g. [146, 101, 26, 6], aim to keep the com-



**Figure 5.3:** Illustration of how long it took each repetition to converge on an optimal solution. The Y-axis shows the proportion of repetitions that have converged.

parison fair by using the same representation (and therefore the same search landscape) throughout. This is something that we have been unable to do, due to the distinctness of the techniques. Moreover, Rothlauf argues that the choice of representation is vital for the performance of a metaheuristic algorithm [147]. Therefore, one might argue that it is unfair to compare algorithms on the same representation as it may be biased towards certain algorithms. As such, it is difficult to fairly compare the performance of Crepe against the GE-based prototype implemented, however we can address other aspects.

As previously mentioned in section 3.4, research has shown that GE has low locality and is outperformed by GP on many benchmark problems. The low locality of GE is one possible reason for Crepe outperforming it on the SAF example. For instance, for EX1 GE was only able to find an optimal solution approximately 67% of the time, compared to Crepe’s 100%. However, the performance of random search suggests that the model-based representation used in Crepe produces a search landscape with many peaks.

We now address the disadvantages of GE that we described in section 3.4. GE is suited to textual languages (as it was designed for), the metamodels of which Crepe struggles with and a refactoring was required. Therefore, although Crepe *can* be applied to textual DSLs, it may require the metamodel to be refactored into a more amenable form. Crepe does, however, support cross-referencing (both intra-model and inter-model) and has the ability

to specify data in a way that is decoupled from the metamodel.

### 5.1.5 Summary

This section has applied Crepe and MBMS to tackling the SAF problem defined in section 3.2. We have discovered that the ability of Crepe to effectively solve a problem is closely related to the metamodel being used to define the solution space. As a consequence of this, when using Crepe, one may be required to refactor their metamodel. This refactoring follows a specific set of rules: flattening the meta-class hierarchy and reducing the number of references. As such, we posit that it may be possible to automatically optimise metamodels for use with Crepe. This will require Crepe to be applied to many problems, and so we leave this as future work.

## 5.2 Runtime Adaptation

Software that can adapt to changes in its environment without, or with minimal, human intervention is a key challenge in current software development [24]. One method of deciding upon the most appropriate adaptation to perform is to utilise *metaheuristic optimisation techniques*. These techniques may be used to efficiently locate (near) optimal solutions by assessing how close candidate solutions are to solving a problem and using this information to guide the search over the the space of all possible solutions [65, 62]. Therefore, in reformulating system adaptation as an optimisation problem, these techniques can be applied.

The previous section demonstrated that it is possible to select interesting opponents for a given human-defined player. In essence, the system (the SAF game) is responding to its input (the human player(s)): for different human players, different opponents will be selected. The player then attempts to improve their fighter at the end of each game in order to win more fights. In a directly interactive game, where the fighter's behaviour is defined dynamically using, for example, a joystick, one can imagine a case where the game attempts to learn patterns in the player's behaviour in order to select suitable opponents. An opponent's behaviour could be even updated during a fight as the game learns more about the human. This kind of system response is analogous to system adaptation at runtime. France et al. [49] identify the utilisation of models at runtime as one of the important research areas for the coming years. In section 2.3.2 we presented existing work that has addressed the challenge of optimising models at runtime. For instance, Ramirez and Cheng [136] use metaheuristics to discover optimal system configurations at runtime. These configurations are represented as a graph of interconnected system components. A genetic algorithm is used to

activate and deactivate links between components in the hunt for an optimal configuration, using sensory information from the environment to evaluate candidate configurations. Earlier work by Goldsby and Cheng [59] uses metaheuristics to discover optimal component behaviour models (represented as state machines). Performance issues, related to an expensive fitness function (see section 2.3.1) and the time taken to generate models, were cited for this technique as only being effective at design time [136].

The focus in this section is adaptation at the component level, providing a fine-grained level of adaptation whilst aiming to overcome the performance issues encountered by Goldsby and Cheng [59]. Rather than encoding static information from the environment in fitness functions (as in [136]), our approach uses metaheuristics to extract a model of the environment's dynamic *behaviour*. From this environment model, we apply a second metaheuristic to discover a model of the optimal system/component behaviour. To achieve this, we use our generic search-amenable representation of models – meaning that our approach can be applied to any problem domain where adaptable parts of the system, and the environment's behaviour, can be expressed as a model. It is worth noting that, in general, modelling the behaviour of the environment can be very challenging and expensive.

We apply our approach to adaptation in the context of SAF. Our work in the previous section demonstrates the game adapting to the player's behaviour by selecting appropriate opponents. The experiment, however, was unrealistic in the sense that a real human interacting with a game using a joystick or similar, would not follow such a strict set of rules and would in fact behave more inconsistently. In this section, we use a search-based approach to *extract* a model of the human player's behaviour based on a collection of fight log information. The extracted model can then be used to select opponents in the way previously presented.

We describe the background behind self-adaptive systems and using models at runtime, as well as related work, in section 5.2.1. In section 5.2.2, we outline our model- and search-based approach to performing runtime adaptation. Section 5.2.3 tailors the approach to SAF and illustrates the need for a two step process for extracting the model. In section 5.2.4, we perform a detailed, principled experiment that investigates whether Crepe can successfully extract FDL models from fight logs. We compare a genetic algorithm to random search on 18 variations of the problem, altering the complexity of the human, introducing noise into the log files, and applying different *seeding* strategies. Furthermore, we *tune* the parameters of Crepe in order to optimise the performance of each experiment.



### 5.2.1 Background and Related Work

The inherent uncertainty found in software development makes it very challenging to be able to accurately predict how a system should behave in every situation, especially those which could not be foreseen [23, 24, 154]. Manually reconfiguring software at runtime is costly and so many modern systems need to be able to autonomously adapt to dynamic operating contexts [24, 154, 118]. Whereas traditional software has been implemented as an open-loop system, self-adaptive software is a closed-loop system where the environment and the system itself provides the feedback in the loop [154]. Cheng et al. [24], the output of a Dagstuhl seminar, present a research roadmap on software engineering for self-adaptive systems. With respect to the challenges laid out in [118], they claim that:

... software systems must become more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changing operational contexts, environments or system characteristics. [24]

Adding to this, Salehie and Tahvildari [154] state:

The primary reason (for needing self-adaptive software) is the increasing cost of handling the complexity of software systems to achieve their goals. Among these goals, some deal with management complexity, robustness in handling unexpected conditions (e.g., failure), changing priorities and policies governing the goals, and changing conditions (e.g., in the context of mobility).

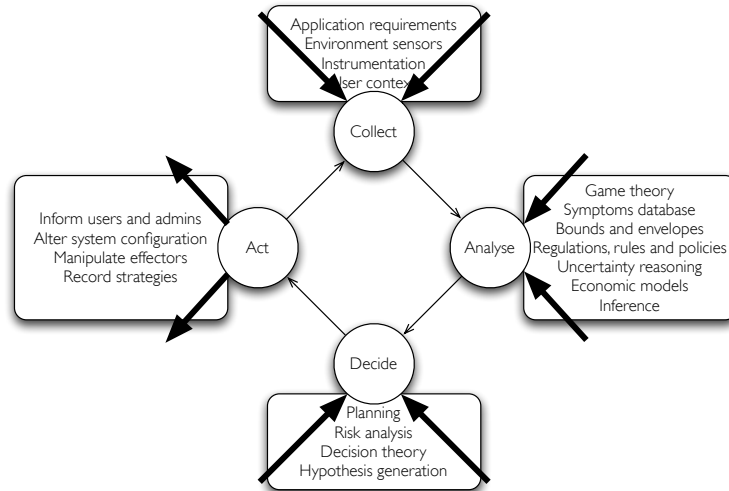
Thus, an important research topic today are systems that can *self-adapt*; the common element that provides this self-adaptation is software [24]. In 2003, IBM released the seminal paper that outlined the adaptivity-related properties that software should have in order to fulfil these goals – these properties are known as *self-properties* [83]. The properties include the abilities of: adjusting configuration properties, optimising the system to increase performance, identifying and fixing issues, and protecting from attack; all of which needs to be performed autonomously.

Figure 5.4 shows the autonomous process that self-adaptive systems commonly follow. Briefly, the system monitors and *collects* information from the environment in which it resides, this information is *analysed* and used to *decide* on the most appropriate response, which the system then *enacts*.

**Runtime Adaptation and Search** Cheng et al. [24] summarise the state-of-the-art in software engineering for self-adaptive systems. One of the research challenges of self-adaptive systems raised in [24] is performing trade-off analysis between potentially conflicting goals. SBSE techniques have been developed for multi-objective problems such as this [62].

There have been a few attempts at using search techniques to aid runtime adaptation. As mentioned, Goldsby and Cheng [59]

**Figure 5.4:** *The control loop of self-adaptive systems.*  
Adapted from [24].



use search to discover resilient behaviour models (state diagrams) of autonomic systems, focusing on aiding the development of systems at design time, as opposed to adapting an existing system at runtime. Ramirez et al. [136] use search to dynamically self-configure systems at runtime. They evolve simple graph-based representations of systems; the work presented in this section is generic to finding system models conforming to any metamodel. Our approach extracts a behaviour model of the environment from which to discover optimal system models whereas [136] uses sensory data to directly influence their system model search.

Search has successfully been applied to the other cases of inferring information from a corpus (i.e. model extraction). Ratcliff et al. [139] use genetic programming to infer code invariants from a corpus of program traces. Wyard [188] demonstrates the use of genetic algorithms to infer context-free grammars from a corpus of strings. Kammeyer and Belew [80] apply a combination of genetic algorithm and local search to a more complicated grammar inference problem: that of *stochastic* context-free grammars.

**Utilising Models at Runtime** Using models at runtime allows application-level (as opposed to code-level) reasoning about the system, and enables design-time requirements to easily be mapped to runtime behaviour [100, 157].

Ferry et al. [45] argue that defining metamodels for key system components enables system behaviour and the adaptation process to be validated at runtime. They propose a process that uses model transformations to adapt composite services based on given rules. Lehmann et al. [100] provide guidance for defining metamodels for runtime models, and present a task-specific modelling language for formalising runtime metamodels. Sanchez et al. [157] define an extensible platform for utilising executable runtime models. The platform uses extensible metamodels which

enables application-agnostic monitoring and adaptation tools to be developed.

Runtime models have been shown to be beneficial for self-adaptive systems, but there has only been a small number of attempts to utilise search to aid runtime adaptation. The next section outlines our approach to runtime adaptation that utilises both models and SBSE.

### 5.2.2 *A Model- and Search-Based Approach for Runtime Adaptation*

In this section, we describe our approach to self-adaptation at runtime which utilises both models and search. The approach is built around three key ideas:

1. *The data being produced by/monitored from the environment is captured in a model.*

The sensory data being emitted from the environment can be challenging to manage or interpret, and require complex supporting tools [157]. For instance, mapping runtime data to business goals can be difficult [157]. Runtime monitoring models address this by capturing the the runtime information in terms of the domain, enabling better runtime decision making [157]. In our approach, this data model is used to aid in the extraction of a behaviour model of the environment.

2. *The environment's behaviour can be modelled.*

Our approach aims to transform the environment data model into a model that describes how the environment is *behaving*. This could be, for example, a set of rules or a finite state machine. By determining a model of the environment's behaviour, we can use metaheuristic search to discover a model of the optimal system/component behaviour in response to the environment.

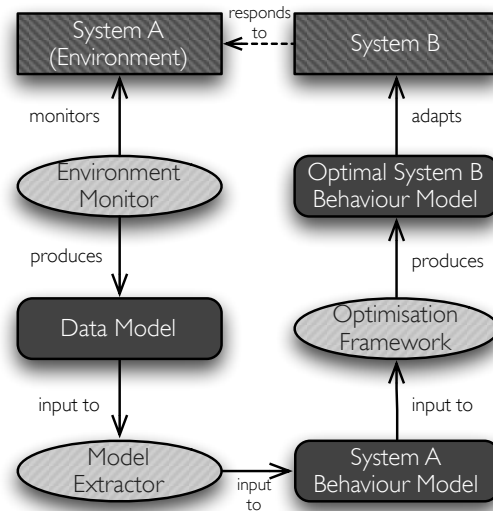
3. *The adaptable part(s) of the system is expressed as a model (or can be updated from a model, e.g. via a model transformation)*

As demonstrated in SAF, where models of fighters are queried in order to play the game, models can be physical components of a system. When designing a system that needs to be able to support self-adaptation at the component level, the designer/developer knows which part(s) of the system will be adapting and can develop metamodels to capture those elements at a higher-level of abstraction [100]. Sanchez et al.'s *Cumbia* platform [157], for instance, utilises executable runtime models for this purpose.

Figure 5.5 shows our runtime adaptation approach that combines the use of models and metaheuristic search. The first step

in the process is to capture the environmental monitoring data in a model, called the *data model*. This model acts as the input to a search algorithm which aims to extract a model that can express the *behaviour* of the environment. This behaviour model then drives an optimisation algorithm that aims to derive the appropriate system response: i.e. an updated version of the model of the adaptable part(s) of the system.

**Figure 5.5:** *The process of extracting a model of the environment and using it to search for optimal system behaviour.*



We now overview the steps taken in our approach. They follow the *collect, analyse, decide, act* control loop of self-adaptive systems (figure 5.4).

*Step 1: Capture a Model of the Environment* Sensory information is the basis for determining the appropriate adaptation. Figure 5.4 lists a number of sources that produce this data. Selecting an appropriate way to collect this data is domain-specific. Using metamodels here can relate this information to parts of the system, or to system requirements. Furthermore, it allows the information to be used with model management operations, such as model transformations, to aid in the decision making process.

*Step 2: Extracting a Model of the Environment's Behaviour* A model can be extracted from a corpus of data in numerous ways, for example using game theory or inference from domain knowledge [24]. In our approach, we examine an extraction process based on metaheuristic optimisation. The process assumes the existence of two metamodels: one that meaningfully captures data coming from the environment (i.e. the data model); and another that represents the environment's behaviour – this could, for example, be a state machine or a domain-specific model. The information contained in the environment data model is used to guide the metaheuristic algorithm to infer the model of environment's behaviour. In order to infer the

behaviour model, we utilise our generic, search-amenable representation of models defined in chapter 4. For model extraction purposes, we search over the space of environment behaviour models (defined by the metamodel) using the environment data model to guide the search.

*Step 3: Discover the Optimal System Response* The environment behaviour model becomes the input to a second genetic algorithm which aims to discover the optimal response that the system should perform.

*Step 4: Update the System* The optimal system model can then be used to update the running application. If the adaptable component is itself a model, then a simple model replacement could be performance. Alternatively, a model transformation (e.g. to generate new configuration files) could be performed, or some other domain-specific action will be taken.

***Near-Optimality*** In terms of implementation, the search algorithms would occur in the background whilst the system maintains its current behaviour. Ferry et al. [45] call this the *transitional state*. However, resources permitting, this optimal behaviour search could occur continuously throughout the system's lifetime. The ever-updating environment data model could adjust the fitness function on a periodic basis.

One of the benefits of using a metaheuristic approach to extracting the environment behaviour model, as opposed to a constructive approach, is that there is always a 'best' solution. If response time is a major factor, metaheuristic search could provide a solution that is 'good-enough' if not optimal. Furthermore, a good-enough solution could be selected and the system adapted, but the GA could continue to execute in an attempt to discover a better solution. The system could then be updated with its optimal behaviour at a later time, however the good-enough model ensured that it performed adequately until then. Each solution can be assigned a *confidence* score – a function of its fitness and the time allotted to produce it.

***Potential Application Areas*** In the subsequent section, we apply the above approach to extracting a model of player behaviour in SAF. Determining, and responding to, a player's behaviour or skill level can obviously be very useful in computer games – balancing game difficulty is a challenging and delicate task [172, 12]. Moreover, the *gamification* of software-related problems is becoming an increasingly popular topic. This is where the concepts used in computer games are applied to other situations in order to encourage user engagement or enjoyment<sup>1</sup>. The leading conference series on the ways humans interact with computers, *Conference on Human Factors in Computing Systems (CHI)*, has hosted a workshop on gamification since 2011, and a recent

<sup>1</sup>[www.gamification.org](http://www.gamification.org)

search on the DBLP Computer Science Bibliography [33] (date 27 August 2013) returned 60 publications (10 journal, 50 conference or workshop) and the ACM Digital Library [47] (date 27 August 2013) returned 276 publications for the keyword ‘gamification’. Our approach may prove useful in these gamification applications.

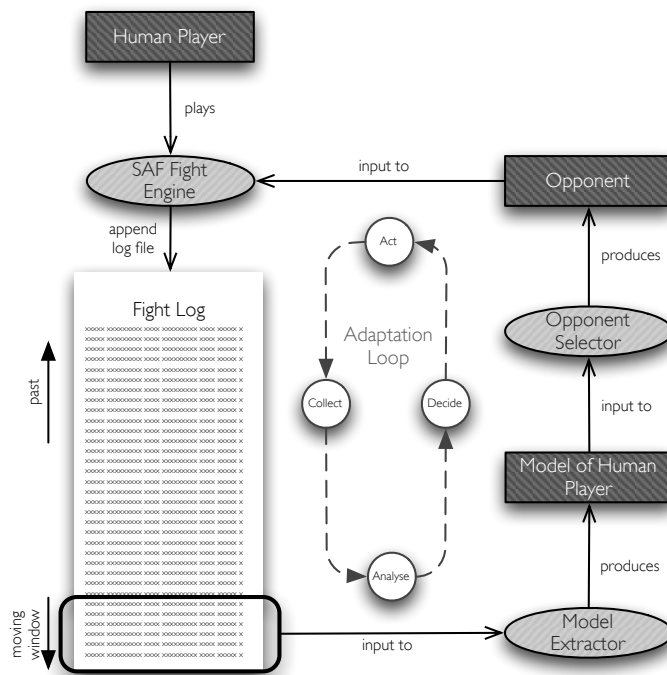
Games and gamification applications are not the only areas where our approach can be applied. Our approach targets situations where the environment exhibits tangible *behaviour*, as opposed to single-state readings. Take, for example, a popular web-based application such as a university library search engine. The university provides a number of web servers that are used to respond to the library application requests. The number of page requests received by the application will vary at different times of the day. By monitoring the requests received over a day, one could extract a model of the website traffic. This could then be used to determine the appropriate number of web servers to deploy in order to maximise throughput and minimise response time. Servers that are not used are a cost to the university, so extracting this information would help them save money. Extracting a model of requests (the environment behaviour model) based on the time of day isn’t particularly challenging and would not require a metaheuristic search algorithm. However, a simple analysis of a single day may miss *patterns* of behaviour. Website traffic will increase not only based on the time of day, but also the day of the week, and even the time of year. It is likely that traffic will increase in the time running up to coursework deadlines or the exam period. Metaheuristics could be used to detect interesting patterns in the website traffic that would enable the library to better managed the traffic whilst cutting costs.

With respect to the case studies used in the literature, there are a number of scenarios in which an environment behaviour model may be beneficial. For instance, Schneider and Becker [160] discuss the need for runtime adaptation in *Ambient Assisted Living* (AAL) systems: used primarily to aid the less able bodied. Different individuals and living environments require different behaviours from the AALs, and these requirements may change over time (e.g. as the user ages) [160]. Extracting a model of the environment (the individual and their living space) could help this adaptation.

### 5.2.3 Case Study: Adaptation in SAF

In section 5.1, we demonstrated how we could select suitable opponents based on a given FDL model of a fighter. In this section we focus on describing the steps needed to extract a FDL model from a collection of fight traces. Once this model has been extracted it can be input to the metaheuristic technique presented in section 5.1.

In SAF's current form, player's specify their behaviour using a FDL model, which may not accurately reflect human behaviour in a more interactive, reactive game. To address this, when generating the fight logs, our experiments (section 5.2.4) create a number of mutants of the human model as a method of introducing non-uniform behaviour into the data to better simulate the behaviour of a human player. Having the original FDL model of the actual human, however, does allow us to validate the extracted model by comparing it against the human model.



**Figure 5.6:** The proposed adaptation loop for automatically selecting opponents in SAF based on the logs of a player's behaviour.

Figure 5.6 illustrates how adaptation could occur in a game similar to SAF. The most recent behavioural information is used to extract an up-to-date model of the player's behaviour. This model is then fed in to our metaheuristic-based opponent selector (the search algorithm described in section 5.1 and [186]) and a (set of) suitable opponent(s) is selected. Adaptation could take place between fights, or even during a fight (where the opponent's behaviour is updated in place).

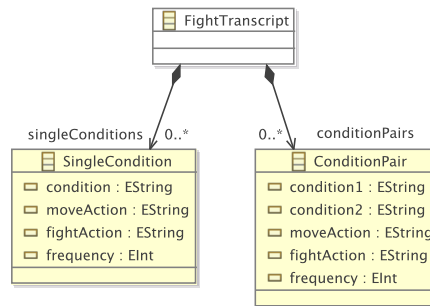
We now describe how we can apply our adaptation approach to SAF by exploring how the key aspects of the approach apply to SAF. In the remainder of this section, we make the assumption that SAF is now a directly interactive game, controlled using a joystick, but opponent behaviour is still specified in FDL. Section 5.2.4, which performs a principled experiment around applying our adaptation approach to SAF, simulates human interaction by mutating the original human FDL model and selecting mutants at random for the fights used to produce the trace log.

**The Environment** SAF produces a log file for each fight that takes place, formatted as follows:

$$P1_X, P1_H, P1_M, P1_F; P2_X, P2_H, P2_M, P2_F$$

where  $PN$  refers to either player one or player two;  $X$  is the player's position;  $H$  is the player's health; and  $M$  and  $F$  are the player's movement action and fight action that they are performing. Some actions in SAF take more than one time step. In these cases, the game logs the player as being *blocked* and stops the player from performing another action. The contents of the log files are the behaviour trace being produced by the environment.

**Figure 5.7:** The data meta-model defined to meaningfully capture the information contained in SAF log files.



As shown in figure 5.6, we extract the data model from the log being produced by SAF. We have defined a data metamodel (figure 5.7) to capture the information contained in the logs in a more useful way – frequency counts of conditions and actions. In every game state, there are always two conditions that are applicable. This is due to the fact that conditions come in two distinct categories: those related to health (e.g. *much\_weaker*, *stronger*) and those related to distance from the opponent (*far*, *near*). Exactly one condition from each category will be applicable in each game state. Therefore, our data model captures the log file information in two ways. Firstly, it captures the frequencies of actions against single conditions, and secondly, it captures the frequencies of actions against pairs of conditions. The two conditions applicable in each game state are calculated using the same rules that SAF uses. For example, the condition *near* is applicable when  $|P1_X - P2_X| \leq 5$ .

Logging two conditions in every state makes manually inferring the player's behaviour rules very tricky. When defining behaviour rules in FDL, the player is not required to specify two conditions and so a rule whose only condition is *near* may be executed in any of the health-related conditions. Therefore, although manual inference of the behaviour rules initially seemed trivial, further inspection suggested that it is not and therefore might be a good candidate for metaheuristic optimisation.

**Partial Model Extraction** The logs produced by SAF only give information about the behaviour rules of the fighter, and



not its *personality*. This information cannot be extracted from the data model<sup>2</sup> and so we propose a two stage extraction process. Firstly, we extract the behaviour rules from the data model using a *genetic algorithm* (GA), and then we use a *hill climbing* algorithm to discover the personality parameters and thus complete the environment behaviour model. We use a GA to discover the rules because the behaviour rules are complex and interrelated, meaning that there are likely to be many local optima which makes the problem unsuitable for hill climbing. To discover the personality parameters, however, the search space is much smaller and contains fewer local optima and so hill climbing is used.

To evaluate the fitness of a candidate set of behaviour rules (a solution in the GA), we contrast them against the information contained in the data model. Each condition pair entry in the data model is passed to the candidate solution as if it were a game state in which the candidate solution was participating. The first rule in the candidate solution that is applicable with respect to the pair of conditions is selected and ‘executed’ the same number of times that the condition pairs appear in the data model. The rule is ‘executed’ multiple times because it may contain action choices and therefore result in different actions. The frequencies of the resulting move and fight actions are compared against the frequencies found in the data model. The fitness ( $f_1$ ) is calculated as the sum of squares of the distance between the target frequency ( $h_{\text{target}}$ ) and the actual frequency ( $h_{\text{actual}}$ ):

$$f_1 = \sum_{i=1}^{n_{\text{cps}}} |h_{\text{target}}^i - h_{\text{actual}}^i|^2 \quad (5.1)$$

where  $n_{\text{cps}}$  is the total number of condition pairs in the data model. We also reward partial matches (e.g. where the move action matches, but the fight action does not).

**Population Seeding** For many metaheuristic optimisation algorithms, the effectiveness of the algorithm can depend on where in the solution space the algorithm commences. For population-based metaheuristic algorithms, such as GAs, it is common to create an initial population of diverse candidate solutions in order to promote exploration of the search space.

We can, however, make use of the information contained in the data model and attempt to create an initial population which starts the search in an area of the solution space that is closer to the optimal solution. This process is called *seeding*. The goal here is not to perform complex inference from the data model: the seeding process should be cheap. In order to seed the initial population, we use the data model to produce a set of FDL models. These models are then transformed to the genotype using the phenotype-to-genotype transformation described in section 4.3.1 and added to the initial generation in the search model. We have implemented two different types of seeding which we analyse in

<sup>2</sup> Thorough analysis of the log file can actually shed *some* light on the personality. For instance, by tracking the distance moved when running, or analysing the effects of punches that make contact with the opponent. This, however, is out of scope for this paper.

section 5.2.4. Although seeding can be useful, it may bias the search towards sub-optimal solutions. As such, when seeding we also include some other solutions created randomly.

*Random Seeding* Our random seeding algorithm creates a set of simple candidate solutions (fighters) by randomly selecting entries from the data model and creating behaviour rules. Up to 4 composite condition rules are created from randomly selected condition-pair entries in the data model, and up to 3 single condition rules are created by randomly selecting single condition entries from the data model. Once the set of rules has been created, the fighter is mapped down to its genotypic form and added to the initial population.

*'Intelligent' Seeding* Instead of randomly selecting entries from the data model, we can do a *small* amount of inference using domain knowledge to create the members of the initial population. For example if we have two condition-pair entries which have different action sets, we can create a composite condition rule which uses FDL's choice construct to select between the actions. The goal is to *quickly* create potentially high quality individuals that we can further improve using search. We do not want to write a complex algorithm that aims to extract perfect information from the data model, as this would be costly.

*Model Completion* As previously mentioned, it may not always be possible to infer the entire environment behaviour model, as is the case in SAF. In this instance, we propose using a different metaheuristic optimisation algorithm called hill climbing. This algorithm attempts to find the correct allocation of personality parameters to the partial model that resulted from the GA. As SAF is aware of the previous opponents that the player has fought, we can use them to evaluate candidate personality parameters. The fittest discovered partial model is, therefore, updated with the candidate personality and then pit against the same opponents that produced the data in the original environment data model. The fitness ( $f_2$ ) of the personality, and therefore the entire candidate solution, is calculated as the distance between the resulting data model and the original data model:

$$f_2 = \sum_{k \in \{dk^o \cup dk^c\}} |g(dk^o, k) - g(dk^c, k)| \quad (5.2)$$

where  $dk^o$  and  $dk^c$  are the condition-action tuples (both condition pair and single condition) in the original data model and the candidate data model, respectively.  $g(d, k)$  returns the frequency for the given tuple  $k$  in the data model  $d$ . If the tuple does not appear in the data model, zero is returned.

*Implementation* We have implemented the model extraction technique(s) as a series of model management operations, illus-

trated (with respect to the experimentation) in figure 5.8. As with the experiments in section 5.1.2, we apply our search framework to the simplified version of the FDL metamodel. However, we have also defined a two-way model-to-model transformation from our simplified metamodel to the FDL metamodel. During the evaluation of a candidate solution, it is transformed into FDL for use in SAF. The process builds on top of MBMS, adding extra MMOs to perform domain-specific tasks. The MMOs and MBMS are chained together using Epsilon’s orchestration framework [97, chapter 12].

We illustrate our adaptation approach by investigating whether or not it is able to successfully extract FDL models from fight logs produced by human models of increasing complexity. It is assumed that once this model has been extracted that we can apply the work from section 5.1 to select the suitable opponent(s), and so we do not discuss this here.

#### 5.2.4 Case Study

We evaluate our model extraction technique by attempting to successfully extract the models of three input fighters. In particular, we wish to understand the effectiveness of our model extraction approach and analyse the effects of population seeding on the quality of the extracted model, whilst also demonstrating the efficacy of Crepe and MBMS.

The three fighters investigated are of increasing complexity: the *simple* fighter (listing 5.2) uses only atomic conditions and no action choice; the *medium* fighter (listing 5.3) has some rules with composite conditions but still no action choice; and the *complex* fighter (listing 5.4) has rules with both composite conditions and action choices.

```

1 simple {
2   kickReach = 5
3   kickPower = 5
4   punchPower = 9
5   punchReach = 4
6   far [ walk_towards punch_high ]
7   much_weaker [ stand kick_low ]
8   always [ jump kick_low ]
9 }

```

**Listing 5.2:** *The FDL for the simple human model*

Each human fighter is pit against a set of random opponents to produce a collection of log files. These log files act as the sensory information coming from the environment. Our goal is to firstly transform this information into the data model and then use that to extract a behaviour model of the environment – i.e. the goal is to extract the original input fighter.

To make the problem more challenging and simulate the non-uniform behaviour of a real human playing a game similar to SAF, we automatically create new instances of each fighter with

slight variations, called *mutants*. When creating the data model, one of the mutants is selected randomly to participate in the fight. This produces a noisier data model, which is intended to simulate more realistic behaviour.

```

1 medium {
2   kickReach = 10
3   kickPower = 10
4   punchPower = 10
5   punchReach = 10
6   far [ walk_towards block_high ]
7   near and stronger [ walk_towards kick_high ]
8   near and much_weaker [ jump kick_high ]
9   always [ jump kick_low ]
10 }

```

**Listing 5.3:** The FDL for the medium human model

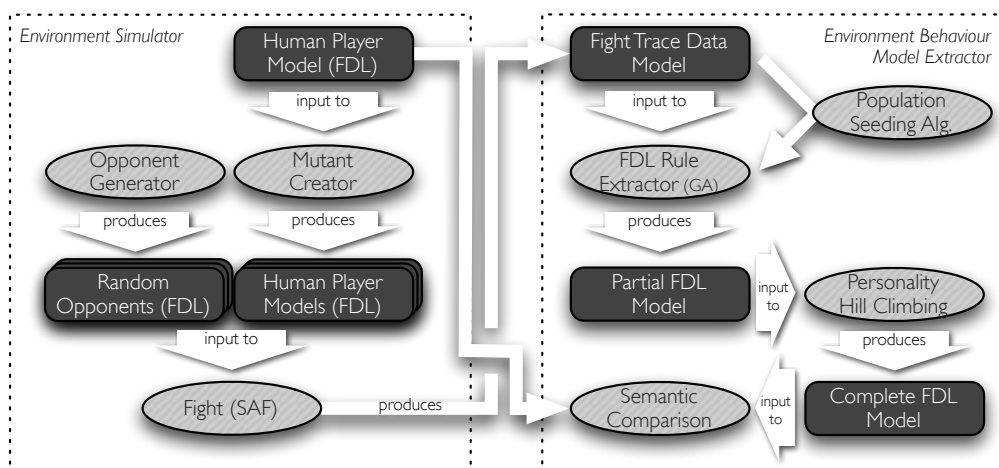
```

1 complex {
2   kickReach = 8
3   kickPower = 3
4   punchPower = 6
5   punchReach = 3
6   far [ choose(walk_towards stand) block_high ]
7   near and stronger [ walk_towards punch_low ]
8   near and much_stronger [ stand punch_high ]
9   near [ choose(walk_away run_away) kick_low ]
10  always [ run_towards punch_high ]
11 }

```

**Listing 5.4:** The FDL for the complex human model

Once the data model has been produced from the logs, it is input into the GA-based (MBMS) partial model extractor. The initial population of the GA is optionally seeded based on the experiment being performed. After the partial model has been extracted, we perform a hill climb on the personality (as described in the previous section) which results in a complete model.



**Figure 5.8:** The process of extracting a model of the player used for the experimentation.

**Algorithm Settings and Implementation** Following the process outlined in figure 5.8, we perform 18 sub-experiments, evaluating six scenarios for each of the three human models. We analyse the effects of the human complexity, the mutants, and of

the population seeding on accurately extracting the model of the human. Table 5.3 shows the experimental design.

Identifier	Human	Mutants	Seeding Type
S01	simple	no	none
S02	simple	no	random
S03	simple	no	intelligent
S04	simple	yes	none
S05	simple	yes	random
S06	simple	yes	intelligent
S07	medium	no	none
S08	medium	no	random
S09	medium	no	intelligent
S10	medium	yes	none
S11	medium	yes	random
S12	medium	yes	intelligent
S13	complex	no	none
S14	complex	no	random
S15	complex	no	intelligent
S16	complex	yes	none
S17	complex	yes	random
S18	complex	yes	intelligent

**Table 5.3:** *The 18 sub-experiments we perform to evaluate three factors that affect adaptation in SAF.*

For fairness, each sub-experiment,  $S_i$ , is executed thirty times, each with a different seed to the pseudo-random number generator. The environment is simulated by fighting the mutants against 50 random opponents. The resulting trace data can be viewed as one instance of the moving window from figure 5.5.

Table 5.4 shows the parameters used in all parts of the workflow. The parameter values selected for a metaheuristic technique plays a crucial role in its efficacy. At this stage, we are not concerned with efficiency (although this is obviously extremely important for adaptation at runtime) and so no substantial effort was made to tune the parameters to this problem. Our initial goal is determine the feasibility of using this approach to adapt components at runtime, analyse whether population seeding is a potential method of improving the performance, and understand the effects of human complexity on the problem difficulty.

**Response** In order to validate whether or not we successfully extracted the human model, we devised a *semantic fighter comparator*. A structural comparison of the fighters would not be enough because different combinations of rules and personalities can result in equivalent behaviour (as shown previously in figure 3.9). Our semantic comparator, therefore, aims to see if the the extracted model is semantically equivalent to the input model. That is, when fighting against the same opponents, do they perform equally as well. In order to calculate a semantic similarity score, the extracted model and the input model both fight against

Parameter	Setting
GENOTYPE TO PHENOTYPE MAPPING	
Number of segments	30
Number of feature pairs	10..30
GENETIC ALGORITHM (BEHAVIOUR RULE SEARCH)	
Maximum generations	20
Population size	10
Seed size (where used)	6 (plus 4 random)
Selection method (for reproduction)	Tournament (size 6)
Number of elites	2
Reproduction method	Single point crossover
Mutation method	Integer mutation (random value)
Class bit mutation probability	0.15
Feature selector bit mutation probability	0.15
Feature value bit mutation probability	0.0.15
Segment destruction probability	0.15
Segment creation probability	0.15
HILL CLIMBING ALGORITHM (PERSONALITY SEARCH)	
Maximum generations	50

**Table 5.4:** *Parameter settings for the workflow.*

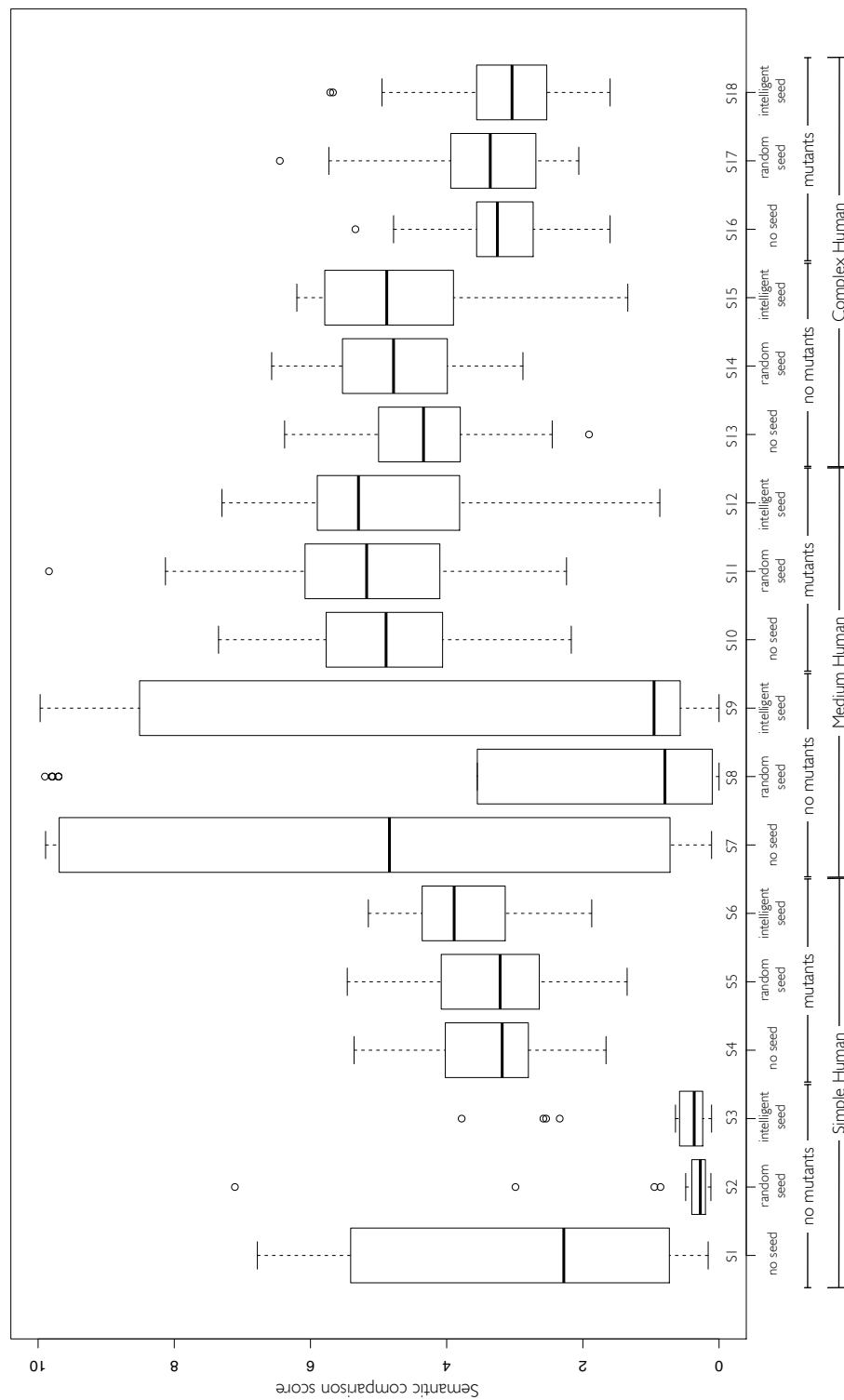
100 randomly generated opponents. Each opponent is fought ten times, and the number of times that the human model (extracted or input) wins is noted. If the human has mutants, one of the mutants is selected randomly for each individual fight, as occurred during the creation of the data model. The semantic similarity score is then calculated as:

$$sim = \frac{\sum_{i=1}^{100} |\#WIN_{input}^i - \#WIN_{extracted}^i|}{100} \quad (5.3)$$

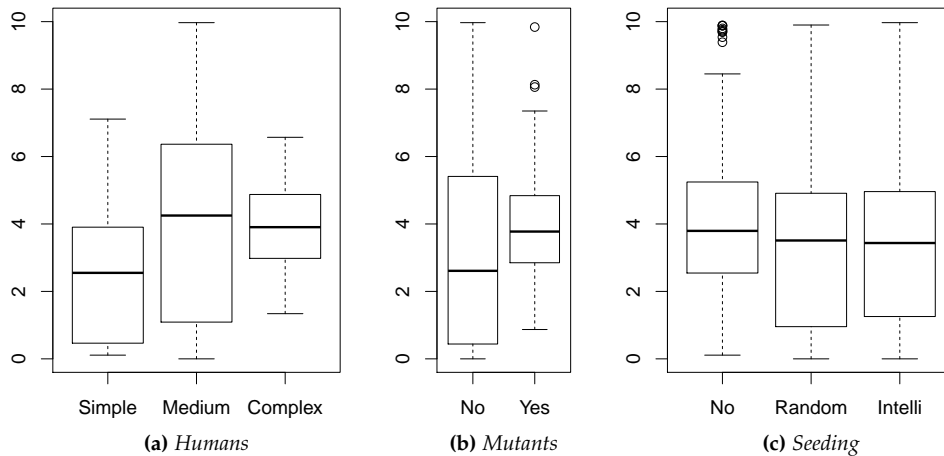
where  $\#WIN_{model}^i$  is the number of times out of ten that the *model* beat opponent  $i$ . Scores range between 0 (semantically equivalent) and 10 (semantically dissimilar).

It is worth emphasising that this semantic comparison would likely not occur at runtime; it is used here purely as a way of validating the approach. In addition to a semantic similarity score, we log the time taken for the experiment to execute from start to finish (i.e. follow the entire path through figure 5.8).

**Results** Figure 5.9 shows the results of each sub-experiment. The increase in problem difficulty caused by adding mutants is apparent when examining the results of the simple human and (to some extent) the medium human. Without mutants the meta-heuristics find near optimal solutions (when seeded), but struggle when mutants are introduced. Unexpectedly, the addition of mutants looks to improve the performance for the complex human; the variation in the results suggests that the fitness landscape is



**Figure 5.9:** *The results of the experimentation. The lines in the centre of the boxes represents the median.*



**Figure 5.10:** The SAF adaptation experimental results, grouped by factor.

noisy, which could attribute towards this phenomenon. The variation in the medium human results also supports the claim that the landscape is noisy. Furthermore, the definitions of human complexity were based on intuition and although more language constructs were employed in the complex fighter, it may in fact behave more uniformly, hence why the results for the complex human appear better than the medium.

The mutant-less simple and medium human results also highlight the effect of seeding. The GA performs poorly without being seeded – likely due to the randomly initialised population containing many more complex solutions than simple ones. This randomness appears to be advantageous for the complex human where the results suggest that it is better to omit than perform seeding – highlighting that our seeding techniques may need re-examining and fine-tuning. Furthermore, the random seed outperforms the intelligent seed for a similar reason – the intelligent seeder creates much more complex solutions using choice and composite rule conditions which are not needed by the two simpler humans.

To determine whether the effects of seeding, mutants and the human’s complexity are statistically significant, we group the data by factor (seeding, mutants, and humans: shown in figure 5.10) and perform a *Kruskal-Wallis rank-sum test*<sup>3</sup> on each group. The Kruskal-Wallis test is a non-parametric test: inspecting the data showed that it is not normally distributed, ruling out the use of ANOVA for the analysis. The results show that, in all three groups, at least one of the levels has an effect; see table 5.5. In order to find out which levels in each group are significant, we performed a pairwise *Wilcoxon rank sum test*<sup>4</sup> on the levels. We found that the results of the simple human are significantly lower than both the medium and the complex humans ( $p = 2.222 \times 10^{-09}$  and  $p = 1.026 \times 10^{-15}$ , respectively). The results of the medium and complex humans are not significantly different (medium

<sup>3</sup> Using R’s `kruskal.test` function: <http://stat.ethz.ch/R-manual/R-patched/library/stats/html/kruskal.test.html>.

<sup>4</sup> Using R’s `wilcox.test` function: <http://stat.ethz.ch/R-manual/R-patched/library/stats/html/wilcox.test.html>.



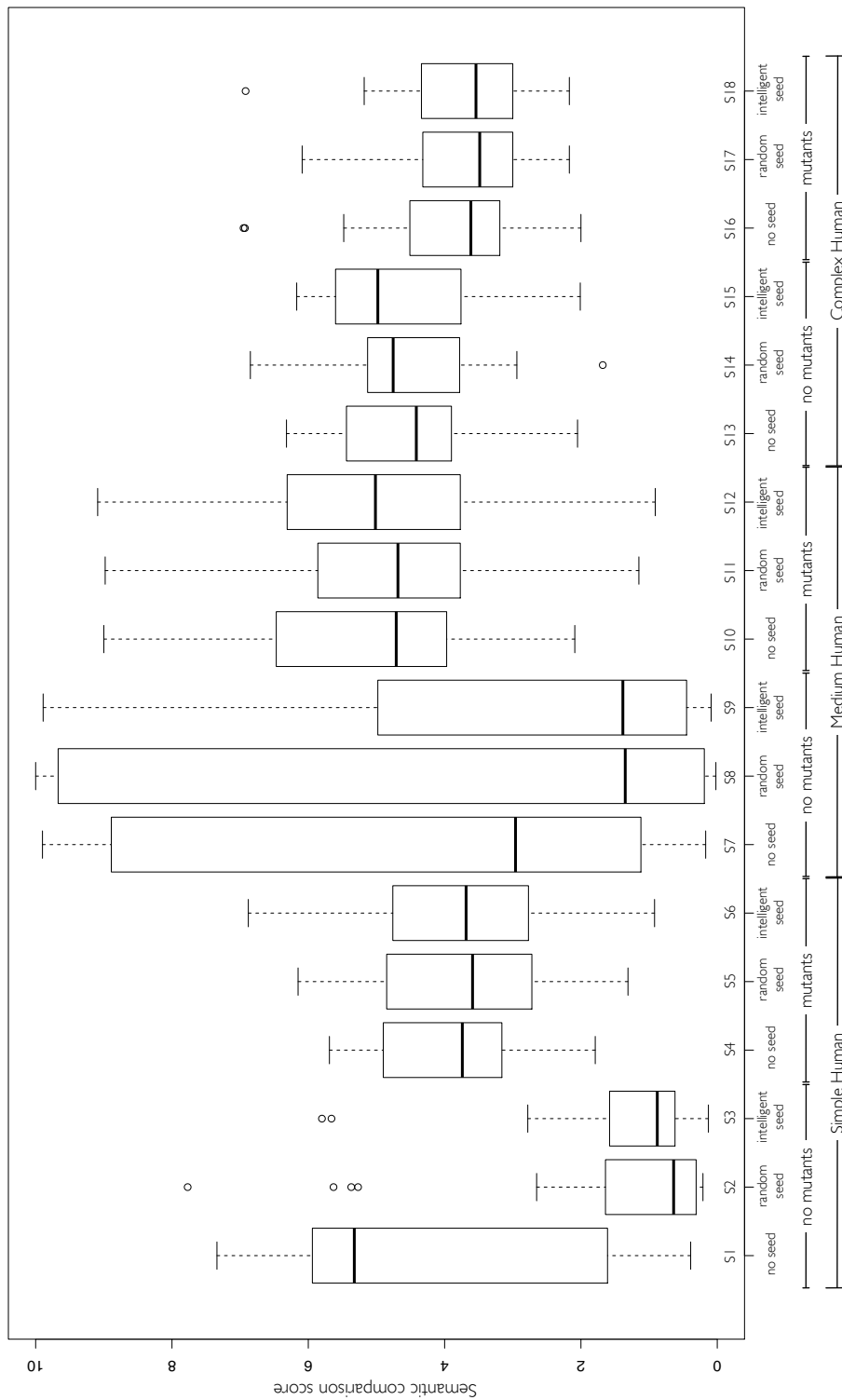
lower than complex:  $p = 0.7955$ ). The results of introducing mutants significantly increases the response score ( $p = 3.857 \times 10^{-06}$ ). Both random seeding and intelligent seeding are shown to produce lower responses than not seeding the population ( $p = 0.0206$  and  $p = 0.01358$ , respectively). No significant effect is found between random and intelligent seeding ( $p = 0.4581$ ).

Group	Chi-squared	DF	P-Value
Humans	63.6123	2	$1.537 \times 10^{-14}$
Mutants	20.0101	1	$7.703 \times 10^{-6}$
Seeding	6.0313	2	0.04901

**Table 5.5:** *Kruskal-Wallis rank sum test for each factor. DF = degrees of freedom.*

**Comparison with Random Search** Even with these factors influencing the performance, figure 5.9 shows that the extracted fighters are not as semantically similar to the input fighter as we would like in a runtime adaptive system. There could be a number of reasons for this: we have made no attempt to optimise parameter selection for the GA, the population size might be too small, the number of generations that the GA executed might be too small, the search space may be rugged or deceptive, or the fitness function may be badly defined. In order to validate that, even though high quality solutions are not found, the GA is performing and that Crepe and MBMS exhibit evolvability, we compare the results against random search. All eighteen permutations of the experiment were repeated with each of the genetic operator probabilities set to 1.0 for the GA responsible for extracting the rules, and all other parameters and stages kept the same. This means that elites are still kept in the population, enabling random search to keep track of the fittest solutions it encounters. The results are shown in figure 5.11. The results exhibit a similar trend to the GA, though random search appears to perform worse on average and has larger variances. The results of a unpaired Wilcoxon rank sum test for each sub-experiment show that the differences between the results of the GA and random search are only significant on six out of the 18 sub-experiments. Four of these are for the simple human (S1–S4), and two for the complex human (S16, S18). Seeding the random search helps for the simple human: the initial population is likely to contain the fittest found at the end of the run. As with the GA, the mutant-less medium human scored either very highly or very badly, emphasising the noisiness of the problem. The results show that the problems that the GA found easy were non-trivial for random search. A Kruskal-Wallis rank sum test on the factor groups also showed all three factors have an effect ( $p < 0.01$ ).

**Execution Time** The experiments were performed on the *Volvox grid*: a computing cluster at the University of York, managed by the *Oracle Grid Engine* software. Execution times for the entire



**Figure 5.11:** The results of random search. The lines in the centre of the boxes represents the median.

extraction process in the original (GA) experiment ranged in the region of under one minute to an hour. The variation in time, and extended periods of some of the experiments is due to heavy load on the grid during the execution of the experiments. Furthermore, the logged times include the opponent generation, trace data generation, and semantic validation phases which would not occur at runtime. As mentioned previously, no effort has been made to tune the performance of the implementation of Crepe or MBMS. While this approach may not be fast enough for systems requiring near-instantaneous adaptation, a tuned implementation may adapt sufficiently quickly for systems where the environment changes more slowly (such as the change in the ability of a game player considered here).

***Tuning the Parameters*** The above experiment was performed with little care in tuning the parameters: the probability of the GA applying any of the genetic operators was 0.15, a value selected based on intuition. To address this, we tune the operator-application probabilities in order to produce a principled set of results. Due to the problem complexity differing between sub-experiment, we tune each sub-experiment's parameters independently.

In order to discover optimal parameter configurations, we apply a *Central Composite Design* (CCD) [116] over the parameter space. A CCD aims to determine the shape of the *response surface* – the fitness associated with combinations of the input parameters. The surface can then be used to select optimal parameter configurations for the given problem. The surface is produced by performing a factorial design over the parameter space. For full details, see [116]. The design was generated using the Matlab `ccdesign` function<sup>5</sup> which, for our five parameters, produced a design consisting of 36 points. The parameter values in each design point are *coded* in the range of -2 to 2. This range represents the values that can be taken by each parameter, and need to be decoded for use in an experiment. Although in reality the probabilities of each genetic operator can range from zero to one, for this experiment we select the range 0.01 to 0.6 as it represents a more realistic range of values that the parameter might take. Anything above 0.6 would tend towards random search.

For this experiment, we execute each design point ten times to gain statistical significance. Therefore, for each of the sub-experiments (S1-S18), we execute 360 runs: 36 parameter configurations, each repeated ten times. The median value is taken from the results of each design point, and used to fit a quadratic model to the design. The (mathematical) model is then used to select the optimal parameters for that sub-experiment. The optimal coded parameters discovered for each sub-experiment are shown in table 5.6. The majority of the extracted optimal parameter values fall at the extremes: -2 or +2. This may indicate that the optimal

<sup>5</sup>[www.mathworks.co.uk/help/stats/ccdesign.html](http://www.mathworks.co.uk/help/stats/ccdesign.html)

parameter values lie outside the selected range, i.e. less than 0.01 or greater than 0.6. The lack of correlation between the choice for parameter values is indicative of a noisy algorithm. The optimal parameter configuration of each sub-experiment was executed ten times and the results are plotted in figure 5.12.

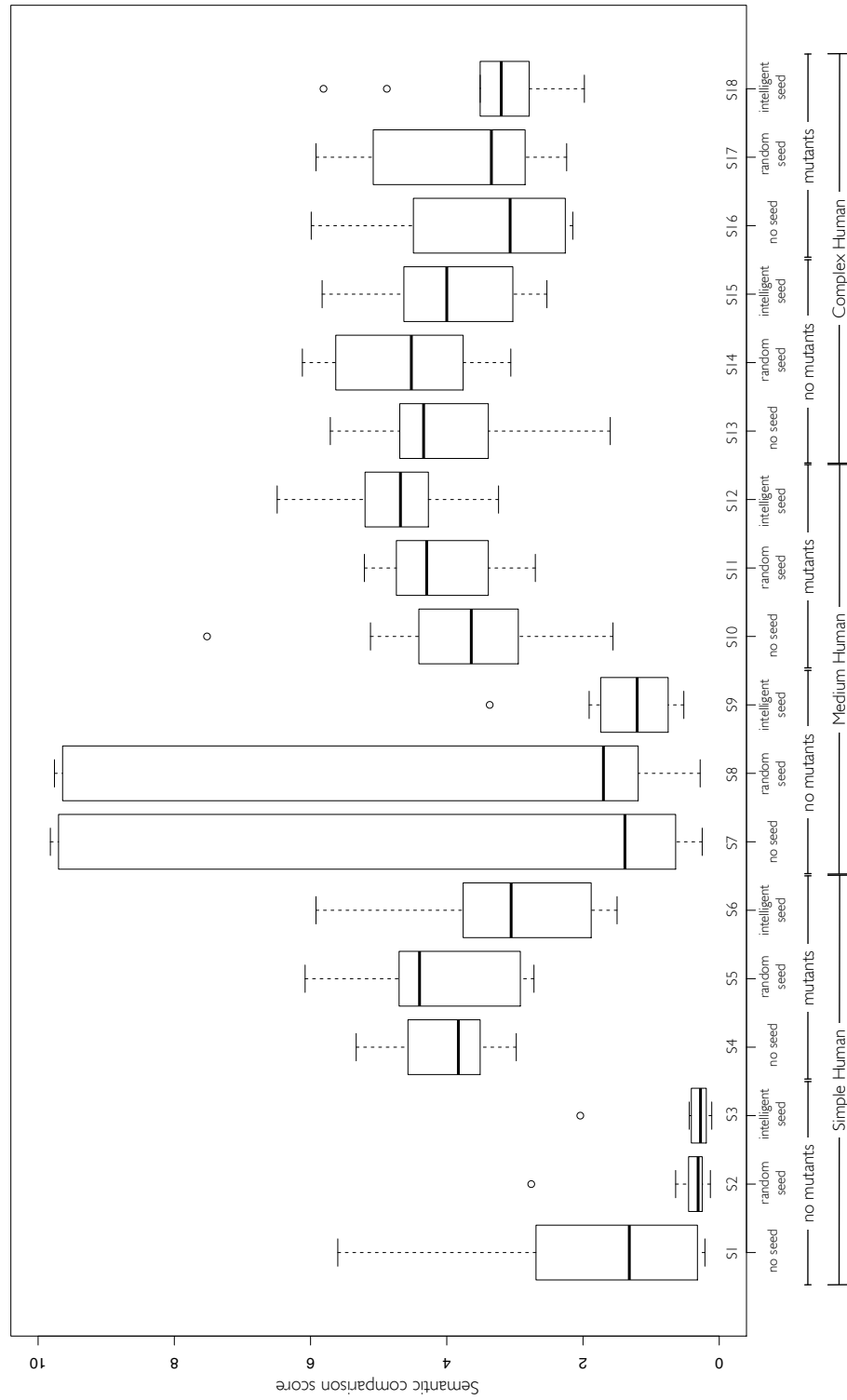
**Table 5.6:** *Optimal coded parameter configuration predicted by the central composite design.*

Exp.	CB	FSB	FVB	CON	DES
S01	-2	-2	-2	-1.882	-2
S02	2	2	-1	2	2
S03	-2	2	2	2	-2
S04	2	1.3931	2	2	-2
S05	2	2	2	2	2
S06	-2	2	2	2	2
S07	-0.9488	-2	2	2	-1
S08	-2	2	2	-1	-2
S09	-2	-0.7356	1.7249	-2	2
S10	-2	-2	-2	2	2
S11	2	-2	2	-2	0.3933
S12	2	2	-2	2	2
S13	-1.9675	-2	-2	-2	-2
S14	2	1.0333	-2	-2	2
S15	-2	-2	2	-2	2
S16	-2	-2	-2	-2	-2
S17	2	-2	-2	2	-2
S18	2	-2	2	0.6867	2

As with the original experiment, we performed a Kruskal-Wallis rank sum test on the three factors. The results (table 5.7) showed that the humans and the mutants have a significant effect on the response, but the seeding does not. This may be due to the fewer number of repetitions of the experiment. As previously, a Wilcoxon rank sum test showed that the simple human outperforms both the medium and complex humans ( $p = 0.002012$  and  $p = 7.272 \times 10^{-06}$ , respectively), and mutants increase the difficulty ( $p = 4.738 \times 10^{-06}$ ). A pairwise comparison (unpaired Wilcoxon rank sum) of each sub-experiment against the original results found that there were statistically significant differences ( $p < 0.05$ ) between the results of S10 and S11, only. Therefore, the tuning had little effect on the overall performance of the algorithm for the problems. This, along with the extracted optimal parameter values indicate that the adaptation algorithm is noisy, and finding optimal solutions is hard.

**Table 5.7:** *Kruskal-Wallis rank sum test for each factor of the tuned experimental results. DF = degrees of freedom.*

Group	Chi-squared	DF	P-Value
Humans	19.6428	2	$5.428 \times 10^{-05}$
Mutants	19.6268	1	$9.414 \times 10^{-06}$
Seeding	3.1094	2	0.2113



**Figure 5.12:** The results of running the experiment at the optimal parameters discovered using CCD.

**Discussion** In this section we used our model-driven, search-based approach to perform self-adaptation in the SAF case study. We attempted to extract an accurate FDL model from a collection of fight logs using a complex multi-stage algorithm. We set up a principled experiment to consider three factors that affect the case study: the human fighter, the behavioural-noise-inducing mutants, and population seeding. We found that the GA outperforms random search on simple problems, and showed that the introduction of mutants and more complex humans increases the difficulty of the problem. Population seeding was shown to have some effect on the algorithms ability to discover optimal solutions, but further investigation is required.

Regarding threats to the validity of our experiments, the main concern is the mutant-based representation of human behaviour. Detailed analysis of SAF-like interactive games is required to determine whether the trace logs produced by the mutants replicate the logs of an interactive game.

### 5.2.5 Summary

The aim of this section was to explore the potential of utilising Crepe and MBMS to enable efficient component-based adaptation at runtime. We presented a model-driven approach to adapting systems at runtime which utilises metaheuristic optimisation in order to, firstly, extract a model of the environment, and secondly, discover the optimal system response. Our focus was on using metaheuristic optimisation techniques to extract a model of the environment's behaviour, and we performed a detailed analysis of the effects of three important factors of the SAF case study. We showed that our representation enabled SBSE techniques to be applied to extracting runtime models, but found the solution space to be noisy. Further work is required to determine whether a model- and search-based approach to system adaptation would be practical for other scenarios. Our representation, Crepe, would enable this strand of research to be taken further. Crepe offers generic way to search for optimal configurations of the adaptable parts of a system; without Crepe, practitioners would need to devise their own problem representation. Crepe allows practitioners to focus on defining good fitness functions, and explore different SBSE techniques with which to tackle their problem.

One issue with our approach is that of performance. Search can be expensive to perform and may not be appropriate for systems whose adaptation time is very small. On the contrary, search can be useful in some time-dependent scenarios, as it can always return a 'good-enough' solution at any point in its execution (as opposed to a constructive approach to model inference which is unable to return a solution until it completes). One goal of this work was to determine whether we could overcome the performance issues found in Goldsby and Cheng's work [59].

No performance-related data was provided by the authors, and so a comparison is difficult. However, we believe that an optimised implementation of the representation and search algorithms could be successfully used for component-level adaptation. Crepe provides a standard approach to generating models, and supports domain-specific adaptation, as opposed to Goldsby and Cheng's state diagram-based approach, and Ramirez and Cheng's system-level adaptation approach [136]. Fitness functions (a major cost in Goldsby and Cheng's work) are also domain-specific, enabling developers to choose the trade off between performance and accuracy.

There is much work that we would like to investigate in the future. We would like to perform further analyses of seeding strategies and search parameters on different case studies to improve performance, and also optimise the search algorithms using some runtime analyses to find bottlenecks. Integrating our approach with Sanchez et al.'s [157] extensible executable runtime model platform may also prove fruitful. Relating to the SAF case-study, it may be interesting to attempt to extract a model based on time: player behaviour may change at different points in a fight, such as the start or end. Furthermore, we may be able to co-evolve the fighter rules and personality to reduce the number of stages in the process. More importantly, we need to analyse the realism that mutants introduce, and apply the techniques to other case studies. In particular, it would be interesting to repeat the experiments in this section on an interactive version of SAF. This would help validate the results found here. Comparing the results of metaheuristic search against other model extraction techniques is also important.

### 5.3 *Summary*

This chapter has illustrated how Crepe and MBMS can be applied to two important challenges in MDE: discovering optimal models of systems, and extracting a model from a corpus of system trace information. We demonstrated that our representation is amenable to search by showing empirically that a genetic algorithm outperformed random search in section 5.1, and on the less difficult sub-experiments in section 5.2 (and was no worse on the other sub-experiments).

In the process of performing these case studies, we discovered that the search space-defining metamodel has a key impact on the searchability of the representation. To address this issue, we defined a refactored, but equivalent, version of the SAF metamodel, limiting the number of references. We postulate that this refactoring may be automated, but this will require the implementation of many case studies and so leave this for future work. In chapter 7, we further analyse how properties of the metamodel affect the representation.





# *Sensitivity Analysis in Model-Driven Engineering*

# 6

ALL SOFTWARE ENGINEERING SUFFERS from some degree of uncertainty [192]. Any form of modelling is subject to different levels of uncertainty, such as errors of measurement or interpretation, incomplete information, and poor or partial understanding of the domain [155]. When model management operations (MMO) are applied to a model, uncertainty can lead to unexpected behaviour, or a small change in a model might result in a large change in the output of the MMOs. Modelling uncertainty can have a significant impact on artefacts that use the models or their information.

A model may contain inaccuracies (i.e., it may not be a faithful abstraction) or errors (i.e., the modeller may have introduced an error). When a model is manipulated with an MMO, these inaccuracies or errors may lead to unexpected or erroneous behaviour, particularly if the inaccuracy or error in the model is highly influential in the MMO's execution. A small change in part of the model might result in a large change in the output of its associated MMO. As all areas of software engineering suffer from some degree of uncertainty [192], a modelling "inaccuracy" could have a huge impact on the artefacts that utilise its information.

*Sensitivity analysis* provides a means to explore how changes in an input model affect the output of an MMO. Sensitivity analysis can provide a modeller with confidence that a model and its associated MMO(s) resemble the domain, and can expose areas in the domain that require a deeper understanding [155]. Furthermore, highlighting sensitive parts of a model can provide insight into the execution of an MMO, if the execution is influenced significantly by sensitive parts of the model [140]. Sensitivity analysis can also show the relationships between model elements that may not be apparent from simply examining a model and its MMO.

In this section we introduce sensitivity analysis in MDE, and thus demonstrate how Crepe can be utilised for tasks other than SBSE. We present an extensible framework that enables metamodel developers to provide sensitivity analysis tool support for their domain. We provide a metamodel for expressing uncer-

## Contents

6.1 Motivating Example . . . . .	136
6.2 Background . . . . .	139
6.3 Sensitivity Analysis in MDE . . . . .	141
6.4 A Framework for Sensitivity Analysis in MDE . . . . .	144
6.5 Case Study: CATMOS and ACM . . . . .	148
6.6 Conclusion . . . . .	151

tainty in existing models, which is used to analyse the effect of uncertainty on the results of manipulating the model with an MMO. Our generic representation for models facilitates systematic model refactoring in order to vary the input model with respect to the specified uncertainty. We then manipulate these variants with the MMO and measure the effect on the MMO's output in order to discover how the uncertainty manifests.

To motivate and illustrate the use of sensitivity analysis, we introduce *CATMOS* [18], an acquisition planning tool for capability management (section 6.1). *CATMOS* generates a set of solution models from a problem analysis; providing an analysis of these solutions improves the basis for acquisition decision-making.

**Chapter Contributions** The contributions of this chapter are summarised below.

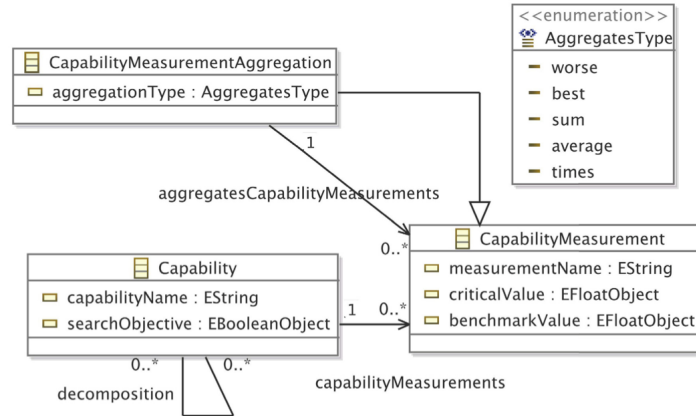
- The identification of three areas where uncertainty can arise in MDE.
- An extensible framework for enabling rapid development of uncertainty-based tool support for models.
- An illustration of applying sensitivity analysis to models produced by a decision making tool, discussing the insights that can be gained and highlighting how uncertainty can affect the confidence one has in a model.

**Chapter Structure** We start in section 6.1 by presenting an example that motivates the need for sensitivity analysis in MDE. Section 6.2 describes uncertainty in more detail and introduces sensitivity analysis, a technique developed in the modelling of natural systems [155]. Section 6.3 discusses how sensitivity analysis might be applied to MDE, identifying the areas where uncertainty arises in MDE, and highlighting the challenges of performing the analysis. In section 6.4, we outline our framework for enabling metamodel developers to provide sensitivity analysis tool support for their metamodels. We show how our generic representation of models can aid in the application of sensitivity analysis. We develop an instance of the framework for the *CATMOS* tool, and apply it to a set of *CATMOS* solution models in section 6.5. Finally, 6.6 considers the pragmatics and challenges of sensitivity analysis in MDE, and discusses how a generic representation of models can aid such analyses.

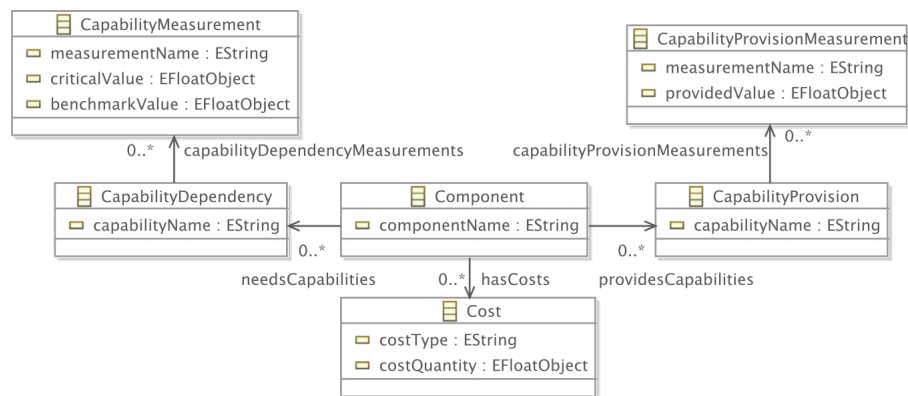
## 6.1 *Motivating Example*

The *Capability Acquisition Tool with Multi-Objective trade-off Support* (*CATMOS*) is an MDE tool that provides support for capability-based planning, to facilitate systems of systems acquisition [18].

CATMOS is targeted at acquisition scenarios that involve significant costs. In these situations, a better understanding of uncertainty can increase confidence in decisions, and thus in investments to be made in acquisition.



(a) Simplified metamodel for defining acquisition scenarios in CATMOS (taken from [18]).



(b) Simplified metamodel for defining components and their capabilities in CATMOS (taken from [18]).

In CATMOS, users model their problem scenario by defining a set of desired system capabilities and associating quantitative measurements with capabilities. Additionally, users define a set of available components: each component has a cost and provides a set of capabilities, each with a measure stating the extent to which they satisfy the capability (some capabilities need multiple components to satisfy them). Components can also have dependencies on other capabilities (e.g. a ‘cup of coffee’ component would help to satisfy the capability of ‘publish research paper’, but it depends on the capability of ‘having hot water’). To illustrate these relationships, figure 6.1 shows the simplified metamodels (taken from [18]) for the scenario modelling language and the component modelling language.

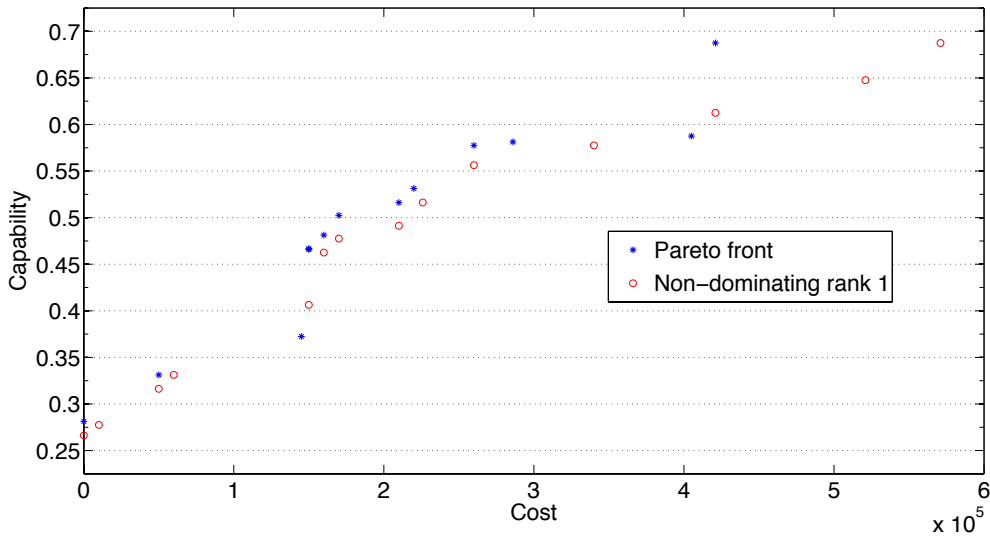
CATMOS generates a set of output models that represent combinations of components that can meet the capability requirements of the scenario. Every solution model has a cost and a capability score, and the solution set can be represented as a Pareto front of capability versus cost (i.e. no solution dominates any

**Figure 6.1:** The two modelling languages of CATMOS.

other in all objectives), which can be used by stakeholders in decision making.

### 6.1.1 The Airport Crisis Management Scenario

In this chapter, we apply sensitivity analysis to a CATMOS application to airport crisis management (ACM). CATMOS is used to inform a decision as to what set of resources should be invested by an airport in order to best respond to a scenario in which a fire breaks out at an airport gate [128]. For this scenario, Burton, the developer of CATMOS, defined 17 capabilities, including: *arrival at fire scene*, measured in time taken; *fire detection*, measured in time; and *treatment*, measured by the number of people treatable. There are 19 components available to the scenario, including: *ambient air analyser node*, to detect fires quickly; *on-site fire truck*, reducing the time to arrival; *ambulances*, which provide patient transport. CATMOS uses the NSGA-II [34] metaheuristic algorithm to discover combinations of components that satisfy the scenario’s capabilities, balancing total capability measurement score against total component cost.



**Figure 6.2:** The Pareto front and first non-dominating rank produced by CATMOS for the ACM problem.

Figure 6.2 shows a Pareto front produced by CATMOS for the ACM case study, in which each star represents one of the solution models proposed. The circles represent models from “non-dominating rankings”: in-effect, the next best Pareto front. The measurements used to calculate the capability (y-axis) are features of the problem scenario that are all subject to uncertainty; the level of uncertainty can affect the interpretation of the results produced by CATMOS. For example, due to traffic, it may be unknown exactly how long it will take for an ambulance to reach the airport in the event of a fire at a gate, and so estimates will be made. If this estimate is well under the actual time taken in the real event, the implications may be serious. Therefore, for instance, if the ca-

pability of a Pareto-optimal solution is subject to high uncertainty, but the capability of a nearby non-dominating solution is less uncertain, the decision makers may prefer the non-dominating solution. Sensitivity analysis allows us to explore how the uncertainty in each of the measurements can affect the calculated capability score of a proposed solution.

We now examine uncertainty in more detail, presenting the related work in the MDE-domain, and describing sensitivity analysis in more detail.

## 6.2 Background

Any form of modelling is potentially subject to uncertainty that can be introduced by numerous possible sources, such as errors of measurement, incomplete information, or from a poor or partial understanding of the domain [155]. Uncertainty has been studied in the software engineering community for some time. Ziv et al. [192] introduced the Uncertainty Principle in Software Engineering (UPSE) in 1997, declaring that “uncertainty is inherent and inevitable in software engineering processes and products”. In this section, we briefly review uncertainty and sensitivity analysis.

### 6.2.1 Uncertainty

Ziv et al. [192] describe three sources of uncertainty in software engineering: the problem domain, the solution domain and human participation. They show how *Bayesian belief networks* can be used to model uncertainty in properties of software artefacts. The framework presented in this chapter targets analysing data uncertainty of individual components (models) in a system, but could be used to provide belief values for those components in the Bayesian belief network of the entire system.

Easterbrook et al. [36] use *multi-valued logics* to model uncertainty through the creation of additional truth values, and examine uncertainty using a multi-valued symbolic model checker. Similarly, Celikyilmaz and Turksen [22] describes how *fuzzy logic* can be used to model uncertainty. Autili et al. [5] describe an approach to cope with uncertainty around the behaviour of reusable components with respect to a given goal. The approach aims to aid people in deciding which components to reuse in their system by extracting behavioural information from the set of possible components and estimating how well they fit together to solve the goal.

Uncertainty in models is often resolved hastily due to the fact that MMOs can only be applied to fully implemented models and developers fear delaying development [40, 42]. Famelis et al. [40, 41] outline a research agenda for systematically and robustly

managing uncertainty in MDE using *partial models*. Many modern MDE tools and languages do not provide a way to specify uncertain information – many tools enforce models to be syntactically correct at all times and do not enable alternative designs to be easily modelled or compared. The partial models approach proposed by Famelis et al. [40, 41] allows modellers to express design alternatives on the same model. Each component in the model is annotated with an identifier and propositional logic is used to specify the combinations of components that make up the design alternatives. This allows modellers to check that required properties hold in the structural design alternatives, and thus filter out bad or infeasible designs. The process of defining partial models in a metamodel-independent manner is shown in [153]. Furthermore, the authors define four kinds of model uncertainty, each of which has a formally defined annotation which can be assigned to partial models. The four kinds of model uncertainty (or *partiality*) defined in [153] relate to: 1) the existence of elements (*May* partiality), 2) the uniqueness of elements (*Abs* partiality), 3) the distinctness of elements (*Var* partiality), and 4) the completeness of a model (*Open World* partiality). Salay et al. [152] present an automated method to verify that a refinement transformation reduces the uncertainty in any partial model. A valid refinement is one that does not increase the set of valid concretisations, whilst ensuring that at least one concretisation remains [152]. Recently, Famelis et al. [42] present an approach to automatically adapt model transformations that enable them to transform models that contain *May* partiality.

As described in the literature chapter (section 2.3.1), Goldsby and Cheng [58], Ramirez et al. [136, 138, 105, 137] and Cheng [23] utilise SBSE to address uncertainty in adaptive systems.

### 6.2.2 Sensitivity Analysis

Sensitivity analysis aims to determine how variation in the output of a model<sup>1</sup> can be attributed to different sources of uncertainty in the model's input [155]. The goal is to increase the confidence in a model by understanding how its *response* (output) varies with respect to changes in the inputs.

In [155], sensitivity analysis is proposed for the following forms of model validation: to determine whether a model is sensitive to the same parameters as its subject; to identify that a model has been tuned to specific input values (and thus is inflexible to model change); to distinguish factors that result in variability of output and those with little influence on the output (which could be omitted); to discover regions of the space of input values that maximise result variation; to find optimal input values (for model calibration); to find factors that interact (and thus need to be treated as a group) and expose their relationships.

Sensitivity analysis approaches broadly fall into local or *one-*

<sup>1</sup> In this section, the term “model” is used in the abstract sense of modelling, and although this includes MDE models, the term should be read in the broader setting.

*at-a-time* (OAT), and *global* analyses. OAT analyses address uncertain input factors independently, revealing the extent to which the output is determined by any one input [155]. Global analyses perturb all input factors simultaneously, and can thus address dependencies among inputs. To avoid combinatorial explosion, sensitivity analysis approaches use *input space sampling* techniques, such as random sampling, importance sampling, and latin hypercube sampling [155]. A range of statistical correlation or regression approaches can be used to interpret the results of sensitivity analysis.

Harman et al. [66] study the effects of data sensitivity on the results of a metaheuristic search algorithm for solving the *next release problem* [7]. Their aim is to identify which requirements are sensitive to inaccurate cost estimation. We believe that metaheuristic search techniques could prove fruitful at the sampling phase of sensitivity analysis. The search goal might be to discover a sample of input factor configurations which produce responses that vary dramatically from the original. The modeller would then be able to examine these extreme cases more carefully.

In the MDE domain, the closest work relating to sensitivity analysis that we have found is by Fleurey et al. [46] who apply mutations to models in order to optimise a set of test models with respect to some metamodel and data coverage criteria. Creating these mutated models is similar to the process of creating variants of a model for sensitivity analysis, but their work is driven by different motivations – that of optimising test sets as opposed to discovering and analysing data sensitivity.

The next section contextualises sensitivity analysis in MDE.

### 6.3 *Sensitivity Analysis in MDE*

Ziv et al.’s uncertainty principle [192] links well with sensitivity analysis, which aims to quantify the effects of uncertainty. In MDE, a *model* in sensitivity analysis terms can be considered as representing both the abstract model of a domain and its operating context – the set of MMOs that are applied to it. The areas of uncertainty are then potentially both the set of variable input factors (model elements) and the parameters of MMOs. The *response* is (part of) the output of the MMO(s).

The uses of sensitivity analysis relate to validation: determining whether a model faithfully represents its domain, or that a model is faithful to a more abstract specification. Model validation in MDE is important, and is commonly addressed using task specific languages, such as OCL [61] or EVL [93]. Sensitivity analysis provides an alternative, exploratory approach to validation. Furthermore, it presents an opportunity to understand the effects of modelling decisions, which can be crucial in complex or poorly understood domains.

The works of Famelis, Salay et al. [41, 153] on partial models addresses uncertainty from a different angle. They provide a powerful, formal way to model uncertainty and provide proofs that refinements reduce uncertainty or that transformations maintain uncertainty, and check properties of different design alternatives. This allows the modeller to better visualise uncertainty and take care in removing uncertainty whilst not slowing down development. They do not, however, allow for the effects of uncertainty to be explored. Sensitivity analysis offers model uncertainty to be investigated in more detail.

In this section, we define three areas where uncertainty can arise in MDE, and discuss the challenges of measuring the response of an MMO in order to utilise sensitivity analysis.

### 6.3.1 *Areas of Uncertainty*

The sources of uncertainty are the same in software engineering as in scientific modelling (errors of measurement or interpretation, incomplete information, poor or partial understanding of the domain [155]). We identify three areas of uncertainty in MDE and consider ways in which we might quantify their effects on the application of MMOs.

**Data Uncertainty** In section 6.1, we note that the capability measurements defined in CATMOS models is affected by uncertainty. When modelling data structures, uncertainty is introduced when deciding types and values of attributes – for instance, string types and \* multiplicities are often used because of uncertainty about the domain. In modelling transitional systems (e.g. using state machines), uncertainty arises in determining the values used in transition guards. In modelling reactive systems (e.g. using Petri nets), the firing conditions of transitions may be similarly uncertain. Strengthening or weakening Boolean conditions may significantly affect the behaviour of systems. Sensitivity analysis can be applied to attributes, guards and firing conditions, to validate the states and behaviours of modelled systems.

To analyse data uncertainty, we can vary the values of attributes with respect to a range or distribution of possible values, and see how the MMO output is affected.

**Structural Uncertainty** Uncertainty also arises when making decisions about the structure of a model. One example of structural uncertainty is deciding between aggregation and composition for an association. This decision could have a huge effect on an MMO. For example, if the MMO deletes the owner element then the type of association determines which elements remain available for the remainder of the MMO's execution.

Partial models [40, 41] offer a means to explore structural uncertainty. This analysis might alternatively be achieved through



pattern matching and replacement. A modeller could define a set of patterns of model elements (possibly with respect to the underlying metamodel) where each pattern represents a set of equivalent patterns. Sensitivity analysis could then be used to analyse the effect of replacing parts of the model with different patterns, to discover the effect on the output of the MMO. This would allow the modeller to optimise their models in conjunction with the associated MMOs. Alternatively, one might perform simple, but well-defined, mutations to a model (e.g. deleting elements) and analysing the effects of these mutations on the output. This would highlight the parts of the model that are important to the execution of the MMO(s) and thus sensitive to change.

***Behavioural Uncertainty*** Whereas the previous categories relate directly to models, behavioural uncertainty relates to the operating context of the model – i.e. the set of MMOs that consume it. The model and its MMO(s) may be developed by different, independent teams, and it may not be known to the modeller what the operation does or how it does it. Knowing the operating context of a model can help to alleviate uncertainty: for example, a modeller may make a different design decision if the model is to be used for code generation as opposed to illustration. Analysing behavioural uncertainty is challenging, though has been attempted in [5] and [59].

In the rest of this chapter, we focus on analysing data uncertainty. First, we consider the issue of measurement.

### 6.3.2 *Measuring the Response*

Sensitivity analyses in scientific modelling often address readily-evaluable parameters. However, in MDE, the result of applying an MMO is often another model (e.g. in the case of a model-to-model transformation), so it can be challenging to provide a useful measure of the effects of changing the input model. A simple count of differences between the original output model and each of the output models created from varying the input, may not be a suitable measure of impact. Focusing the analysis on the effects on a small part of the output model, however, may be more appropriate. In the cases where a model is used to generate code, the response might be measured by executing the generated code and analysing, for example, the execution trace, the memory consumption, or the program's output.

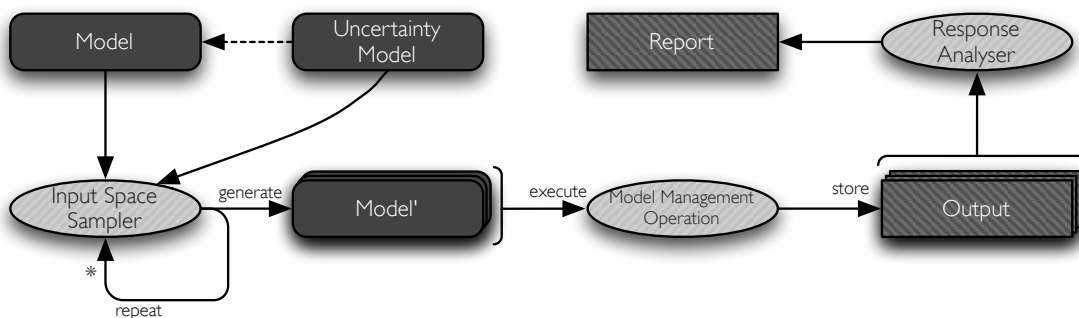
Measurement is domain specific. Saltelli et al. [155] comment that different measures of sensitivity directly affect the outcome of the analysis, and declare that there is no universal recipe for measuring the response. Any MDE framework that supports sensitivity analysis needs to provide the ability to support multiple forms of response measurement and comparison.

## 6.4 A Framework for Sensitivity Analysis in MDE

This section presents our implementation of an extensible framework for applying sensitivity analysis to models in MDE. Specifically, the framework is implemented on top of the Eclipse Modeling Framework (EMF) [170]. The framework provides a repeatable way of allowing metamodel developers to create tool support for rigorous analysis of models in their domain. Section 6.4.1 overviews the process which our framework implements. The framework utilises Crepe to make the creation of variants of a model straightforward. We introduce a metamodel for expressing data uncertainty in section 6.4.2. Section 6.4.3 describes the framework's support for different sampling methods, and presents two default methods. Section 6.4.4 shows how we provide support for automated analysis of responses, as well as producing a sensitivity analysis report for the modeller.

### 6.4.1 Overview

Figure 6.3 illustrates the process of applying sensitivity analysis to a model. First, we need an *uncertainty model* that expresses the data uncertainty in the *input model* (explained in section 6.4.2). These two models are fed into an *input space sampler* – a bespoke model generator that uses Crepe to produce variations of a model within the scope described by its uncertainty model. Selection is controlled by the sampling method of sensitivity analysis being applied. The framework currently supports two methods of sampling the input space: *one-at-a-time sampling* and *random sampling* (described in section 6.4.3).



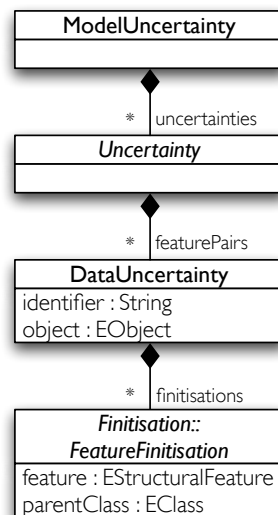
**Figure 6.3:** The process of applying sensitivity analysis to a model.

Each of the generated models (*Model'*) in the sample is executed against the associated set of *model management operations* and the *output* is logged. The model management operation may be a model transformation, a simulation, or a complex workflow

of operations, as determined by the user. Once all generated models have been executed, the set of outputs is fed into a *response analyser* which applies the sensitivity analysis and produces reports for inspection. As mentioned previously, there is no single, optimal way to analyse sensitivity, and the output of an MMO can take many forms. Therefore, whilst providing a number of default response analysers for numerical output, our framework also allows users to develop their own (see section 6.4.4).

### 6.4.2 A Metamodel for Data Uncertainty

In order to apply sensitivity analysis, the modeller is required to enumerate the uncertain parts of a model. Partial models would be one approach to capturing this information. However, in our framework we capture uncertainty in a separate model. This allows developers to manipulate this information with MMOs and separates the concerns of uncertainty from the model. Figure 6.4 shows the metamodel that we have defined to allow users to describe the data uncertainty in their models. The metamodel appropriates the *FeatureFinitisation* class from the finitisation metamodel presented in section 4.1 (figure 4.2). This allows us to simplify the genotype-phenotype transformations, which usually require a finitisation model to be defined. Instead, the finitisation model is automatically computed from the uncertainty model and the values already in the model.



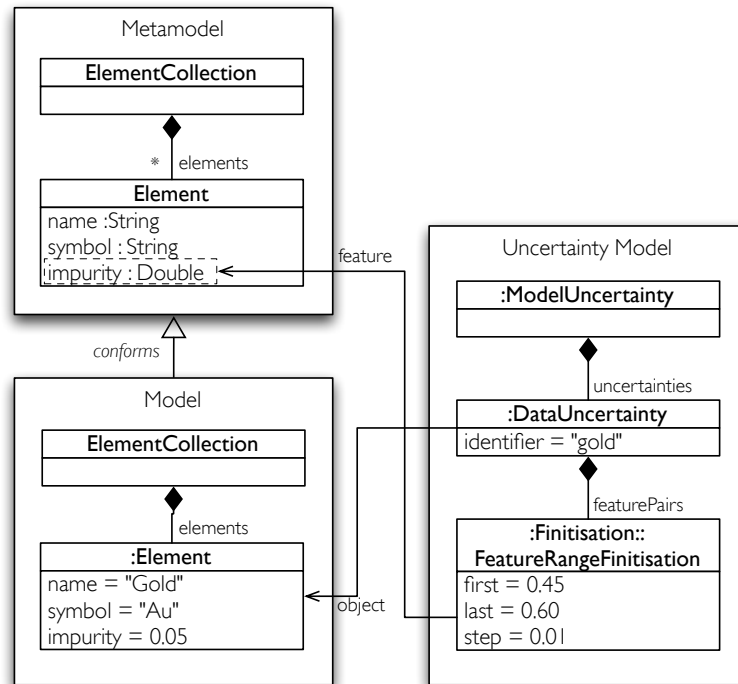
**Figure 6.4:** The metamodel for expressing uncertainty in models.

An uncertainty model can define a set of *DataUncertainty*s. Each *DataUncertainty* object has an *identifier*, used to distinguish between the uncertain objects (input factors) during the response analysis. Assigning the object reference of a *DataUncertainty* object allows a modeller to assign the uncertainty definition to a specific object in the model. A *DataUncertainty* object's *finitisations* define the uncertain values of a given feature, by reference to the metamodel (as is standard finitisation practice – see section 4.1). Uncertainty val-

ues can be defined using the ranges, lists of values, or references to objects in other models. If the user does not assign the object reference of DataUncertainty, then the uncertainty values defined by the finitisations are valid for all instances of that attribute.

We illustrate this in figure 6.5. The metamodel describes a Collection of scientific Elements. Elements have names, chemical symbols, and impurity percentages. The example model shows an instance with just one element in the collection: gold. The value of gold is based on its impurity and the collector is unsure whether his impurity measurement is accurate and so she defines an uncertainty model, specifying the range of impurities. The gold value calculator could be used as the response measure to understand how the level of impurity will affect the value of the collection. If the object reference from the DataUncertainty instance is not defined, then all elements in the collection would have the impurity uncertainty associated with them.

**Figure 6.5:** An example uncertainty model used to capture the uncertainty regarding the impurity of elements in a collection.



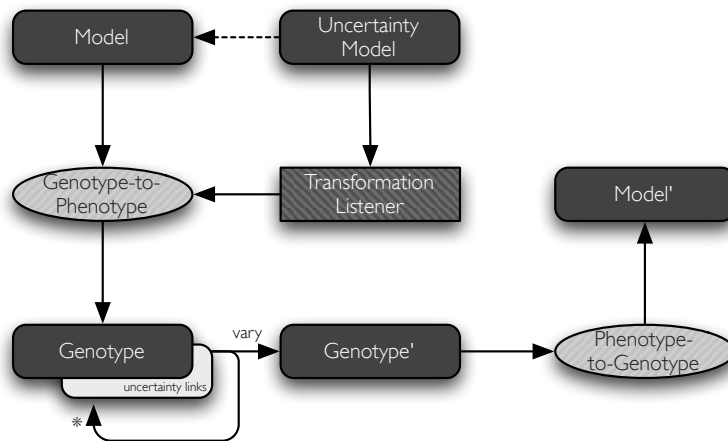
The uncertainty metamodel has been designed with evolution in mind. Currently the metamodel only supports data uncertainty, but the metamodel can be extended to capture other forms of uncertainty (by extending the Uncertainty class) whilst maintaining backward compatibility with existing uncertainty models.

We now describe how different sensitivity analysis methods can use the uncertainty model and Crepe to create variations of a model for which to analyse.

### 6.4.3 Sampling the Input Space

The process of sampling the input space (i.e. generating variants of a model) is illustrated in figure 6.6. The variation of a model is created by mapping the model to its genotypic form and altering the appropriate feature value bit before transforming it back to a model which contains the new data. We incorporated hooks for *listeners* into the phenotype-to-genotype transformation to detect when segments and feature pairs are created<sup>2</sup>. We have defined a feature pair creation listener which detects whether the feature pair being created represents one of the features described in the uncertainty model. If so, we keep a reference to that feature pair, along with the associated values described in the uncertainty model. During the mapping, therefore, we keep track of all *uncertain feature pairs*.

<sup>2</sup> The work presented here was developed for the original Java implementation of Crepe. All aspects of the processes described here are still possible in the current EOL-based implementation.



**Figure 6.6:** Utilising our representation of models to create variants of a model with respect to a set of specified uncertainties.

Once the phenotype-genotype transformation is complete, the desired sampling method controls which of the uncertain feature pairs to vary. Each variation of the segment list is mapped back into a model for consumption by the MMO. As there is commonly a large number of possible variations, our framework allows users to create custom sampling methods and provides two by default. Our *one-at-a-time* sampler adjusts each attribute independently, creating samples for each possible value that each attribute can take, whilst maintaining all other attributes in their original form. Our *random sampling* method adjusts all attributes simultaneously, selecting a value for each attribute at random from the uncertainty model. The user specifies the size of the sample that they desire.

### 6.4.4 Response Measures

The input space sampler produces a set of variations of a model which are then fed through the MMO to obtain the associated responses. The final component in our framework is the *response analyser* which provides various analysis methods and a HTML

report generator. Framework implementations can choose which analyses to apply and include in the report. Furthermore users can extend the analysis methods with custom methods, or integrate with existing statistical analysis packages, such as *jHepWork*<sup>3</sup>, *R*<sup>4</sup>, or *MatLab*<sup>5</sup>. The results are saved to disk as a comma-separated-values (CSV) file, so users can apply further analyses as they gain a deeper understanding of the results. (In future iterations of this work, we plan to develop a metamodel to capture the results as models. Each analysis method will then be written as a model transformation, removing the burden of using CSV files and improving reusability.) The reporting component of the framework utilises an open source template engine, *StringTemplate*<sup>6</sup>, and users can define their own templates to incorporate new analyses in the reports.

Currently the framework provides scatter plot based analysis for both OAT and global sampling methods, as well as some useful statistics such as mean, median, percentiles and standard deviations. JFreeChart<sup>7</sup> is used to create a scatter plot for each of the uncertain attributes, illustrating the effect that the attribute has on the MMO's response.

### Summary

This section has presented an overview of our extensible framework for supporting the analysis of uncertainty in models. Metamodel developers can extend the framework to provide sensitivity analysis support for their modelling language. Users are required to develop a model that captures the uncertainty in their model under test. They can then make use of the pre-defined input sampling algorithms and response measures, utilise those provided for their domain by the metamodeler, or develop their own. We now present an instantiation of this framework for the CATMOS tool in order to analyse the airport crisis management scenario described in section 6.1.1.

## 6.5 Case Study: CATMOS and ACM

Section 6.1 introduced an acquisition decision tool, *CATMOS* [18], and a case study to which the tool has been applied. We have developed an instance of our sensitivity analysis framework for the *CATMOS* tool. This section briefly presents our implementation and describes some interesting results, highlighting the utility of sensitivity analysis and supporting the call for its adoption by the MDE community.

<sup>3</sup> <http://jwork.org/jhepwork/>

<sup>4</sup> <http://www.r-project.org/>

<sup>5</sup> <http://www.mathworks.co.uk/>

<sup>6</sup> <http://stringtemplate.org/>

<sup>7</sup> <http://www.jfree.org/jfreechart/>

### 6.5.1 CATMOS Sensitivity Analysis

Extending the framework to support CATMOS required defining two Java classes, totalling approximately 300 LOC. Although more would be required to provide a more sophisticated user interface, extending the core components of the framework is trivial. One of the two classes controls the loading of models, specifies the sampling method(s) and starts the analysis, and the other provides the response calculation, controls the types of sensitivity analysis applied to the set of responses and generates reports. For repeatability, we have made the models analysed and generated reports available online<sup>8</sup>.

<sup>8</sup>Results available at: <http://www.jamesrobertwilliams.co.uk/models12-sa>

The experimental goals that we wanted to analyse were:

- G1 Determine how each factor contributes to the overall capability score (the response) of the model;
- G2 Understand how the different model factors relate to one another;
- G3 Provide insight into the confidence of the frontiers produced by CATMOS.

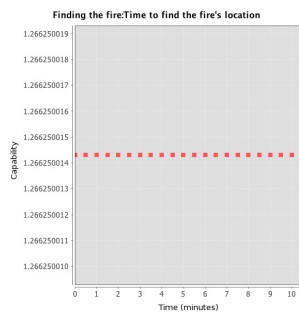
Uncertainty values were determined in discussion with Frank Burton, the creator of CATMOS and the ACM case study. Each of the capability measurements provided by a component was deemed to be uncertain. CATMOS uses these measurements to calculate the overall capability score of the solution – varying these factors will affect this score, and therefore the ability of the solution to satisfy the requirements of the scenario. The response measure, therefore, was the CATMOS function used to calculate the capability score of a solution.

We applied OAT sampling and random sampling to create scatter plots for each factor in the 28 solution models (the Pareto front and first non-dominating rank) produced by CATMOS for the ACM case study. We developed a custom response measure which produces a plot showing the response distribution, based on the sampling, for each of the models in the solution set. In total, we evaluated nearly 20,000 model variants which took approximately 3 hours to execute on a 2GHz Intel Core 2 Duo Macbook with 2 GB of RAM.

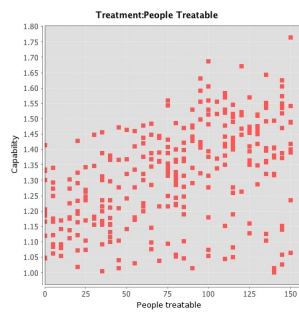
### 6.5.2 Results

The results found from attempting to answer the experimental goals described in the previous section are now presented. We do not aim to provide a detailed analysis of the results: we aim to illustrate the kinds of knowledge that can be extracted from using sensitivity analysis in MDE.

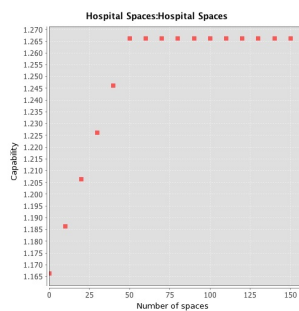
**G1: Response Contribution** Figure 6.7 illustrates three types of response contribution which were observed. Figure 6.7a shows



(a) No effect



(b) Positive Correlation



(c) Upper limit

**Figure 6.7:** Three different types of contribution towards the response of CATMOS.

a factor which has no effect on the calculation of the capability provided by that solution (response of the MMO). One possible cause of this is that the range of values analysed were not appropriate, and more discussion with a domain expert would be required. However, upon analysing the solution more closely, it became clear that the component providing the particular capability had an unfulfilled dependency, meaning that it actually had no effect on the output. In this case, the offending component should be removed from the model to avoid misleading decision makers.

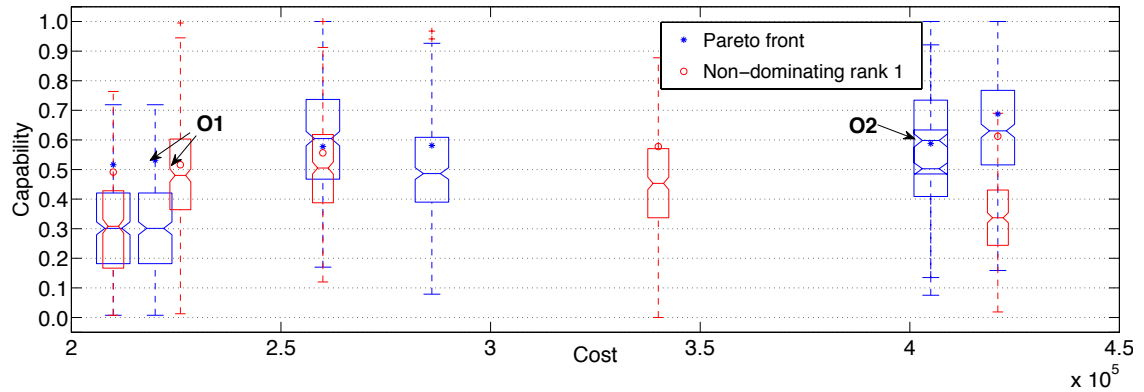
Figure 6.7b shows the effect that the “People Treatable” component had during the random sampling method. You can see that there is a positive correlation, highlighting the importance of this factor as it is heavily influential in determining the capability score. Furthermore, it validates the assumptions of the model – a solution that is able to treat more people should have a higher capability score. Other factors also showed relationships with the response that further validated the model. Finally, figure 6.7c illustrates a factor which exhibits an upper limit. This kind of result proves useful in understanding the problem, allowing stakeholders to act out ‘what-if’ scenarios.

**G2: Factor Relationships** 28 solution models were analysed and examining the OAT results for each model showed that some factors affect on the response differs between solutions. This shows a dependency on other factors in the model. Figure 6.7c, for example, is limited by the factor defining the number of people who can be transported to the hospital. Examining the random sampling plots shows little to no correlation for most of the factors. Therefore, further analysis is required to understand the relationship between multiple factors. This is, however, out of scope.

**G3: Frontier Confidence** Figure 6.8 shows the response distributions, provided by random sampling, overlaid on top of part of the original Pareto front and first non-dominating rank produced by CATMOS (figure 6.2). Two interesting observations can be seen. Firstly, **O1** in figure 6.8 highlights the case where a first non-dominating rank solution (red circle) appears to be, on average, better than a solution in the Pareto front (blue star). Secondly, **O2** highlights a case where CATMOS has returned two solutions in the Pareto front with the same capability and cost (this is allowed by CATMOS). A decision maker, may therefore think themselves safe in selecting either solution. The associated box plots, however, show that they have different interquartile ranges – suggesting that one solution is more sensitive to uncertainty than the other.

One flaw with this analysis method is that it does not take the probability distribution of each factor into account. For example,





the random sample may have collected a large number of improbable combinations of factors, which then skewed the distribution. We plan to address this in future work (see section 6.6).

### 6.5.3 Summary

In this section we have shown how our implementation of the framework for the CATMOS tool has provided new insight into the results which CATMOS produces. This has shown how sensitivity analysis can provide deeper understanding of a problem, aid in the validation of a solution, and discover areas of a solution which need further investigation by domain experts.

**Figure 6.8:** Part of the random sampling response distribution for each model in the Pareto front and first non-dominating rank produced by CATMOS. Box width simply denotes the series – the wider boxes being related to the Pareto front.

## 6.6 Conclusion

In this section we have motivated the need for supporting sensitivity analysis in MDE. Uncertainty can appear in all aspects of MDE; we have defined three categories of uncertainty – data uncertainty, structural uncertainty, and behavioural uncertainty. These extend and complement the uncertainty that can be expressed in Famelis and Salay’s partial models [41, 153]. Due to the need for domain specific analysis of uncertainty, we have presented a framework that enables metamodelers to provide sensitivity analysis tool support for models conforming to their metamodels. This framework has an architecture which allows new sampling methods and response measures to be integrated, making it easy to tailor to new domains. Sampling is achieved using Crepe by altering the genotypic form of the input model with respect to user-defined alternate values. We have illustrated the usefulness of this kind of framework by instantiating it to support an existing acquisition tool, and applied the analysis to a real problem. The analysis provided new insight into the solutions produced by the tool, which would better inform the acquisition decision makers.

There are a number of interesting directions for future work. Firstly, a deeper exploration of uncertainty in MDE is required. Uncertainty should be identified and dealt with, and so understanding where it can arise, and providing a way to aid users with managing uncertainty, would be very beneficial. This could possibly be in the form of tool support, or a set of guidelines or best practices. With respect to the framework, we plan to extend the uncertainty metamodel to account for structural and behavioural uncertainty, and devise a method to analyse these. For example, for structural uncertainty, changing the type of an attribute would require changes to the underlying metamodel, which may not be trivial, and so some thought is required to determine the best approach to take. It would be interesting to see if we use these ideas to search for partial models. Partial models capture design alternatives, but these may not be complete: the modeller may have not considered all possibilities. Employing SBSE techniques to search for alternative, constraint satisfying, designs could present the modeller with a more complete partial model, enabling them to more thoroughly address uncertainty.

Additionally, we wish to include support for probability distributions for specifying data uncertainty. This would add an extra dimension to the analysis as it would also show the likeliness of a particular result occurring. We also plan to develop a metamodel to capture the results of analysis in order to define analysis methods as model transformations, and thus increase reusability of analyses. Finally, although Crepe is only used here for model variant generation, we can utilise it further to search for input factor configurations that result in large response deviations. Sampling techniques may miss highly deviating configurations, but search could be used to discover them. This would allow modellers to inspect the worst case scenarios, and either discard the scenario or improve their model accordingly.

# *IV*

## *Evaluation and Conclusions*



# *Analysis of the Properties of the Model Representation*

# 7

THIS THESIS HAS PRESENTED a metamodel-agnostic representation of models that is amenable to many standard SBSE techniques, and has applied it to discovering optimal models, extracting models from system behaviour traces, and utilised it to analyse the effects of model uncertainty. Up to this point there has been little consideration of the properties of the representation or how the different genetic operators affect the evolvability of the representation. This chapter presents a preliminary study that aims to answer questions related to these topics. In particular, we focus on understanding the *locality* and the *redundancy* of the representation – two properties that have been shown to be important for the performance of evolutionary search. Regarding the effects of the adaptation operators, section 5.2.4 applied a *Central Composite Design* [116] over the probabilities of each operator’s application on the SAF adaptation case study to find the optimal parameter configurations for each sub-experiment. The optimal parameter configurations discovered highlighted that the SAF adaptation problem suffers from noise, and so we cannot draw many conclusions from the results. In order to truly understand the effects of the adaptation operators we would need to analyse numerous, well-defined search problems, and so we leave an exhaustive study for future work.

To effectively analyse properties of the representation, we need to do so with respect to multiple metamodels, as the different characteristics of metamodels affect the properties of the representation. Indeed, by analysing metamodels with different characteristics, we can begin to understand *how* these characteristics affect the representation and therefore guide users towards desirable search parameters for the metamodel used in their problem. As such, this chapter also includes a structural analysis of a corpus of more than 500 publicly available metamodels in order to determine what metamodels commonly look like. The analysis focuses on metamodel structure due to the fact that our genotype-phenotype transformations use the metamodel’s structure to perform their mappings. Therefore, different structural characteris-

## Contents

7.1	Properties of Representations	156
7.2	What do Metamodels Really Look Like?	159
7.3	Locality	168
7.4	Redundancy	179
7.5	Discussion and Future Analysis	189

tics of the metamodel will affect the transformation. From this corpus we select representative metamodels to use for the analysis of our representation, or other similar representations.

*Chapter Contributions* The contributions of this chapter are as follows.

- The definition of 19 structural metrics of metamodels and the automated analysis of a corpus of 500 publicly available metamodels – the largest known such analysis to date. This provides utility not just in evaluating our representation, but in understanding how metamodels are being developed in practice, highlighting common traits of metamodels, and practices of metamodelers.
- The identification of four metamodels that are representative of the corpus which can act as benchmarks for analysing the properties of the representations of models.
- The empirical analysis of the locality of our representation, gaining insight into how the metamodel that defines the problem space affects this.
- The empirical analysis of the redundancy of our representation, also discovering the effects of different characteristics of the metamodel that defines the problem space.
- A plan for future empirical analysis to gain a deeper understanding of the properties of the representation and the influence that the metamodel has both on these properties and on the metaheuristic.

*Chapter Structure* We start in section 7.1 by describing the properties of representations which have been shown to be important in efficient evolutionary search. We also discuss the different features of our representation that will affect these properties. Our analysis of the structural features of a large metamodel corpus is presented section 7.2. Section 7.3 presents an empirical evaluation of the locality of the representation on four representative metamodels, selected from the corpus. Section 7.4 describes an empirical analysis of the redundancy of the representation with respect to the four representative metamodels. Section 7.5 discusses other properties of the representation previously not addressed and defines a plan for further detailed analysis.

## 7.1 *Properties of Representations*

Rothlauf [147] is the seminal work on representations for evolutionary algorithms. A representation should, at least, be able to express all possible solutions to a problem [147]. Moreover, the

representation can have a dramatic effect on an evolutionary algorithm’s ability to solve a problem – in some cases turning simple problems into challenging problems, or challenging problems into more manageable problems [147]. The *locality* and *redundancy* of a representation are two properties that have been shown, theoretically and empirically (see [147]), to influence the performance of evolutionary algorithms. These were described in section 2.2.2. Locality relates to the effects that small changes to a genotypic individual have on its phenotypic form. A representation is said to ‘have locality’ if neighbouring genotypes correspond to neighbouring phenotypes [52, 147]. Redundancy relates to areas of the genotype that are not used in the mapping to the phenotype [147]. This section examines two particular aspects of our representation of models which may affect these properties.

### 7.1.1 Features of our Representation

When using our representation, there are a number of configuration parameters that the user can tailor to suit their problem (see Table 4.1). Each of these parameters will influence the properties of the representation described above. This section discusses the effects of the representation-related parameters.

**Maximum Allele Value** When initialising a population of individuals at the start of the execution of an evolutionary algorithm, it is common to create each member of the population at random (we showed in section 5.2.4 the benefits of *seeding* the initial population). For example, in a binary representation, each allele is randomly set to be a zero or a one. In an integer representation where the mapping process takes the allele value modulo some other value (such as in Grammatical Evolution or our representation), the maximum allowed allele value has an impact on the phenotypes that can be represented (and therefore discovered by search). If this value is too low then not all phenotypes can be represented, and if this value is too high then there will be more genotypes than phenotypes. In other words, the maximum allele value plays an important role in creating redundancy in the representation. Furthermore, the maximum allele value has some impact on the locality of the representation. An overly large value will create extra, redundant phenotypical neighbours which increases non-locality.

Although there are proponents on both sides of the argument, it is commonly believed that redundancy should be limited as it can result in inefficient search. Goldberg [56] argues that the alphabet used to define the representation should be as small as possible. Therefore, the maximum allele value is defined as:

$$\max(\#seg, fin_{max}) \quad (7.1)$$

where  $fin_{max}$  is the largest number of finitisation values given to

a single feature in the finitisation model. *#seg* is required in case there is an object with an association that can reference every object in the model (including itself). This ensures that all segments and all finitisations are able to be referenced by a feature pair, thus ensuring that the encoding is complete with respect to the length of the individual. We discuss further the effects of the maximum allele value in section 7.4.

***Individual Length*** As well as the maximum allele value, the length of an individual can impact both redundancy and the size of the model space (i.e. the number of possible phenotypes that can be encoded). In our representation, the number of segments in an individual impacts the number of objects that can appear in the associated model. Adding more segments to that individual allows for larger models to be represented. Similarly with feature pairs, the more feature pairs present in a segment, the more object features can be assigned values (this is particularly important for references with large upper bounds). If the individual length is fixed, then to discover a solution model that consists of very few objects, the search algorithm would need to find solutions with many redundant segments (which may not be possible). Conversely, if the solution model is large, it may not be able to be encoded by the limit of the individual's length – meaning that some knowledge of the solution is required to select an appropriate genotype length. It is for this reason that our representation allows users to specify the minimum and maximum number of segments and feature pairs per segment, thereby creating individuals with varying lengths. The segment creation and destruction search operators are then able to increase or reduce individuals. Incidentally, Miller et al. [111] demonstrate an increase in the evolvability of the search algorithm with extremely large genotypes with very high redundancy (>90%) for two standard problems.

Throughout this chapter we will empirically demonstrate the effects of these two aspects of our representation, illustrating in particular how the length of an individual has significant effect on both locality and redundancy.

### 7.1.2 *Summary*

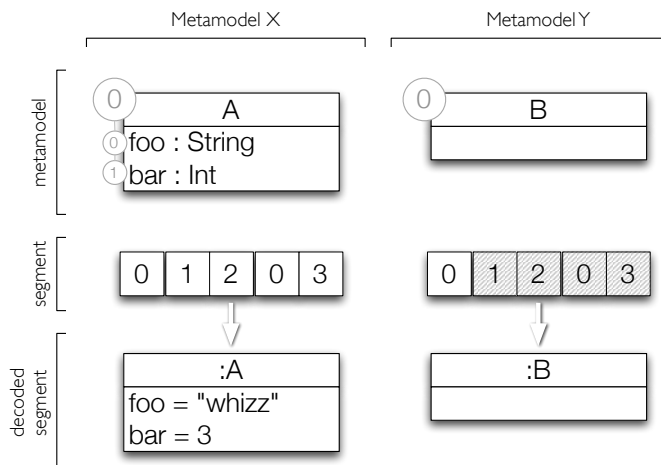
In this section we have discussed the aspects of our representation of models which affect two key properties of representations. We touched upon the idea that the metamodel being searched over influences the properties of the representation. In the next section we perform an analysis of a corpus of publicly available metamodels to discover common structural properties of metamodels, and use this to select a set of representative metamodels with which to analyse the properties of our representation.



## 7.2 What do Metamodels Really Look Like?

**Note:** The work described in this section is based on joint work with Athanasios Zolotas, Louis Rose, Nicholas Matragkas, Dimitrios Kolovos, Richard Paige and Fiona Polack, and has been published in [187]. All work presented here was performed by me, with the exclusion of the associated related work section which was originally written by Zolotas and Matragkas. The metamodel corpus was collected by Matragkas. The paper [187] includes extensions of the statistics presented here that are not directly relevant to representations and so are not presented.

Our model representation (section 4), Crepe, is based on a structured linear genotype. The structure of the genotype relates to three specific aspects of metamodels – meta-classes, meta-features, and the values that can be assigned to meta-features. The structure of the metamodel impacts on the properties of the representation. For instance, figure 7.1 shows two metamodels: one with features and one without. Two identical segments express different properties when mapped to objects with respect to different metamodels. The first segment (figure 7.1, left) has no redundancy as each feature pair maps to different meta-features. This compares to the second segment (figure 7.1, right), which exhibits redundancy due to the meta-class being instantiated not containing any features. Therefore, any mutations to the feature pairs would have no effect – resulting in non-locality.



**Figure 7.1:** A simple illustration of the effects that the structure of a metamodel has on the representation. The shaded boxes represent redundant feature pairs. The ovals containing numbers are the meta-element identifiers used by the genotype-to-phenotype transformation.

Therefore, the level of locality and amount of redundancy in Crepe is dependent on the particular metamodel being represented. Having information about the metamodel can aid decision making when configuring the representation for search. For example, if every meta-class in the metamodel has zero features, then there is no need to have feature pairs, as these would simply be junk and therefore waste computational effort and increase redundancy and locality. In a less extreme case, it may not make sense to have a high number of feature pairs when there are few

meta-features. Conversely, it would not make sense to have a low number of feature pairs when there are many meta-features. Furthermore, the *quantity* of meta-features may not be a useful guide to selecting the number of feature pairs. References have upper bounds – if each reference in the metamodel has an explicitly defined upper bound then it would be possible to select the maximum number of feature pairs in a segment. However, if many references define ‘many’ or ‘\*’ as the upper bound, this is not possible. The solution may simply be to choose a large number of feature pairs, accept redundancy as a consequence, and determine effective means to exploring the enlarged search space.

This section, therefore, attempts to understand how metamodels are commonly structured by automatically analysing a set of structural characteristics of a corpus of more than 500 publicly available metamodels. Analysing the structural characteristics of many metamodels illustrates the common practices that are made by metamodelers – the most used parts of the metamodeling language, and the ways in which domain concepts are typically expressed. This analysis will guide the evaluation of the representation by allowing us to select appropriate metamodels to investigate how the metamodel affects the representation, whilst providing insight into how best to make use of the representation.

There are, however, wider implications for this work. There has been much research into the quality of models, but there is little empirical analysis of metamodels. If we can analyse metamodels in the general case, we would be able to identify and detect good and bad practices, and understand the ways in which people are commonly structuring their metamodels today. Understanding structure is just a first step: once we can analyse metamodels in different contexts and for different purposes, we can identify patterns of metamodeling best practice for different contexts, and metamodel refactorings that facilitate model operations such as transformation. We can also develop understanding of how metamodels evolve over time, and seek to control the complexity of evolving metamodels to minimise the effects on model artefacts, operations and tools. We are currently producing a set of standard metrics and analyses for metamodels – similar to what exists in other domains (e.g. OO) – and developing a supporting automated metamodel measurement workbench [187].

A more thorough analysis can be found in [187] and at [www.jamesrobertwilliams.co.uk/mm-analysis](http://www.jamesrobertwilliams.co.uk/mm-analysis), including consideration of how metamodels evolve over time.

Section 7.2.1 introduces a set of metrics, focusing for now on structural analysis of metamodels. Section 7.2.2 presents the results of analysing the corpus of metamodels. Section 7.2.3 describes related research. Section 7.2.4 summarises the findings of the experiment, and section 7.2.5 selects four metamodels from the corpus with which to analyse our representation.

### 7.2.1 Structural Properties of Metamodels

We define a set of structural metrics that offer us insight into the common ways in which people develop metamodels. Arbitrarily selecting a set of metamodels to use for analysing the representation may not produce generalisable results, as the metamodels may not be representative of those that are being developed in practice. Analysing a corpus of existing metamodels allows us to, firstly, understand how practitioners are developing metamodels, and secondly, select representative metamodels to use in the evaluation of our representation or even guide the automatic generation of metamodels with representative characteristics.

The metrics considered in this section are all defined in EOL [92] because it provides an executable query language, akin to OCL, that can be executed on Ecore metamodels (see Section 7.2.2).

In general, metamodels consist of two sorts of elements: meta-classes, and the meta-features defined by those classes (attributes, and references to other meta-classes). The example metrics are grouped into metrics concerning meta-classes and metrics concerning meta-features.

**Metrics concerning Meta-classes** Our initial set of meta-class metrics, summarised in Table 7.1, focuses on the number occurrences of meta-classes with various properties in a metamodel. For instance, metric  $C$  gives an indication of the *size* of a metamodel, whilst  $C_c$  focuses on concrete meta-classes. Further metrics could explore the depth and shape of meta-class hierarchies in more detail.

Identifier	Description
$C$	Total number of classes
$C_c$	Total number of concrete classes
$C_F$	Total number of completely featureless classes
$C_{FA}$	Total number of completely featureless abstract classes
$C_{FC}$	Total number of completely featureless concrete classes
$C_I$	Total number of immediately featureless classes
$C_{IA}$	Total number of immediately featureless abstract classes
$C_{IC}$	Total number of immediately featureless concrete classes
$CF_{AVG}$	Average number of features per class
$CA_{AVG}$	Average number of attributes per class
$CR_{AVG}$	Average number of references per class

Metrics focusing on numbers of features gives us insight into the ways developers model domains. Here, we provide metrics on two kinds of featureless meta-class: *immediately* featureless classes are those that have no attributes or references, but may inherit features from a superclass ( $C_I$ ,  $C_{IA}$ ,  $C_{IC}$ ); *completely* featureless classes have absolutely no features ( $C_F$ ,  $C_{FA}$ ,  $C_{FC}$ ).  $C_I$  is a subset of  $C_F$ . Further metrics might explore the frequency of reference

**Table 7.1:** Structural metrics defined for meta-classes.

features, as compared to attribute features, or the distribution of features across hierarchies.

In addition to counting, we can create descriptive statistics such as means and medians. Table 7.1 lists three metrics for the average number of features per class ( $CF_{AVG}$ ,  $CA_{AVG}$ ,  $CR_{AVG}$ ). These metrics can be used to analyse whether there is a tendency to create many small classes, develop ‘God’ classes, or distribute features across classes.

**Metrics concerning Meta-features** The metrics concerning meta-features are summarised in Table 7.2. These metrics are global – they refer to the number of occurrences of features in an entire metamodel. These metrics illustrate how metamodellers commonly define the data (attributes) in metamodels ( $F_A$ ), and how they relate meta-classes to one another ( $F_{RC}$ ,  $F_{RN}$ ).

Identifier	Description
$F$	Total number of features
$F_A$	Total number of attributes
$F_R$	Total number of references
$F_{RC}$	Total number of containment references
$F_{RN}$	Total number of non-containment references
$F_{RU1}$	Proportion of references with upper bound set to 1
$F_{RU*}$	Proportion of references with upper bound set to many
$F_{RUN}$	Proportion of references with upper bound set to N

**Table 7.2:** *Structural metrics defined for meta-features.*

Table 7.2 also has examples of metrics that provide more detailed analysis of metamodelling characteristics, for example by analysing the upper multiplicity bounds of references ( $F_{RU1}$ ,  $F_{RU*}$ ,  $F_{RUN}$ ).

**Summary** The 19 metrics listed in this section are examples of what is possible, and incidentally, though unintentionally, overlap and extend the metrics defined in recent work by Ma et al. [104]. Listing 7.1 shows two of these metrics coded in EOL: they are not difficult to write. There are many more structural analyses that could be supported by similar metrics – for instance, the distribution of data types selected for attributes, the use of string attributes, enumerations, and custom datatypes. Whilst these additional metrics may not be relevant to our representation, they are necessary in the development of a set of standard metamodel metrics.

```

1 var numMetaClasses = EClass.all.size();
2 var numNonContainmentReferences =
3   EReference.all.select(r | not r.containment).size();

```

**Listing 7.1:** *Example EOL statements: metrics C and  $F_{RC}$*

To understand the characteristics of a metamodel, we can also combine metrics. For example, examining the total number of

meta-classes ( $C$ ) and total number of meta-features ( $F$ ) may highlight a metamodeller's preference for modelling – perhaps preferring many classes but few features, or few classes but many features. Furthermore, it can help us in defining the size of each individual used in a search problem for that metamodel.

### 7.2.2 Analysis: What do Metamodels Really Look Like?

This section uses the metrics defined in the previous section to analyse a large number of metamodels in an attempt to determine what the average metamodel looks like structurally. It should be noted that a metamodel that is dissimilar to the average may not necessarily be a 'bad' metamodel – though it may increase the difficulty in using it for a specific purpose. By computing the average metamodel, we hope to inform the community of how people are modelling domains and attempt to learn how to improve current practice. The analysis script and the corpus of metamodels are available online at: [www.jamesrobertwilliams.co.uk/mm-analysis](http://www.jamesrobertwilliams.co.uk/mm-analysis).

**Analysing the Corpus of Metamodels** We have accumulated a corpus of 537 publicly available Ecore [170] metamodels. The corpus is made up of metamodels collected from GitHub<sup>1</sup>, Google Code<sup>2</sup>, the AtlantEcore Zoo<sup>3</sup>, the EMFText Zoo<sup>4</sup>, and from internal projects. The corpus includes many well known modelling languages – such as the UML [122], DoDAF [174], and MARTE [119] – as well as metamodels for many programming languages such as Java, C#, C, and Pascal. Each metamodel was analysed using a script written in EOL to calculate each of the properties described in section 7.2.1. We then collated the scores and now describe the results. The complete analysis script is listed in appendix B.3.1.

<sup>1</sup> [github.com](http://github.com)

<sup>2</sup> [code.google.com](http://code.google.com)

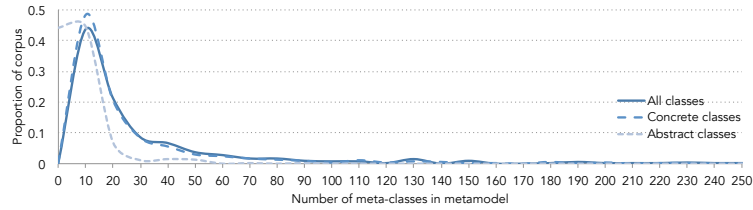
<sup>3</sup> [www.emn.fr/z-info/atlanmod/index.php/Ecore](http://www.emn.fr/z-info/atlanmod/index.php/Ecore)

<sup>4</sup> [www.emf-text.org/index.php/EMFText\\_Concrete\\_Syntax\\_Zoo](http://www.emf-text.org/index.php/EMFText_Concrete_Syntax_Zoo)

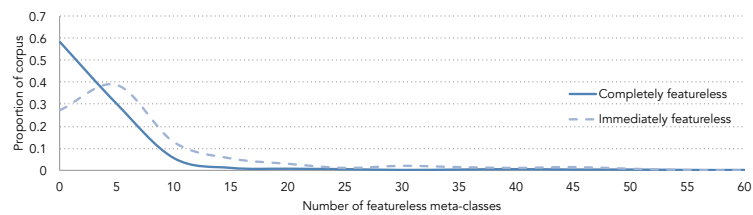
**Meta-class Metrics** Figure 7.2 shows the frequency distribution for the total number of meta-classes. The median total number of meta-classes in the corpus is 13, with a mean of 39.3, a maximum of 912, and a minimum of one. This suggests that metamodels (at least, in this corpus) are often fairly small. Twelve of the 537 metamodels have a single meta-class. Five of these metamodels are meaningless and should be removed, four were extensions of other metamodels, and three were domain-specific metamodels which also defined custom data types or enumeration types. Although small, a single-class metamodel can still define a suitable modelling language for some domains. The corpus showed that abstract meta-classes were not popular: 44% of metamodels did not contain a single meta-class denoted as being abstract. Furthermore, 96% of the corpus has fewer than 20 abstract meta-classes, whereas only 69% of the corpus has fewer

than 20 concrete meta-classes. This is arguably due to the small average size of the corpus: smaller metamodels are likely to contain only concrete classes, whereas larger metamodels are more likely to utilise abstract classes and inheritance hierarchies.

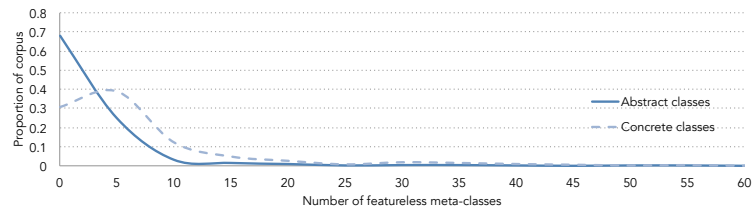
**Figure 7.2:** Frequency distribution of the total number of meta-classes ( $C$ ,  $C_c$ ,  $C_a$ ).



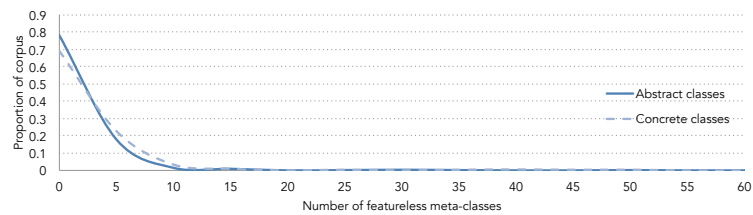
**Figure 7.3:** Frequency distribution of featureless meta-classes.



(a) Total number of featureless meta-classes ( $C_F$ ,  $C_I$ ).



(b) Frequency of immediately featureless meta-classes ( $C_{IA}$ ,  $C_{IC}$ ).

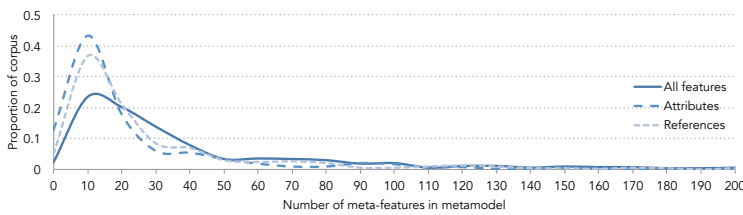


(c) Frequency of completely featureless meta-classes ( $C_{FA}$ ,  $C_{FC}$ ).

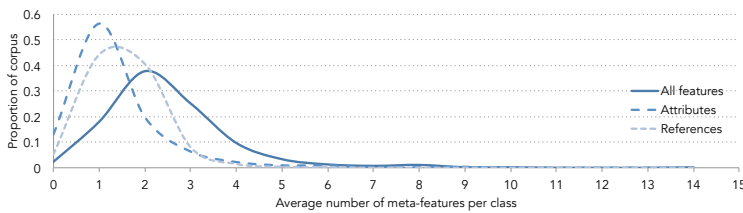
Featureless classes were uncommon. Figure 7.3a shows that 58% of the corpus has no completely featureless classes, and 27% have no immediately featureless classes. Interestingly, in the UML metamodel (developed by the Eclipse UML2 project<sup>5</sup>), 50 of the 227 meta-classes were immediately featureless, 40 of those were concrete. Immediately featureless classes are more common than completely featureless ones, and it is much more likely that these immediately featureless classes are concrete. Further analysis would likely show that these are specialisations of abstract classes (figure 7.3b), perhaps used to add extra semantics to the hierarchy.

<sup>5</sup> [www.eclipse.org/uml2](http://www.eclipse.org/uml2)

**Meta-feature Metrics** Figure 7.4a shows the frequency distribution of the total number of meta-features contained in the meta-model corpus. The median number of meta-features per meta-model is 23.5, with a mean of 69.2, a maximum of 2410, and a minimum of zero. Furthermore, metamodels commonly have more references (median 13.5, mean 43.0) than attributes (median 8, mean 26.2). Figure 7.4b shows that the average metaclass has 2.1 features: 1.15 references and 0.95 attributes. The large number of featureless classes present in the corpus (figure 7.3) affects this data. If we exclude featureless classes when calculating the average features per class, we obtain the same distributions, however the mean number of features per meta-class increases slightly to 2.3, with 1.3 references and 1.0 attributes.



(a) Total number of meta-features ( $F$ ,  $F_A$ ,  $F_R$ ).



(b) Average number of features per class ( $CF_{AVG}$ ,  $CA_{AVG}$ ,  $CR_{AVG}$ ).

Metrics  $F_{RC}$  and  $F_{RN}$  showed that on average metamodels contain more non-containment references (median 6, mean 27.3) than containment references (median 5, mean 15.7). With respect to metrics  $F_{RU1}$ ,  $F_{RU*}$ , and  $F_{RUN}$ , the upper bounds of references are set to 'one' 52% of the time, to 'many' 47% of the time, and are explicitly given a value just 1% of the time. The trend towards selecting 'many' as opposed to explicitly defining an upper bound might be attributed to the inherent uncertainty in modelling [192, 179] (of course, sometimes specifying an upper bound as 'many' is perfectly acceptable and not related to domain uncertainty).

### 7.2.3 Related Work

**Note:** This section draws extensively on Zolotas and Matragkas' contribution to the background section in [187], and included for completeness.

While there is a significant amount of work in the field of analysis of MDE artefacts, the majority of the related work we have encountered has a different focus to this paper. While this paper provides an empirical study on a large corpus of metamodels, the majority of existing work proposes either new metrics or new

**Figure 7.4:** Analysis of meta-features.

approaches for measurement of models. Another characteristic, which makes this paper novel is the size of the corpus, which gives more validity to the obtained results.

The closest work to ours is Cadavid et al. [20] who present an empirical analysis of the ways MOF and OCL are used together. They define metrics to analyse the complexity of 33 metamodels, their constraints, and the coupling between the two. The work in this paper aims to complement Cadavid et al's work with deeper analysis of the metamodel structure (as opposed to its relationship with its constraints). Additionally, Vepa et al. [175] measure a set of metamodels that is stored in the Generative Modeling Technology/ATLAS MegaModel Management (GMT/AM3) Repository. This work focuses on the model repository and the measuring technique, rather than the presentation of the results of the analysis.

O'Keeffe and Ó Cinnéidie [124] define five object-oriented metrics that are used to evaluate refactorings to class diagrams. These metrics partially overlap with our metrics. In particular, they define a featureless class metric – aiming to minimise this “since they have no responsibilities and increase design complexity” [124]. They also define metrics related to methods, which should be added to our metric collection.

Ma et al. [104] propose a model to assess the quality of metamodels based on five properties (reusability, understandability, functionality, extendibility, well-structuredness). In their study they measure the quality of four well-known metamodels (UML, SysML, BPMN and MOF). Kargl et al. [81] propose a metric that compares the concepts that appear in the abstract syntax of modelling languages with the concepts that are used in the concrete syntax, and apply this to two versions of UML. Finally, Arendt et al. [4] describe an Eclipse plugin, *EMF Metrics*, that can be used to assess the quality of EMF metamodels based on nine quantitative criteria. The aforementioned approaches focus mainly on model quality (as with software), while we (and Cadavid) are interested in understanding the usage of metamodelling languages and how metamodels are built.

#### **7.2.4 Summary of Experiment**

In this section we have posited the need for a deeper understanding of metamodels, not only to provide guidance for tailoring Crepe to specific problems, but also to understand, analyse, and inform the practices of metamodellers. We illustrate structural analysis on a corpus of over 500 Ecore metamodels, gaining insight into how metamodels are commonly structured. We are now in a position to start the analysis of good and bad practice in metamodelling, for general purpose modelling languages or domain-specific modelling languages, and in different model management contexts.



To facilitate development of further metrics, we are creating a metrics metamodel. We plan to create a web-based automated metamodel measurement workbench that allows users to upload and analyse their own metamodels, which will automatically augment to the results given here. We plan to devise a comprehensive set of metrics, and develop state-of-the-art analyses for metamodels, taking inspiration from similar domains, such as bad smell detection [48] and design patterns [27].

### 7.2.5 Selecting Representative Metamodels

From the results of analysing the corpus, we have gained insight into the most common ways in which metamodels are structured. By gaining this knowledge, we are able to determine which structural characteristics have the most effect on different properties. We can then either guide users towards refinements to their metamodels that improve evolvability, or use the knowledge to refine the representation itself, so that it is tuned towards the most common metamodel structures. We have selected four metamodels with which to use when evaluating properties of our representation, and to act as benchmarks for other model representations. We refer to these metamodels as *MUTs*: metamodels under test. The detailed analysis of the MUTs are shown in Table 7.3. Each MUT has approximately the median number of meta-classes, but varying numbers of features. We keep the number of meta-classes (almost) constant as it doesn't affect the representation dramatically. The more meta-classes, the larger an individual would need to be to ensure that each meta-class is fairly represented, making analysis more costly. Moreover, properties of meta-features will have more variable effects than meta-classes.

Name	C	C <sub>A</sub>	C <sub>C</sub>	C <sub>F</sub>	C <sub>I</sub>	F	F <sub>A</sub>	F <sub>R</sub>
EG-MOF	13	0	13	0	11	6	0	6
Java	13	3	10	0	2	30	12	18
ATOM	14	0	14	0	4	60	38	22
DslModel	14	3	11	1	1	23	8	15
Corpus median	13	2	11	0	2	23.5	8	13.5

**Table 7.3:** The metric scores for four metamodels, and the median scores of the corpus.

Each of the selected MUTs are now described, and are shown graphically in appendix B.4.

*EG-MOF* This metamodel comes from the AtlantEcore Zoo and defines the concepts for drawing graphs of program execution. An ExecutionGraph contains a collection of *Nodes*. Each node has a set of predecessor nodes and a set of successor node. There are 11 specialisations of the *Node* class, including Start, Branch, and Fork. This metamodel represents those who have fewer than the median number of features.

*Java* This metamodel was found on GitHub and defines a basic

metamodel for representing Java programs. A Model contains many *Elements* – Packages, Classes, Enumerations etc. Classes contain Methods and Fields. Methods have MethodParameters and Statements. This metamodel represents those who have more than the median number of features.

*ATOM* This metamodel, taken from the AtlantEcore Zoo, depicts the *Atom Syndication Format* [60] – a web standard for content syndication, and an alternative to RSS. Each Atom feed has collections of Links, Entries. Entries contain more information related to the content being syndicated. This metamodel represents those who have many more than the median number of features.

*DslModel* Also taken from the AtlantEcore Zoo, this metamodel captures the concepts used to define domain specific languages (DSL). In essence it defines a basic meta-metamodelling language. Models contain *Elements* such as ModelElement and ReferenceLinks between model elements. This metamodel represents the median metamodel.

Now that we have selected our MUTs we can begin to analyse the properties of our representation, starting with locality.

## 7.3 *Locality*

As previously discussed, locality is necessary for mutation-based search [147, 150]. The structured nature of our representation means that different types of gene in an individual encode different parts of the model being represented. Mutating these different types will have varying degrees of impact on the represented model – illustrated in figure 4.14 – a phenomenon known as *positional dependence* [25]. Therefore, in analysing the locality, we are interested not just in the general effect of mutation, but also in the effects of mutating the different types of genes. It seems obvious that mutating class bits will have the most impact on the encoded model – if we can show this, we might wish to set the mutation probability for class bits to a much lower value than those for the feature-related bits.

A result of positional dependence is that the locality of our representation is tightly integrated with the metamodel that defines the model space being explored. Analysing the locality of a set of metamodels with different characteristics can aid users in configuring their metamodel for use with search: a small set of simple refactorings to a metamodel with non-locality may result in locality being exhibited. Manually creating metamodels with certain characteristics may, however, result in contrived metamodels that are not representative of the kinds of metamodels that people develop in practice. As such, in this section we analyse the locality

of the four metamodels (referred to here as MUTs) selected in the previous section.

The set of models that can conform to a metamodel is potentially unbounded. With respect to the representation, adding an extra segment or feature pair to an individual increases the number of models that can be represented by that size of individual. If a metamodel defines upper bounds for all features, then there is a maximum size that models can take and any new segments added to an individual will be redundant. However, if the metamodel does not define upper bounds, the model space is infinite and new segments can be added to an individual without introducing redundancy. Even though a model space may be infinite in size, it is unlikely that there is an upper limit on the size of the models used in the particular domain – especially in the cases of domain-specific metamodels. Therefore, although adding segments will increase the size of the phenotypic space, it may introduce models that are not representative of the domain and thus increase the difficulty of the search problem.

One consequence of positional dependence and infinite state spaces is that full enumeration of the model space is not feasible. Therefore in order to calculate the locality of our representation with respect to each MUT, we sample the model space and in particular, we sample each type of gene.

### 7.3.1 *Experimental Setup*

**Objective** In this experiment, we investigate the locality of the representation, with a focus on determining the locality with respect to the different types of gene used in our representation, and attempt to understand how characteristics of the MUT affect locality.

**Approach** To analyse locality, we randomly generate a set of models conforming to each MUT and mutate samples of the class bits, feature selector bits, and feature value bits.

For each of the four MUTs, we created appropriate finitisation models – 100 random strings and 100 integers were used to finitise any string or numeric attributes present in the metamodels. Table 7.4 lists the parameters used in this experiment. For each metamodel we generate 50 random individuals using the minimum and maximum sizes specified in Table 7.4. For each generated individual, we randomly select and mutate 200 class bits, 500 feature selector bits, and 500 feature value bits. This means that for each model we analyse 1,200 mutants, and for each metamodel we analyse 60,000.

To allow for parallel execution, the experiments were split into groups of ten models, each executed with a different seed to the pseudo-random number generator.

Locality is defined using distance metrics on the genotype and

**Table 7.4:** *The parameters used to analyse locality with respect to a single meta-model.*

Parameter	Setting
Number of models	50
Class bit sample size	200
Feature selector bit sample size	500
Feature value bit sample size	500
Segment size (min-max)	10-50
Feature pair size (min-max)	12-30
Maximum allele value	50

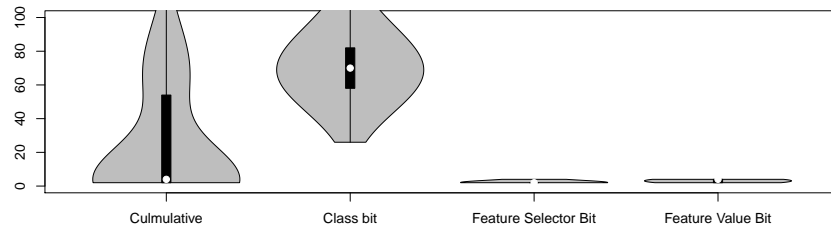
phenotype. For our experiments, we define our distance metrics as follows. A genotypic distance of one is defined as being a single gene difference: i.e. one application of the mutate operator. This value of the differing gene can be anything within the range of zero to the maximum specified. Phenotypic distance is calculated using *EMF Compare*<sup>6</sup> (as was done in section 3.4.1), a framework for comparing EMF models. An alternative would be to define our own matching algorithm, allowing us to have fine-grained control over the matching. In [180] we propose extensions to an existing model comparison language, ECL [90]. These extensions include: the ability to perform partial matching, restricting matching rule application, specifying match multiplicity, and defining custom matching strategies. Having these extensions would allow us to have total control over the matching: for example, enabling us to define different weights for different types of matching elements. Writing a robust matching algorithm, however, offers many challenges and so for the locality analysis the differences computed by EMF Compare are satisfactory and the extensions to ECL are left as future work.

<sup>6</sup> <http://www.eclipse.org/emf/compare/>

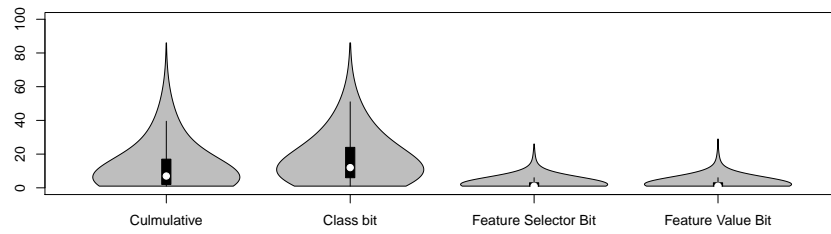
### 7.3.2 Results

Table 7.5 shows a summary of the results of the experiment. As well as listing the locality scores and distances, we also show the percentage of neutral mutations: those that resulted in a phenotypic distance of zero. Figure 7.5 shows the results presented as violin plots. Violin plots combine box plots and a kernel density plot to give a more detailed visualisation of the distribution. Surprisingly, given its simplicity, the MUT that exhibits the most non-locality is *EG-MOF*. Furthermore, the metamodel that exhibits the most locality is *ATOM*, the metamodel with a well above average feature load. *DslModel* and *Java* score similar results and lie in between *EG-MOF* and *ATOM*.

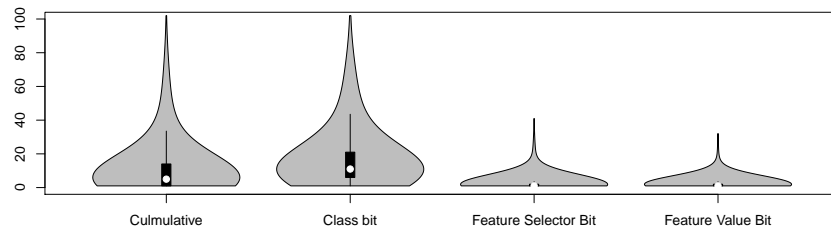
The reason for this unexpected behaviour is that the selection of ranges used to define the individuals' lengths for the experiment is not representative of all of the MUTs. *EG-MOF* exhibits a significant number of differences when a single class bit is mutated. By inspecting the generated models, the reason for these large distances was found to be due to the reference-heavy na-



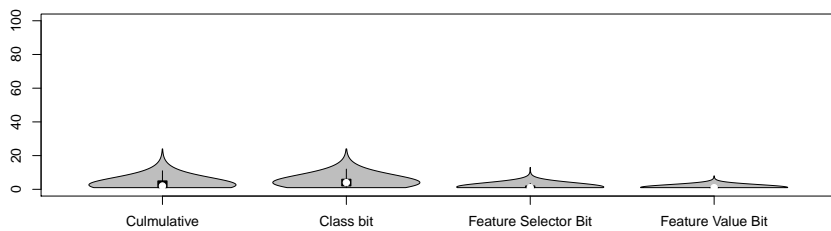
(a) *EG-MOF*



(b) *DslModel*



(c) *Java*



(d) *ATOM*

**Figure 7.5:** Violin plots representing the distribution of distances caused by mutation. The white dot in the box plots represents the median.

**Table 7.5:** The results of analysing the locality of the representation against the four MUTs. CB = class bit, FSB = feature selector bit, FVB = feature value bit.

	EG-MOF	DslModel	Java	ATOM
Overall Locality score	675636	126367	154628	31114
CB locality score	639929	120676	146769	30147
FSB locality score	14001	3198	4974	799
FVB locality score	21706	2493	2885	168
CB mean distance	70.7	16.9	17.4	4.8
CB median distance	70	12	11	4
FSB mean distance	2.3	2.4	2.3	1.3
FSB median distance	2	1	1	1
FVB mean distance	3.0	2.5	2.1	1.1
FVB median distance	4	1	1	1
Overall neutrality	48.6%	80.7%	74.0%	78.9%
CB neutrality	8.2%	24.0%	10.7%	20.2%
FSB neutrality	55.9%	90.9%	84.1%	89.8%
FVB neutrality	57.4%	93.1%	89.2%	91.6%

ture of the metamodel combined with its inheritance hierarchy. The hierarchy of the metamodel means that mutating an object's metaclass likely results in a new object whose meta-class has the same superclass. For example, one might mutate a Branch object into a Fork object. As all references in the metamodel are defined with respect to the super type (Node), any references that were pointing at the original object are instead pointing at the new object. The feature pair size range selected for the experiment had the unfortunate effect of creating a large number of references from each object in the generated models. This meant that the each object referenced 10 or 15 other objects, creating a highly connected graph. Therefore, a change to a single object would affect a huge portion of the model, and thus increase the number of differences computed by EMF Compare. More importantly, these highly connected models are not representative of the domain. *EG-MOF* defines a language to express the execution graphs of a program. Although program execution graphs can be complex, it is unlikely that they exhibit the complexity found in the generated models.

This has highlighted the importance of selecting appropriate sizes of individuals. Selecting inappropriate sizes can have a devastating impact on the locality of the representation, which can lessen the effects of evolutionary search. To address this, in the next section we redesign the experiment to use metamodel-specific sizes of individuals in the aim of analysing the locality of more realistic models.

### 7.3.3 Experimental Redesign

To ensure that the individuals analysed in the experiment are representative of the metamodel, we manually adjusted the range values and manually inspected the generated models. Table 7.6 shows the ranges selected for this. *EG-MOF* has much lower and shorter ranges than in the previous experiment, and the upper limit of the number of segments for *ATOM* has increased. *DslModel* and *Java* are similar to the original values, with a small decrease of the lower limit of the number of feature pairs, a reduction on the upper limit of the number of feature pairs for *DslModel*, and a reduction on the upper limit of the number of segments for *Java*.

Metamodel	Segment Range	FP Range	Max. Allele
EG-MOF	4–20	4–20	20
DslModel	10–50	8–20	50
Java	10–30	8–30	30
ATOM	10–80	12–30	80

**Table 7.6:** The metamodel-specific ranges used when generating individuals for locality analysis.

Note that we were not the authors of these metamodels, and have made assumptions that the models produced using these ranges are representative. Further analysis should more accurately determine what a representative model would look like for each metamodel, and use that to seed the experiment.

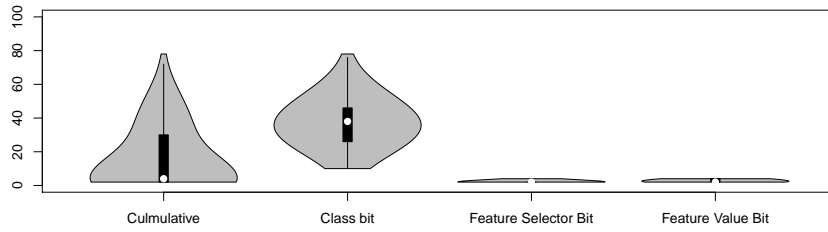
The parameters regarding the experimental design (table 7.4) are kept the same.

### 7.3.4 Redesign Results

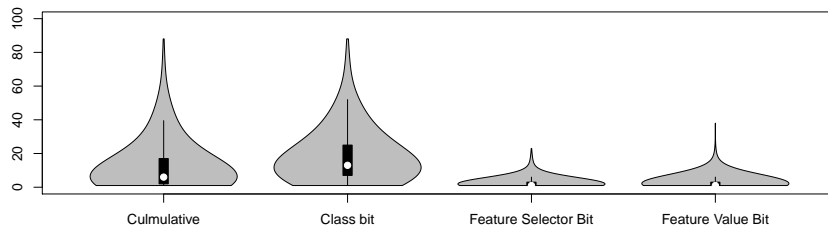
Table 7.7 shows a summary of the results of the redesigned experiment, and figure 7.6 visualises the results as violin plots. The effect of adjusting the defining ranges of the individuals is apparent, although not as much as expected. The average distance caused by mutating the class bit has almost halved for *EG-MOF*, and increased slightly for *ATOM*.

It is obvious from all sets of results that mutating the class bit is much more destructive than mutating either of the feature bits. The metamodels are ordered in Table 7.7 by increasing number of features in the metamodel. Unexpectedly, there appears to be an increase in locality as the number of metamodel features increases. The average phenotypic distance introduced by mutating the class bit decreases from left to right. *EG-MOF* scores the worst for locality, but also has the lowest percentage of neutrality (which will increase the locality score). Feature selector bits and feature value bits have approximately the same effect regardless of the metamodel, although there is still a decrease present in the average distance as the number of metamodel features increases.

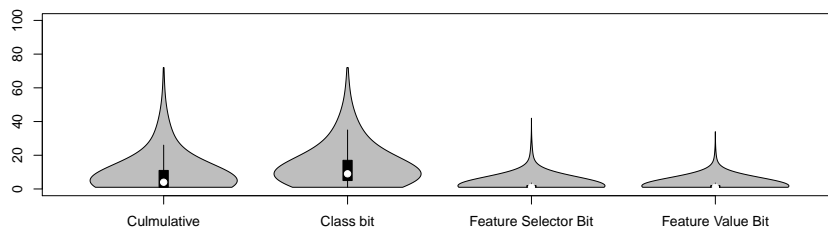
We now take a closer look at the results for each MUT in order to understand how their different characteristics affect locality.



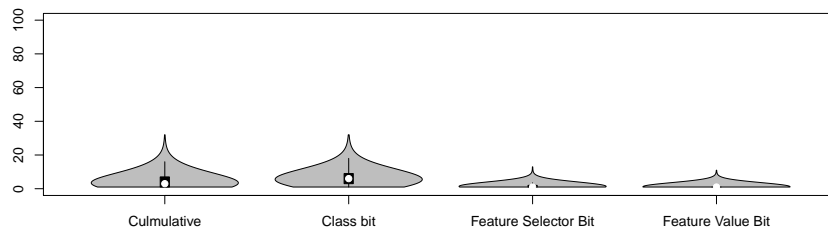
(a) *EG-MOF*



(b) *DslModel*



(c) *Java*



(d) *ATOM*

**Figure 7.6:** Violin plots for the redesigned experiment. The white dot in the box plots represents the median.



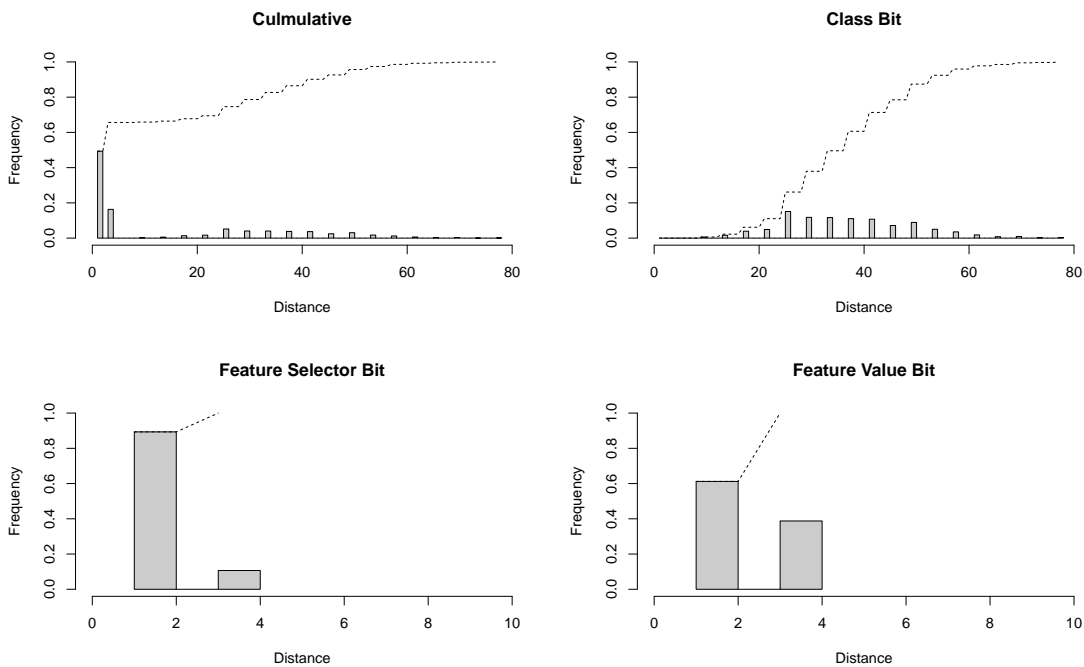
	EG-MOF	DslModel	Java	ATOM
Overall Locality score	355671	142596	118504	25717
CB locality score	329751	136062	110645	24844
FSB locality score	10402	3577	4919	704
FVB locality score	15518	2957	2940	169
CB mean distance	37.4	18.0	13.5	6.6
CB median distance	38	13	9	6
FSB mean distance	2.2	2.3	2.2	1.3
FSB median distance	2	1	1	1
FVB mean distance	2.8	2.4	2.0	1.1
FVB median distance	2	1	1	1
Overall neutrality	56.1%	78.6%	73.8%	79.3%
CB neutrality	9.4%	20.1%	11.6%	19.2%
FSB neutrality	65.7%	88.9%	83.6%	90.4%
FVB neutrality	65.0%	91.6%	88.8%	92.3%

**Table 7.7:** The results of analysing the locality of the representation against the four MUTs. CB = class bit, FSB = feature selector bit, FVB = feature value bit.

**EG-MOF** This metamodel represents those that have below average number of features, having just six references and no attributes (as determined by the analysis in the previous section). The number of differences caused by mutating the class bit has decreased by nearly half as compared to the original experiment, however the number is still extremely high. Again, this is due to the metamodel enabling a model to become highly connected. Reducing the feature pair range further would again bring down the scores, but may end up produce unrealistic models. Further work is needed in understanding how models conforming to *EG-MOF* commonly look, or the metamodel may need to be constrained to restrict unrealistic models from being produced. This information can then guide the selection of ranges.

Figure 7.7 shows a more detailed view of the distributions of the phenotypic distances affected by the experimentation. The class bit frequency plot (upper right) exhibits an unusual step function, with step distance of four – likely due to the structure of the references in the metamodel. Mutating the feature bits has a much less dramatic effect on the model, however *EG-MOF* scores higher on average than the other metamodels regarding feature bits. This is due to the fact that *EG-MOF* has only references, and no attributes. In the general case, mutating the value of an (single-valued) attribute would only cause one structural difference (though perhaps may result in a vastly different model semantically). Mutating a reference value, however, can cause more than one difference. For instance, if the target of a reference from object A to object B is moved to object C and that reference has an opposite reference<sup>7</sup> defined, the opposite reference (B to A) is also unset, and a new reference from C to A is created, therefore resulting in more than one difference. Moreover, if the reference is a

<sup>7</sup> An ‘opposite’ reference is the inverse of a bidirectional reference. Ecore automatically updates/sets opposite references when creating new or modifying existing references.



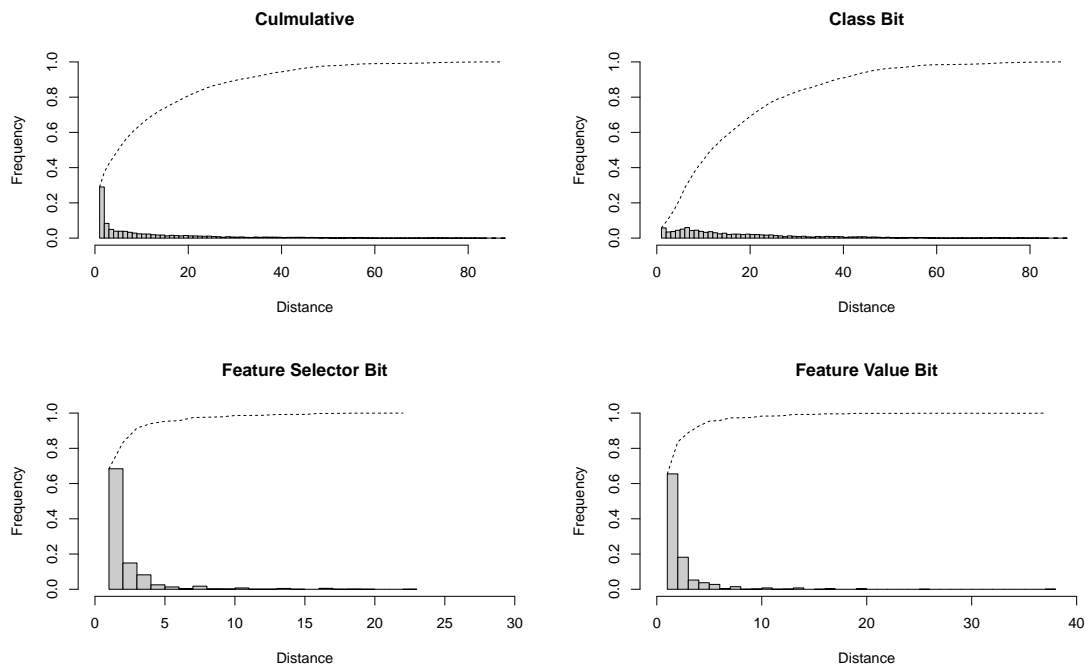
**Figure 7.7:** EG-MOF locality frequency distributions. The dotted line is the cumulative frequency.

containment reference, it will affect the object(s) being contained, again resulting in more than one difference.

**DslModel** This is the metamodel whose characteristics most closely resemble the average (median) metamodel found in the corpus: it is this metamodel whose locality results may be most informative for future case studies. The plots in figure 7.8 emphasise that mutations to feature bits result in high locality, whereas class bits are non-local – with a fairly uniform distribution of distances between one and 40. This highlights the destructive power of the class bit – changing the type of an object can have a huge knock-on effect to other areas of the model and result in a totally distinct model.

Reducing the number of feature pairs had little effect on the feature pair locality, however there is a slight increase in the non-locality caused by the class bit. Neutrality decreased slightly in the second experiment which may account for this increase.

**Java** This is the MUT that represents metamodels with an above average number of features, with respect to the analysed corpus. The results (shown in figure 7.9) are very similar to those of *DslModel* but exhibit slightly reduced effects of mutation. Hierarchically, these two metamodels are similar which accounts for the similarities. The amended segment range produced a reduction in the number of differences caused by class bit mutation. This is a result of there being fewer objects in the generated mod-



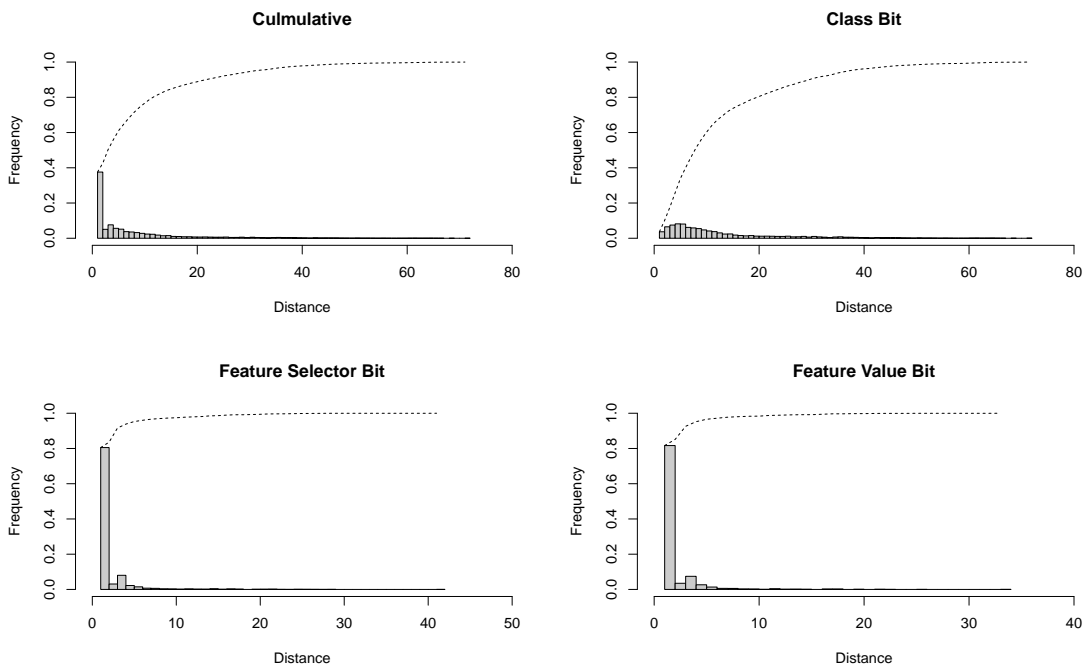
**Figure 7.8:** *DslModel* locality frequency distributions. The dotted line is the cumulative frequency.

els. Models conforming to *Java* are obviously well connected and so class mutation will have a big impact.

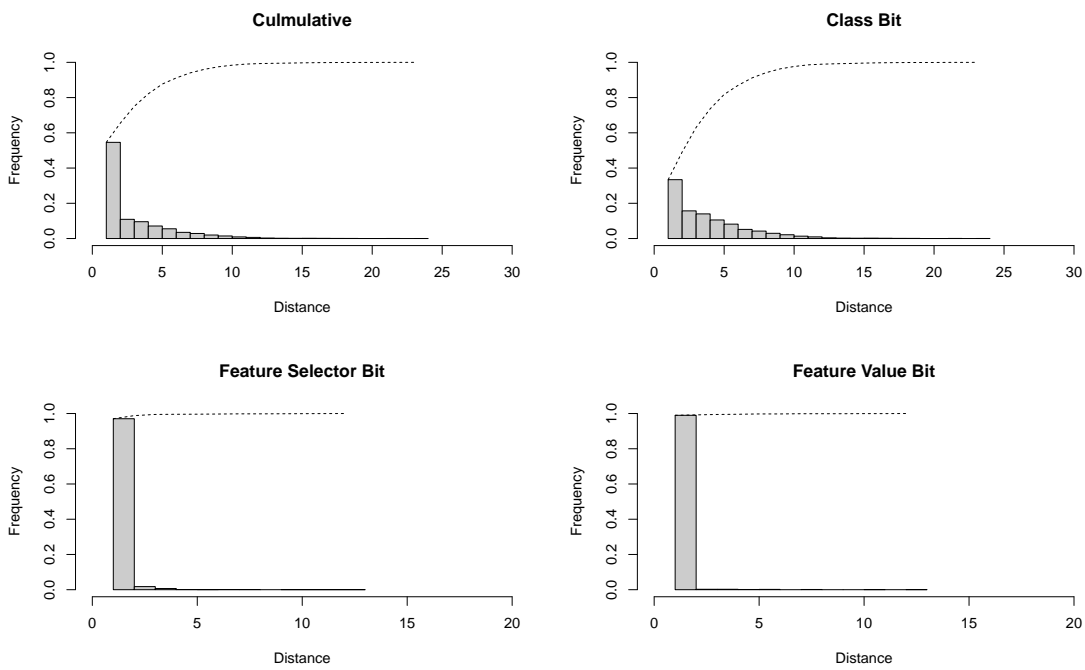
**ATOM** This metamodel was a very feature-heavy metamodel. Surprisingly, it was the most resistant to mutation in both experiments, as illustrated by the results in figure 7.10. By increasing the number of segments, the effects of class bit mutation did increase slightly, but the median distance is still more than half of the closest metamodel (*Java*). The reason behind this is the fact that *ATOM* has a large number of featureless, or feature light, meta-classes. Mutating objects conforming to these classes will have a smaller effect than mutating objects conforming to the few meta-classes with a heavy load. Were the number of segments increased, and the class bit sample size also increase, the effect of mutating these classes would like become more apparent.

### 7.3.5 Discussion

In this section we have shed light on the locality of our representation, by analysing the effects of small mutations to sets of models conforming to the MUTs. We showed how the choice of the defining ranges of an individual's length is vital for ensuring locality. The inheritance hierarchy of the MUT, combined with the distribution and classification of its features, heavily influences the locality of the representation. The results demonstrated that mutating feature bits causes, on average, a small number of changes to the phenotype. In contrast, mutating a single class bit



**Figure 7.9:** Java locality frequency distributions. The dotted line is the culmulative frequency.



**Figure 7.10:** ATOM locality frequency distributions.

can cause a dramatic change to the phenotype. Locality has been shown to produce efficient evolutionary search, therefore, when using our representation it may be appropriate to set the mutation probability of the class bit to a very low value. A user of our representation can analyse their metamodel's locality before applying the metaheuristic in order to gain insight for selecting mutation rates and defining optimal lengths of individual used in the search. Finally, one needs to consider the problem at hand. Although a representation could be shown to have locality, this may not be beneficial to the metaheuristic. Rothlauf demonstrates how non-locality can *reduce* the difficulty of a problem which has a *misleading* fitness landscape (where fitness increases the away from the global optimum) [147].

Of course, we must be careful with our conclusions - there are many other characteristics of metamodels that may influence locality, and should be addressed in future work.

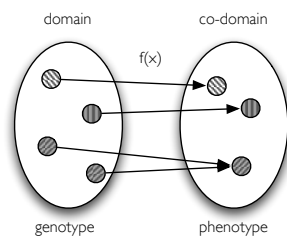
## 7.4 Redundancy

During the transformation from genotype to phenotype, parts of the genotypic individual – i.e. single feature pairs and entire segments – may have no impact on the resulting phenotypic individual. Any part of a genotypic individual that does not appear in its associated phenotypic individual is said to be *non-coding* or *inactive*. Even with non-coding parts, it is possible that different genotypic individuals can encode the same phenotypic individual. These two phenomenon result in *redundancy* in the representation.

It is argued that too much redundancy creates an inefficient representation, which cannot encode as much information as its potential [147]. Some argue that redundancy is beneficial, as mutations to inactive parts of the genotype will *activate* new areas of the solution space [147, 111], whilst increasing the frequency of optimal solutions [82]. For example, changing the value of a class bit may activate a previously redundant feature pair in the production of that model object. This section examines the areas of the representation where redundancy manifests, with a focus on non-coding redundancy. As with locality, we show that the characteristics of the MUT play an important role in determining the amount of redundancy in the representation of its model space. Section 7.4.1 describes how redundancy can manifest in our representation, classifying redundancy into two categories. Section 7.4.2 describes an experiment to analyse the non-coding redundancy of the representation with respect to the four MUTs selected in section 7.2. The results of this experiment are presented in section 7.4.3 and the amount of non-coding redundancy is related to the structural characteristics of the MUTs.

### 7.4.1 Redundancy in the Representation

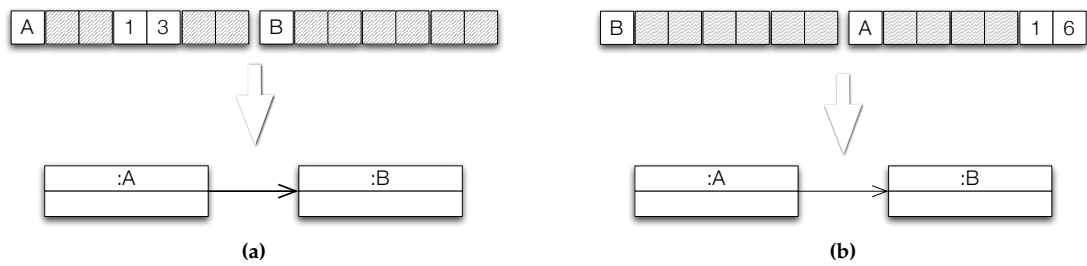
Rothlauf's definition of redundancy is simply an inefficient representation – one where there are more genotypes than phenotypes [147]. Rothlauf uses the terms *synonymously redundant* to express representations where the set of genotypes that encode the same phenotype are similar to one another, and *non-synonymously redundant* to mean the opposite [147]. By those definitions, the neutrality that we exposed in the previous section (see Table 7.5) was synonymous redundancy: these were small mutations that causes a phenotypic distance of zero. In this section, we classify redundancy into two subtly different categories: *surjective redundancy* and *non-coding redundancy*. We use the term *surjective redundancy* to stand for cases where more than one genotype represents the same phenotype (which therefore encompasses both synonymous and non-synonymous redundancy). Non-coding redundancy is lower level: the surjection of the phenotype is not of interest, but instead this relates to determining which specific parts of the genotype do not play a role in defining the phenotype.



**Figure 7.11:** A surjective function.

**Surjective Redundancy** A *surjective function* is one where every element in the codomain is mapped to by at least one element in the domain (see figure 7.11). Surjective redundancy refers to situations where different genotypes encode the same phenotype. In our representation, two genotypes can map to the same phenotype in a number of ways. Firstly, surjective redundancy can be introduced by the value selected to be the maximum allele allowed. A larger maximum value increases the size of the search space and, due to the fact that the modulus is used to select meta-elements and values, two different numbers can represent the same phenotypic element(s). Goldberg [56] argues that the alphabet (i.e. the range of allele values) should be as small as possible. Secondly, the ordering of segments and feature pairs impacts on surjective redundancy. If two segments encode objects that are in no way related in the phenotype, their order can be swapped in the genotype without affecting the phenotype. Figure 7.12 illustrates these two points. The two simple individuals shown have the same segments specified in opposite orders and yet result in the same phenotype. Furthermore, the two feature value bits encode the same reference assignment due to the transformation phase taking the modulus of those values when selecting the target object.

According to Rothlauf's definitions, the example shown in figure 7.12 would be considered to be non-synonymously redundant. This is because a single application of the mutation operator, which defines the neighbourhood of an individual, to the first individual would not result in the second.



**Figure 7.12:** Illustration of surjective redundancy. The ordering of the segments has no effect on the encoded model. The two unshaded feature pairs assign the same reference (from object A to object B) even though their feature value bit differs.

**Non-Coding Redundancy** During the reference construction phase of the transformation, we may be left with a number of objects that are not contained by the root object. To partially address this issue, we force the root object to contain every object it possibly can before we assign the unhandled references. The reference construction phase is allowed to then break an object’s direct containment with the root. Once the reference handling phase has completed, there may remain a number of objects that are not directly or indirectly contained by the root object. These are called *islands* and are automatically removed from the model. The segments that produced the objects in the islands are therefore non-coding segments, and introduce redundancy into the representation.

Where a feature pair relates to a multi-valued feature, it can become non-coding if no referenceable objects have been instantiated, or if the reference has reached its upper bound. With respect to single-valued features, the genotype-to-phenotype transformation will always assign the value specified by the feature value bit to a single-valued features. This may overwrite the current value of the feature, resulting in the feature pair that previously assigned a value to the feature becoming non-coding. This is independent of whether the feature is an attribute or a reference.

The locality experiment carried out in the previous section gave insights into surjective redundancy by recording the percentage of neutral mutations. In the next section we perform an experiment to analyse non-coding redundancy. We aim to understand the source of neutral mutations by analysing non-coding redundancy of the representation. We leave detailed analysis of surjective redundancy as future work and discuss the challenges of this analysis in section 7.4.4.

### 7.4.2 Experimental Setup: Non-Coding Redundancy

**Objective** In this experiment, we perform an investigation into the amount of non-coding redundancy in our representation, and discover how different properties of the MUT affect this form of redundancy.

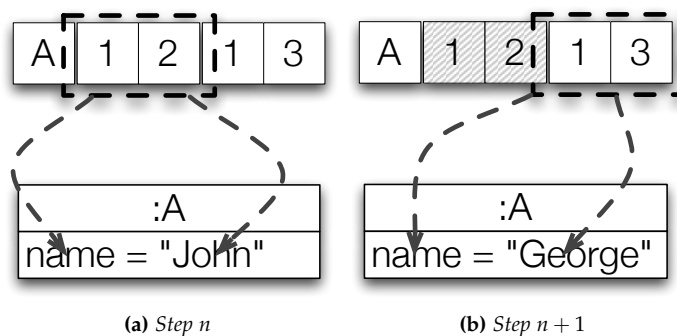
**Approach** To analyse non-coding redundancy, we shall generate a large number of models for each MUT and calculate the

proportion of non-coding segments and feature pairs.

Determining the number of coding segments in an individual is straightforward. Each segment represents one object, therefore to determine how many segments were non-coding, we can simply subtract the number of objects in the phenotype (excluding the root object) from the number of segments in the genotype. Alternatively, we could map the resulting model back down to its canonic genotypic form and count the number of segments.

Calculating the number of non-coding feature pairs, however, is non-trivial. The feature selector bit can represent attributes or references, both of which may be single-valued or multi-valued. When a feature selector bit references a single-valued feature previously set, its associated feature value bit will overwrite the feature's value – as illustrated in figure 7.13. When a feature selector bit references a multi-valued features, its associated feature value bit will select an element to add to the feature's set of values, providing that the feature's upper bound has not been reached. Unfortunately, subtracting the number of feature assignments in the phenotype from the number of feature pairs in the genotype does not accurately give us a count of the number of coding feature pairs. This is because assigning a value to one feature can cause other features in the model to be updated. For instance, in the case of opposite references, setting one reference in an object will update the target object's opposite reference. Furthermore, any attributes that have default values specified in the metamodel which were not modified by a feature pair will add false positives to the count.

**Figure 7.13:** Illustration of overriding a single-valued feature, resulting in a non-coding feature pair. The dashed box represents the feature pair currently being decoded. The shaded boxes represent redundant feature pairs.



This is where using a model to represent the genotype proves beneficial. To support non-coding analysis of both segments and feature pairs, we extended the search metamodel (figure 4.18) to include two new boolean attributes: one in the Segment meta-class, and one in the FeaturePair meta-class. These flags are updated when the genotype-phenotype transformation recognises that the associated segment or feature pair has been made redundant. Once an individual has been mapped to its phenotype, we can inspect it and count the number of non-coding segment and feature pair flags that are set to true. Adding support for this to the transformation involved the addition of approximately 20



lines of EOL and two map data structures – required to keep track of which feature pairs and segments have been assigned to which model elements. The extra computation and memory required is not substantial but should not be included in the general case.

To analyse the amount of non-coding redundancy in our representation, and to understand the influence of the structural characteristics of the metamodel, we perform a three step experiment for each MUT:

1. Generate 1000 random individuals conforming to the MUT.
2. Transform each individual to its phenotypic form.
3. Inspect the individual and log the proportion of redundant segments and feature pairs.

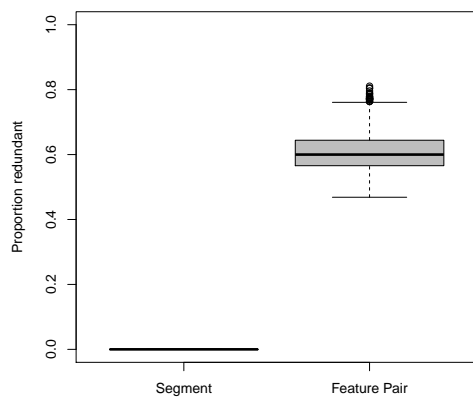
For consistency, the segment and feature pair ranges, and the maximum allele value, used to create the random individuals were reused from the locality experiment (table 7.4).

### 7.4.3 Results

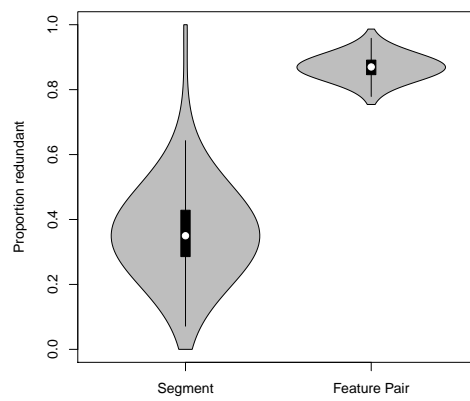
Figure 7.14 presents violin plots showing the proportion of non-coding segments and feature pairs for each of the MUTs. Incidentally, the proportion of non-coding redundancy found in these experiments matches the proportion of neutrality displayed in the locality experiments from the previous section. We address each MUT in turn.

**EG-MOF** This MUT has the lowest amount of feature pair redundancy out of the four. Non-coding feature pairs are likely introduced here due to two of the references of the `Node` class not being able to be assigned values. One of these references points up to the container object `ExecutionGraph` – the root of the model, and therefore not expressed in the genotype. The other reference points to `ExecutionGraph` objects that are contained in separate models – i.e. it is a cross-model reference. The inheritance hierarchy of the metamodel results in there being zero non-coding segments – every instantiable meta-class is a subclass of `Node` and can therefore be directly contained by the root object.

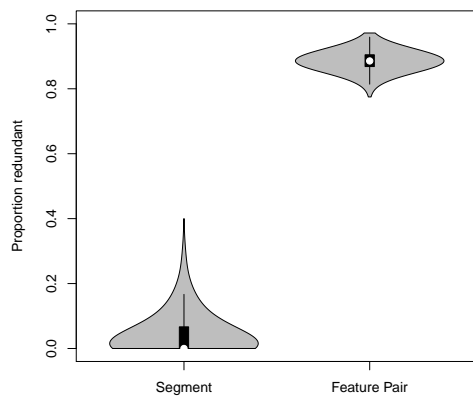
**DslModel** This MUT represents the median metamodel in the corpus. It exhibits a high percentage of non-coding feature pairs (median 87.0%) and a much lower percentage of non-coding class bits (median 35.0%). Non-coding segments are introduced where the segment encodes subclasses of the `NamedElement` meta-class. The inheritance hierarchy of the metamodel separates the meta-classes into two groups: children of `NamedElement`, and children of `ModelElement` (both subclasses of `Element`). The root meta-class (`Model`) has an unbounded containment reference to `ModelElement`,



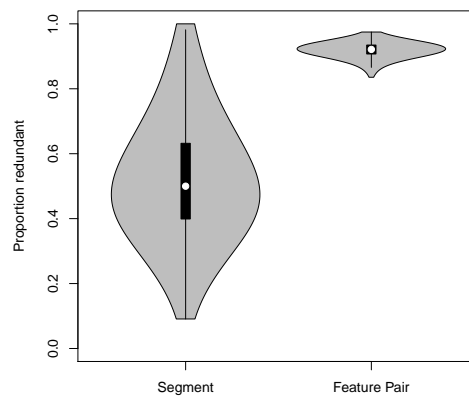
(a) *EG-MOF*



(b) *DslModel*



(c) *Java*



(d) *ATOM*

**Figure 7.14:** *Proportion of non-coding segments and feature pairs of the four representative metamodels.*

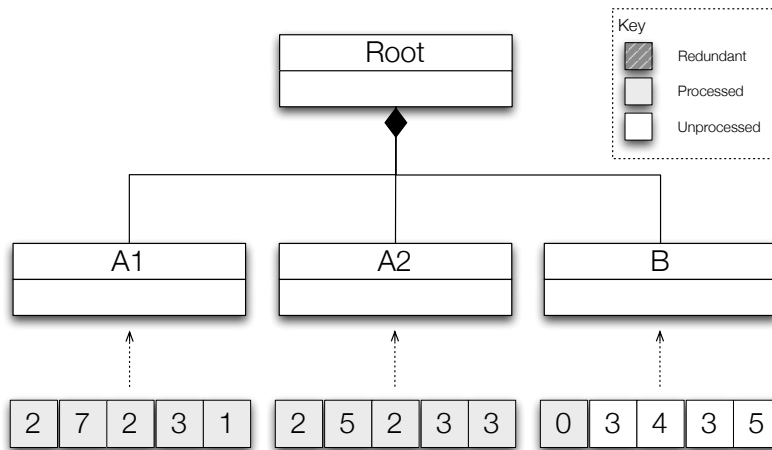
but not `NamedElement` which is, instead, contained by `ModelElement`. Segments that encode objects which are not directly contained by the root are more likely to become non-coding, as they rely on other segments to specify feature pairs to create the containment references that encapsulate them in the model.

The high proportion of non-coding feature pairs is due to the fact that the majority of the meta-features are bounded. Out of 23 features, 17 have an upper bound of either one or two. This highlights that is not the number of features in a metamodel that is most important, it is the upper bounds of those features that affects non-coding redundancy.

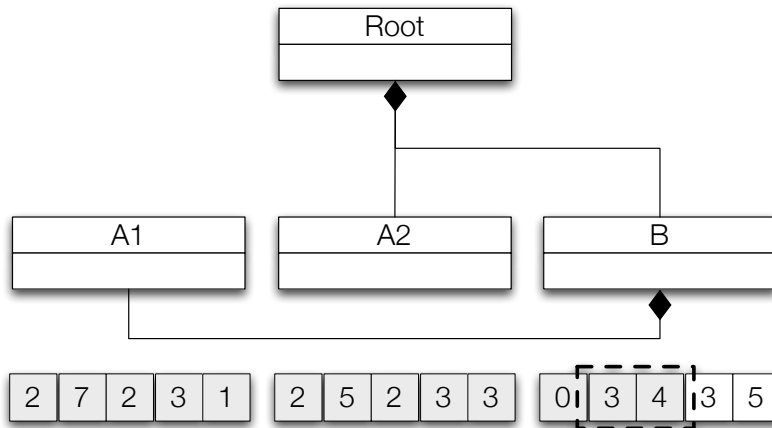
**Java** This MUT has more features than the previous two but exhibits similar amounts of non-coding feature pair redundancy to *DslModel* (median 88.6%). As with the *DslModel* metamodel, the majority of the MUT's features are bounded to one value – with only 10 out of 30 features being unbounded. The non-coding segment redundancy, however, is much lower than that of *DslModel*, with a median of 0.0%. All meta-classes bar one share a common super type: `Element`, enabling the root object to initially contain almost all generated objects. Segments can still become non-coding however, as the mapping transformation can reassign their container away from the root object. Another feature pair could then remove this containment reference by overriding the reference's value. If this occurs, it results in an object not being contained by any other (known as an *island*), making its associated segment non-coding. This is illustrated in figure 7.15: object A1 is initially assigned to the root object (phase 1) before object B becomes its container (phase 2). The containment reference is then reassigned to object A2, resulting in object A1 not being contained by another object, and therefore making its segment non-coding.

One potential solution to this issue is to move the root reference construction phase until after the object reference construction phase. Only objects who are not contained by another are then assigned to the containment references of the root. To examine this idea, we repeated the experiment for the *Java* metamodel using the alternative reference construction phase, the results are shown in figure 7.16b alongside the original plot. The results show the opposite effect to what was expected – there is a slight increase in the 75th percentile, but a reduction in the whisker, but this may be put down to noise in the analysis. The cause of non-coding segment redundancy in this metamodel is therefore the one meta-class which does not share the same super class as all other meta-classes. We conclude that the alternative mapping shows little benefit in reducing redundancy for this metamodel.

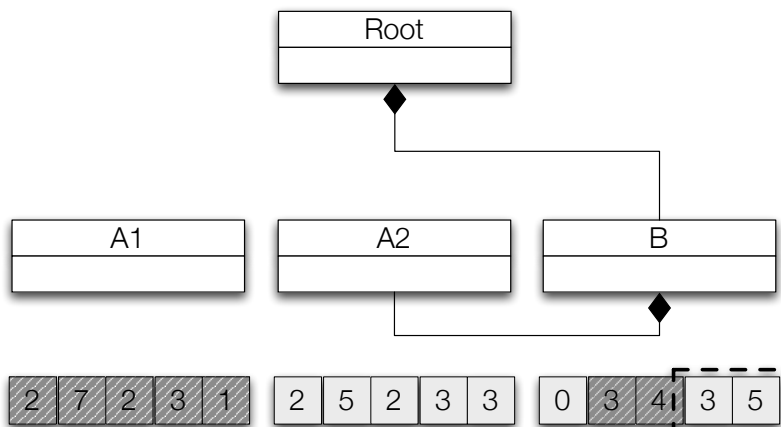
**ATOM** Once more, this MUT displays a large amount of non-coding feature pairs (median 92.1%). 47 out of 60 features have an upper bound of one. As shown in Table 7.3, 38 of these bounded



(a) Phase 1: All objects have been instantiated, their single-valued attributes set, and their container set to the root object (where possible).

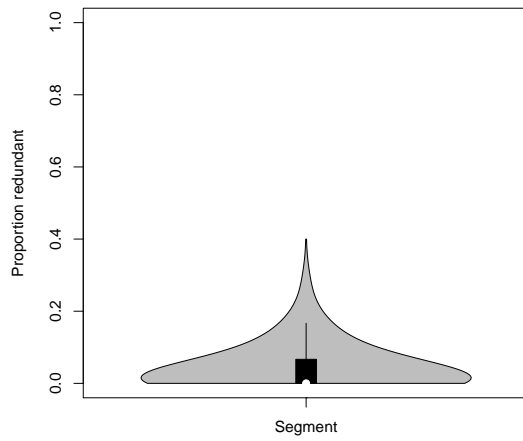


(b) Phase 2: The object reference assignment phase begins and object B takes ownership of object A1.

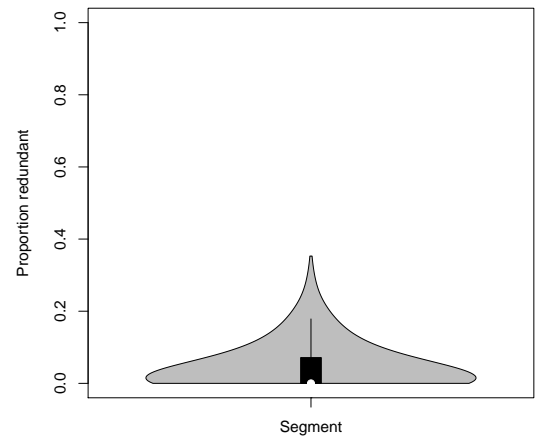


(c) Phase 3: The decoding of a second feature pair means that object B becomes the container of object A2. This containment reference has an upper bound of one, turning object A1 into an island and meaning that its encoding segment becomes non-coding.

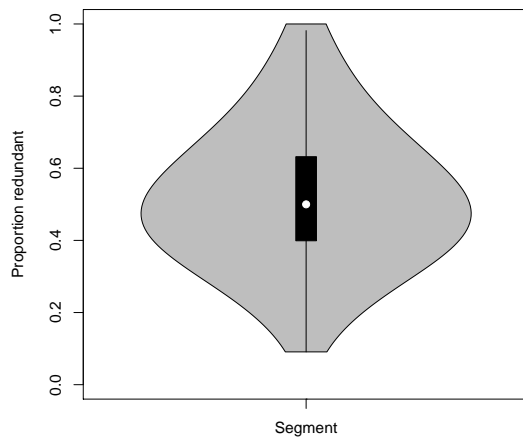
**Figure 7.15:** A containment reference reassignment that results in a non-coding segment.



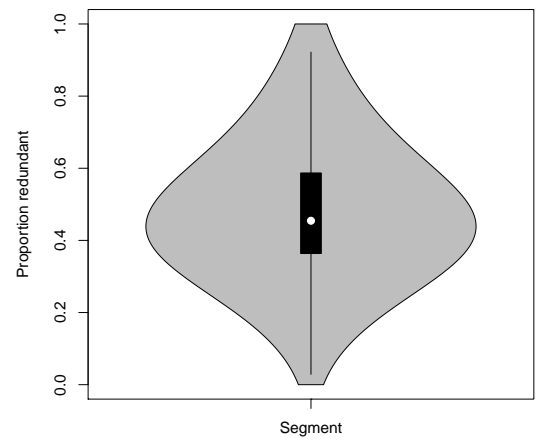
(a) Java (original process)



(b) Java (alternative process)



(c) ATOM (original process)



(d) ATOM (alternative process)

**Figure 7.16:** Repetition of the non-coding redundancy analysis using the alternative reference mapping process.

features are attributes, meaning that the majority of references (60%) are unbounded. Figure 7.14 shows a large variation in the amount of non-coding segments (median 50.0%). The reason for this is due to the flat structure of the metamodel. The MUT uses very little inheritance, but heavily uses containment references, meaning that the island-inducing phenomenon mentioned above can occur regularly.

The non-coding redundancy analysis for this metamodel was repeated with the alternative reference assignment process and the results are shown in figure 7.16d. The median percentage of non-coding segments reduced from 50.0% to 45.5%, and the interquartile range reduced by 2%. The alternative reference assignment process, therefore, can be seen to reduce non-coding segment redundancy for this metamodel. The objects that remained uncontained were instances of meta-classes not directly containable by the root object (Content and its three subclasses). Furthermore, only one meta-class (Entry) defines a reference to Contents and this reference has an upper bound of one. As such, it is understandable that segments encoding these objects will commonly be non-coding.

For completeness, we also ran the alternative reference construction process experiment for the *DslModel* MUT. The effects of attempting to assign all islands to the root were negligible. Due to *EG-MOF* not suffering from non-coding segment redundancy, we did not analyse the effect of the alternative process.

#### 7.4.4 Discussion

In this section, we have empirically investigated the amount of non-coding redundancy found in our representation, using the four archetypal metamodels from section 7.2 as the case studies. This analysis has highlighted how different structural properties influence the non-coding redundancy. Although the four MUTs were originally selected based purely on feature load, some of their other characteristics have been exposed in this analysis, and have helped to shed light on redundancy. A key influencer on redundancy is the inheritance hierarchy of the metamodel. *EG-MOF* and *Java* exhibited little or no non-coding redundancy due to the structure of their meta-class hierarchies, and the ability of their root objects to contain objects of (almost) every type. *Java*, *DslModel*, and *ATOM* all showed us how heavily influenced non-coding feature pair redundancy is by the upper bounds of features.

Reducing the number of feature pairs in each segment would reduce this non-coding feature pair redundancy, whilst also reducing the size of the model space. As discussed previously, however, some practitioners argue that this redundancy can reduce the difficulty of optimisation problems (i.e. it increases the

evolvability) as it increases the number of genotypes with high fitness [147, 82, 111, 189]. Determining an optimal number of feature pairs is a tricky task: one needs to ensure that they don't accidentally exclude solutions from the search space. Further investigation is needed to determine whether this redundancy is good for our representation or not. This investigation needs to analyse the effect of redundancy on a set of well-defined search problems, over a set of metamodels with distinct characteristics.

In this section we have not examined surjective redundancy. Due to the fact that the maximum allele value and the ordering of segments and feature pairs increases surjective redundancy, it is challenging to evaluate. Comparing each generated phenotype is infeasible due to the computational cost: comparing two graphs is NP-complete [169], and this procedure would require comparing huge numbers of models against one another. One potential way of reducing this complexity is to reuse our representation. Each model can be transformed into its canonical form. If we can order the segments and feature pairs in a way that makes direct comparison easier, we can reduce comparison time and perhaps make this analysis practical. This would also allow us to dig deeper into surjective redundancy and evaluate whether the representation is synonymously redundant or non-synonymously redundant, and gain insight into how much of an influence the MUT has on this.

Search spaces that exhibit a 'needle-in-a-haystack' landscape are commonly difficult for evolutionary algorithms. Yu and Miller [190] have shown how it is possible to create a *neutral network* – a network of equal-fitness solutions – that can guide the search towards the needle (global optimum). Therefore, before attempting to limit surjective redundancy, one should also attempt to gain insight into the fitness landscape.

## 7.5 *Discussion and Future Analysis*

In this chapter we have taken the first steps towards a detailed analysis of our search-amenable representation of models. There is much more to do. Our focus here was on analysing the locality and redundancy of our representation – two properties that have been shown to play a vital role in efficient evolutionary search. These analyses would not have been possible without a set of appropriate metamodels to define the search space. We have shown how different characteristics of metamodels affect both the locality and redundancy of the representation. The metamodel characteristics that we investigated, however, only touched the surface of those that are commonly exhibited and each will have some effect, minor or not. For instance, the notion of *connectivity*

will influence locality. Connectivity refers to the connections between meta-classes, and therefore the instances of meta-classes. If a model is loosely connected, then a small mutation to its genotype should result in fewer changes to the phenotype than in a highly connected model. We therefore need to extend our set of metamodel metrics, and devise a method to systematically evaluate the effects of each characteristic, and combinations of characteristics, on properties of the representation and on evolutionary search.

A crucial requirement is a set of benchmark MDE problems that can be tackled by search. In this chapter, we have defined four benchmark metamodels with which to evaluate representations of models. As more detailed analysis of metamodels is performed, these benchmarks should be updated. Work is needed in defining metamodels and fitness functions for a set of search problems. These problems could be grouped into the categories defined in section 2.3: discovery of models, validation of models, discovery of model management operations (MMO), and validation of MMOs. These benchmarks would enable rigorous comparison of different approaches to combining modelling and SBSE, and would help to move the field forward.

We now propose a set of experiments that will enable a deeper understanding of the representation, highlight areas of improvement, and provide guidance to users for tailoring it to their problem.

### ***7.5.1 Further Analyses of the Representation***

There are many more analyses that can be applied to our representation. In this thesis we focused solely on redundancy and locality, however there are other properties of the representation that need addressing. Furthermore, during the design of our representation, certain decisions had to be made that should be evaluated. We present these now.

***Properties of the Representation*** An important requirement for analysing the representation further is gaining a better understanding of how metamodels are commonly structured. Section 7.2.4 proposes a web-based automated metamodel measurement workbench. Such a workbench would allow practitioners to upload their metamodels and receive detailed measurements of their metamodel based on a set of standard metrics. In doing so, this analysis would add to the cumulative summary of the corpus, providing detailed statistics about the construction of metamodels, insight into modelling practices, understanding of the most used parts of the meta-metamodelling language, and more. This information would enable the selection of benchmark metamodels with which to analyse model representations – as we have done in this section. The analyses presented in this section are



not just applicable to our representation, but should be applied to any representation of models, be it a metamodel-specific representation or a generic one, like ours. Developers and users of a representation of this kind need to understand how the representation holds up in the face of differing metamodel characteristics. The information gained from these analyses can help developers improve the representation and help users tailor it to their needs.

The set of analyses that a representation of models should undergo are as follows. This list is inevitably incomplete and further analyses will arise as a result of performing these. This list can, however, be seen as a starting point for detailed research into this area.

- The analysis of the redundancy of our representation focused purely on surjective redundancy. Rothlauf groups representations into two categories based on surjective redundancy: those that are synonymously redundant and those that are non-synonymously redundant [147]. Rothlauf also defines representations to be *uniformly* redundant if the same number of genotypes encode every phenotype [147]. An investigation is required into understanding which category our representation fits, whether the redundancy is uniform, and whether this is the case for all metamodels or whether the metamodel influences the similarity of genotypes that encode the same phenotype.
- Further to knowing the amount and type of redundancy that is present in a representation, it is important to know how this redundancy affects the performance of evolutionary search. Goldberg defines the *complexity model* as a way of understanding this [147, section 3.1.3]. Furthermore, an empirical analysis on a set of benchmark case studies is required.
- Holland [73] introduced the notion of *schemata* to capture common patterns expressed in a genotype, and defined the *schema theorem* which defines how the number of instances of a particular schema evolves over time. Goldberg [56] defined the concept of *building blocks*: highly fit, short in length schemata that propagate through the generations of the evolutionary algorithm. Building blocks and the schema theorem can help to describe how evolutionary algorithms can solve a problem [147]. An investigation into building blocks of the representation could help improve the design of the representation such that it encourages building blocks to arise and propagate.
- Section 5.2.4 went some way to understanding the effects of the search operators by attempting to discover the optimal probabilities of operator application for the SAF adaptation study. The effects of the representation's search operators needs further investigation over a number of different problems.

- In section 7.3 we demonstrated that the size of individuals plays a key role in determining the locality of the representation for each metamodel. A crucial next step, therefore, would be to define an experiment and methodology for determining the optimal segment and feature pairs ranges for a given problem. Moreover, it is important to understand how differing levels of locality affect other properties of the representation, such as redundancy. Optimal sizes for individuals may be defined in a problem-specific manner, balancing the trade-offs between locality, redundancy, and other properties.
- An extension to the previous proposition would be to empirically analyse the size of real-world models. In the same way that we analysed different characteristics of metamodels in section 7.2, we can learn much from analysing a large number of models. Knowing the size and structure of real world models would allow the representation to be tailored to particular problems. Initial populations could be seeded with common structures in an attempt to increase evolvability. Of course we also want to include some diversity in the search population to ensure that optimal models that exist outside the recognised normal boundaries are not missed. A huge hurdle, however, is getting access to a large corpus of models that conform to a given metamodel. Newly developed metamodels will not have such a corpus, and those that do may not be publicly available.

**Design Decisions** The design process of the representation lead to a number of choices being made regarding the structure of the genotype and the genotype-phenotype mapping. Without proper empirical or theoretical analysis of the alternatives, it is possible that a bad decision was made. Here we list two of the design decisions made that require further investigation.

- In designing the crossover operator used in the representation, we decided to only allow crossover to occur at the gap between segments. The reason behind this was to attempt to allow entire objects to be swapped between the two models. An alternative would be to allow crossover to occur anywhere. Obviously this would be a more destructive operation, however the large amount of non-coding redundancy exhibited by the representation in our experiments may mean that this is a more fruitful approach. These two operators would need to be compared on a set of benchmark search problems; in conjunction with varying mutation probabilities. A technique such as the *response surface methodology* [116] would prove useful in determining optimal mutation probabilities for each crossover operator, whilst enabling a comparison of their efficacy.
- During the genotype-phenotype transformation, the phenotypic contribution of a feature pair can be overridden by another feature pair, resulting in non-coding redundancy. This is

a level of redundancy that we currently accept, but perhaps it would be better to instead *repair* the non-coding feature pair so as to remove non-coding redundancy. This would mean that individuals in the search population could be smaller as every bit would be used. This does, however, raise many questions. Do we repair the feature selector bit, or the feature value bit? How would one select an appropriate repair tactic? Do we randomly assign a new value, or use domain knowledge to select this? Perhaps the individual could be duplicated a number of times with each duplicate having a different repair value. Each duplicate could then be evaluated and the best is kept in the population whilst the others are discarded.

Many studies have demonstrated non-coding redundancy to be beneficial at increasing the evolvability of a search algorithm, so perhaps repairing the genotype would be a bad idea. However, the thought of increasing the efficiency of the encoding, and the potential of finding a solution quicker, means that genotype repair could be an interesting avenue of research.

### 7.5.2 *Summary*

This chapter has provided new insights into the ways in which practitioners commonly structure their metamodels, and used the information to select a representative set of metamodels to act as benchmarks in the analysis of representations. These metamodels have guided the empirical evaluation of the locality and redundancy of the representation, which in turn highlighted other characteristics of metamodels that affect these properties. We concluded with a proposal for future analysis of the representation in the hope of refining it and understanding how to best make use of it.



# Conclusion

# 8

THIS THESIS HAS ADDRESSED the challenge of devising a generic approach to applying Search-Based Software Engineering (SBSE) techniques to the challenges found in Model-Driven Engineering (MDE). SBSE has proven to be adept at discovering optimal solution to a broad range of software engineering problems [28, 62, 67]. MDE has been shown to be a promising approach to increasing productivity, maintainability, and portability [162, 176, 77]. There has, however, been little effort in applying SBSE techniques to MDE problems, or utilising MDE for SBSE techniques. The research in this thesis has addressed this gap and explored the following hypothesis, stated in section 1.2:

*A generic, search-amenable representation of models would enable the wealth of existing research into SBSE to be applied to a broad range of problems found in the MDE domain.*

To address this hypothesis, we defined the following objectives in section 1.2:

- To identify existing research that combines SBSE and modelling, and propose extensions where their synergy would be fruitful.
- To design and implement an encoding that can represent any (and all) model(s) that conforms to a given metamodel, and that is applicable to existing SBSE techniques.
- To design and implement a model-driven SBSE framework that uses this encoding and provides standard SBSE algorithms.
- To use the model representation to address a set of known challenges in the MDE domain.
- To evaluate properties of the representation that have been shown to be important for evolutionary search, and use the knowledge gained from the evaluation to provide guidance to users of the representation and to propose improvements to the representation.

## Contents

8.1 Thesis Contributions . . . . .	196
8.2 Future Work . . . . .	197
8.3 Coda . . . . .	201

**Chapter Structure** This chapter summarises the research performed in this thesis towards addressing the hypothesis above: section 8.1 highlights and discusses the novel contributions made in the thesis; and section 8.2 identifies opportunities for future work.

## 8.1 Thesis Contributions

The contributions made in this thesis are summarised below.

***A survey of the literature regarding integration of SBSE and modelling.*** Chapter 2 presented a survey of previous work that has attempted to apply SBSE techniques to MDE problems, or posit how the two fields can be fruitfully combined. Furthermore, we catalogued the case studies used in the literature and proposed to use this catalogue, and existing MDE model and MMO repositories to select a set of benchmark problems for use in the field.

***An initial, grammatical evolution-based prototype representation of models.*** This prototype demonstrated the feasibility of defining a representation that can express models conforming to a wide range of metamodels. Furthermore, we demonstrated that we could effectively apply metaheuristic search over this representation, and illustrated this on a case study to discover optimal models.

***The definition of a novel generic representation of models that is amenable to a wide range of existing metaheuristic optimisation techniques.*** The limitations of the prototype representation, and lessons learnt from its development, lead to the production of a model representation that is capable of encoding any model conforming to any given metamodel. We implemented the representation using state-of-the-art MDE technologies, where an individual is expressed as a model and the genotype-phenotype mappings are model transformations. To support the representation we developed a model-driven metaheuristic search framework that used the representation at its core. Metaheuristic algorithms and user-defined fitness functions are implemented as model management operations, whilst the search space and configuration parameters are embodied in models. Utilising MDE so heavily enabled the creation of a web-based search-space visualisation application. This allows users to inspect the search algorithm to manually validate the algorithm, and to gain insight into the evolution of their problem.

***The demonstration of the feasibility of using the representation to solve in-scope problems.*** In Chapter 3, we applied

our prototype model representation to discovering optimal opponents in a simple fighting game. Chapter 5 later employed our improved representation of models to address the same case study and showed an improvement in performance. Chapter 5 also addressed the challenge of self-adaptive systems, presenting a model- and search-based approach to extracting a model of runtime system behaviour for use in component-level adaptation. Finally, chapter 6 presented a framework for applying sensitivity analysis to MDE models, and utilised the representation for this task. We applied sensitivity analysis to a set of models that are used to aid the decision making process of high-value acquisition scenarios, and highlighted not only how different elements in the model contributed towards the model's value, but also that certain 'optimal' solutions were heavily affected by model uncertainty and should be viewed with less confidence. Furthermore, we identified three areas where uncertainty arises in MDE.

*The empirical analysis of two important properties of the representation.* Chapter 7 performed an analysis of the locality and redundancy of the model representation. This highlighted how different areas of the representation contribute towards these properties and provided some basic guidelines to help users configure the representation for use in their problem. Furthermore, we set out a detailed plan for future empirical analysis of the representation, or any similar representation of models.

*A large-scale analysis of the structure of metamodels.* In chapter 7, we performed a structural analysis of a large corpus of publicly available metamodels, providing detailed insight into the ways in which practitioners commonly build their metamodels.

*The identification of a set of representative metamodels.* The analysis of the metamodel corpus in chapter 7 enabled the selection of a set of representative metamodels that act as an initial set of benchmarks for evaluating the properties of representations of models. As more metamodels are added to the corpus, this benchmark set will be updated and extended.

## 8.2 Future Work

In this section we describe some potential avenues of future work. We separate the work into: extensions to the representation and search framework; extensions to the adaptation approach presented in section 5.2; extensions to the sensitivity analysis presented in section 6; and finally we posit other applications to which the representation and its search framework could be applied.

### 8.2.1 *Representation and Search Framework*

With respect to Crepe and MBMS, we propose the following extensions and investigations.

***Address representation limitations*** Section 4.7 highlighted a number of issues relating to the representation, and section 5.1 illustrated that the representation struggles with reference-heavy metamodels. To address the reference-related issue, it is possible to manually define a refactored version of the metamodel that is more suited to use with the representation. This, however, places an extra cost on the user and potentially raises concerns regarding the semantics of the refactored metamodel. It may be possible to automate this refactoring transformation, based on pattern matching applied to the input metamodel, but semantics may still remain an issue. In section 4.7 we discuss potential ways to address the issues of validation, minimum scoping and custom data types.

***Probabilistic finitisation*** Presently, the information captured in the finitisation model is selected for instantiation without discrimination. Although a user may specify a range of values for an attribute, it may be the case that a probability distribution over that range better captures the semantics of the domain. Rose and Poulding [145] have demonstrated the efficacy of using probability distributions for this, but use a grammar-based representation of the metamodel. Probability distributions don't have to be defined only for attribute data: we can define them for meta-classes as well, enabling the search to produce more realistic models.

***Performance optimisation*** We stated throughout this thesis that our focus was on the efficacy of a generic, search-amenable representation for models, as opposed to developing a highly optimised, high-performance metaheuristic optimisation platform. There are many areas where optimisation can be achieved. For instance, we could cache finitisation information for quick retrieval, and use profiling techniques to discover bottlenecks in the algorithms. One of the known bottlenecks is the fact that EOL is an interpreted language. Rewriting the original representation and search framework in EOL from Java improved maintainability and increased productivity, but it introduced a performance hit. One potential solution might be to use a model-driven approach to developing MBMS. The search algorithms and operators could be specified in high-level models, and performance-optimised code could be generated from them.

***Visualisation*** In section 4.5.4 we demonstrated how we could apply a model-to-text transformation to the search history model to produce a web-based, interactive visualisation of the search.



In its current form, models are represented in the generic textual syntax, HUTN. A more desirable visualisation would be to visualise solutions in their native concrete syntax, however work is required in determining how this could be made possible. Visualisation could be performed whilst the search is taking place: for instance, to support interactive evolution. We could even visualise the search space, not just the search history, though the multi-dimensionality of models would make this very complicated.

**Representation theory** In chapter 7 we evaluated Crepe with respect to two important properties of representations. To gain a deeper understanding of the representation, in order to discover improvements and offer guidance to users, we need to extend this evaluation. Section 7.5 proposed a set of analyses to address this.

**Model and metamodel corpus analysis** In section 7.2, we performed an analysis of a large corpus of metamodels to discover the most common structural characteristics of metamodels. The metrics used in the analysis are very basic and there are many more metrics that we could analyse. Understanding metamodels will enable us to improve the representation and provide useful information for users based on the properties of their metamodel. Furthermore, analysing corpora of models would help to optimise the finitisation probability distributions (above) or provide seeding strategies for MBMS. It is unlikely, however, that large corpora of models would be available for many metamodels, and so this will only be useful for general purpose modelling languages.

**Investigate designing the representation with respect to the Epsilon Model Connectivity [91] (EMC) layer** The representation has been designed for MOF-like metamodels, and Crepe has been implemented for EMF. The model connectivity layer in Epsilon enables differing modelling technologies to be used within Epsilon, even within the same MMO. We utilise that in MBMS: the EMF driver is used to load the metamodel, finitisation model, and search model, and the Properties file driver loads the configuration parameters. EMC works by defining an interface that all drivers need to implement. We could investigate the possibility of reimplementing Crepe such that it uses the EMC model interface. This would allow us to use MBMS on a wide range of modelling technologies. To enable this, we would also need to define a generic approach to metamodel finitisation.

### 8.2.2 Adaptation and Sensitivity Analysis

In chapter 5 we posited that the representation could be used for component-level adaptation and illustrated this on SAF. The results were not as promising as one would have hoped, however

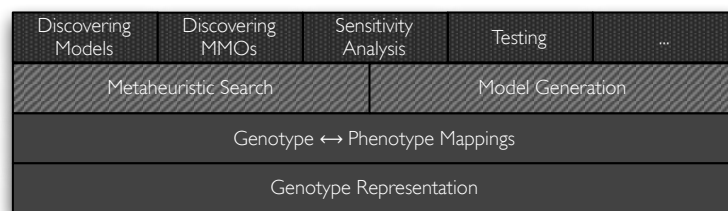
this could be down to issues relating to SAF itself. To determine whether our proposed model- and search-based framework is feasible, we need to apply it to real-world case studies. Furthermore, it would be interesting to continue the investigation into different seeding strategies to improve the performance of the search, and discover ways to optimise both Crepe and MBMS to enable its use at runtime.

Chapter 6 presented a framework that enables metamodellers to create tool support for applying sensitivity analysis to models conforming to their metamodel(s). Uncertainty is an important topic in MDE (and software engineering in general), and so it would be interesting to further explore the capabilities of such a framework. Crepe was used to apply the sensitivity analysis, but no metaheuristic search was performed – instead sampling methods were applied to the genotype. Harman et al. [66] use SBSE to perform sensitivity analysis on system requirements in order to discover which requirements are sensitive to cost. Embedding MBMS in our sensitivity analysis framework would allow the uncertainty space to be explored further. Fitness functions could be used to steer the search towards areas that exhibit large deviations, in order to allow developers to gain a better understanding of their models and devise methods for managing their uncertainty.

### 8.2.3 Other Applications

We showed with the sensitivity analysis work that our representation need not only be useful for metaheuristic search. Figure 8.1 illustrates the building blocks of the representation and how they combined to address MDE challenges. Below we list some other potentially fruitful areas where the representation could be used.

**Figure 8.1:** The components and uses of our representation.



**Model generation** Automatically generating models can be useful for testing purposes. We discussed, in section 2.3, the existing approaches that use SBSE to discover models for testing MMOs. It is unclear whether the existing approaches could deal with *scale*. The size of models being created in industry is increasing and huge models are needed to test systems that deal with such models [115]. To increase the size of a model in our representation, one simply needs to add more segments and feature pairs to the individuals.

**Testing MMOs** As mentioned, there has been work in the area of discovering models for testing MMOs. Applying Crepe and MBMS to the case studies used in those papers would provide evidence as to whether or not a generic representation is capable of performing as well as problem-specific representations (in terms of execution time and quality of test set). The cost of using a generic representation as compared to a task-specific representation is likely to be performance. However, the cost of developing a task-specific representation (e.g. time, expertise) may outweigh the cost of using a generic representation.

**Running transformations in reverse** *Bidirectional* transformations are challenging to implement, but are particularly important in synchronising models [171, 55]. Many modern transformation languages (e.g. ETL, ATL) do not provide support for bidirectionality [55], however we could use Crepe and MBMS to *search* for the input to a transformation when given the output. We could apply MBMS over the input model space to discover a model that, when executed with the model transformation, results in the given output model. If the original input model is known, then it can be used to seed the initial population.

**Pattern detection** The genotypic representation exhibits patterns<sup>1</sup>. The same analyses applied to the metamodel corpus in section 7.2, if couched differently, could be applied to the representation. This has a number of advantages. Loading and reading models takes time: traversing each model element and reading attribute values is slow. There exist, however, many fast algorithms that detect patterns in strings (e.g. [14, 89]). Our representation could be used to not only quickly identify common structural characteristics, or calculate various software metrics, but it could be used to detect more complex relationships, such as correlations between the values assigned to features.

<sup>1</sup> The ideas here came out of discussions at ICSE 2013 with Benoit Baudry, a researcher at INRIA in France (<http://people.rennes.inria.fr/Benoit.Baudry/>).

## 8.3 Coda

Model-Driven Engineering is argued to be the future of software engineering. Raising the level of abstraction, and automating challenging tasks will be an important way to address the complexity and scale of the systems of the future. MDE hasn't, however, been adopted on a universal scale and part of the reason for this is that MDE practices and tools haven't matured to the same level as traditional software development techniques. Search-Based Software Engineering is a practice that can help MDE gain universal status. We believe that a generic representation for MDE models can lower the entry-point for MDE practitioners wanting to benefit from SBSE techniques and therefore provide vital support in the development of modern systems.



# V

*Appendix*



## *Literature Review*

# A

### *A.1 Discovering Models*

Paper Type	Algorithm	Input	Encoding	Output	Operators	Fitness
[46] O	Bacteriological algorithm	Set of models, set of metamodel partitions	Direct	Optimised set of models	Mutation: select model element and replaces its value from a different partition None	Coverage of partitions: promotes coverage of previously uncovered partitions Coupling metrics
[69] O	Multiple-run random search	Java system, from which a UML-like model is extracted	Triples representing method move refactorings (old class, method, new class) String rules that: 1) select elements, 2) construct transitions, 3) replicate organisms As above	Pareto front of refactorings Set of 'compliant behavioural models'	Random mutation; replication	User-defined scenarios; temporal properties; rewards fewer transitions and determinism As above
[59] D	Avida-MDE: 'digital organism'-based evolution	'Instinctual knowledge': system class diagram, existing state diagrams As above	String rules that: 1) select elements, 2) construct transitions, 3) replicate organisms As above	Set of 'compliant behavioural models'	Random mutation; replication	User-defined scenarios; temporal properties; rewards fewer transitions and determinism As above
[58] D	As above	Current system model, monitoring information	Graph representation of system components and connections	As above, but categorised Set of target system models	Mutation: randomly change properties of components and connections; Crossover: randomly exchange components and interconnections Mutation and crossover	Domain-independent functions (cost, network performance, data reliability)
[136, 138] D	Genetic algorithm	Current system model, monitoring information	Graph representation of system components and connections	Set of target system models	Mutation and crossover	Domain-independent functions (cost, network performance, data reliability)
[13] O	SPEA2	Class diagram; dependencies between features; constraints over features Uses cases	Integer genotype: each gene represents a feature, allele represents assignment to a class Groups of attributes and methods; each group represents a class	Set of optimal refactorings	Mutation and crossover	Coupling and cohesion metrics
[167] O	NSGA-II	Class diagram; dependencies between features; constraints over features Uses cases	Integer genotype: each gene represents a feature, allele represents assignment to a class Groups of attributes and methods; each group represents a class	Sets of visualised class diagrams representing the problem	Custom mutation and crossover	Coupling and cohesion metrics
[54] O	Genetic programming	Defect examples; model under test	A set of IF-THEN rules: IF <combination of metrics> THEN <defect type>	Rules to detect defects in the model	Strongly-typed tree crossover; typed mutation	Expected number of defects - number of detected defects
[18] D	NSGA-II	Scenario model, component model	Expressed in an intermediate modelling language, M2M transformation to solution model	Pareto front of solution models	Single-point crossover	Weighted number of satisfied capabilities and overall cost of solution: 1) Weighted: favouring either dissimilarity or coverage;



Paper Type	Algorithm	Input	Encoding	Output	Operators	Fitness	
[21]	D	Simulated annealing	Model fragments	Alloy used to generate model space.	Set of test models	Neighbourhood: add or remove random test model from the set	Maximize both the dissimilarity of solutions and the coverage of model fragments. 2) 'Minimal coverage multi-category': punishes solutions that don't meet a defined minimal coverage (90%)
[39]	O	Evolutionary algorithm	UML and AADL	Direct	Optimised architecture	Architecture-specific operators, based on patterns	-
[137]	D	Genetic algorithm	KAOS goal model; utility functions; executable specification of DAS	Linear genome, each gene represents a single, non-invariant goal: flag states if goal relaxed, relax operator, two floats for relax function	One or more relaxed goal models	Two point crossover and single point mutation	Minimise number of relaxed goals and adaptation;
[30]	D+O	Multi-objective (not specified)	Model of PLA	Direct	Optimal product line design	Five mutation operators taken from software architectures literature; single PLA-specific mutation and crossover operators; model is repaired if operator breaks constraints	PLA metrics
[141]	O	NSGA-II	Feature model	Linear genotype, each gene is a binary combination stating whether a product is enabled or disabled Stochastic model generator	Pareto front of optimal configurations	Single point crossover; mutation	Trade-off between total product value and product line integrity
[117]	D	Differential evolution; Particle swarm optimization	Model generator parameters	Probability distribution over the set of input models (stochastic context-free grammar)	Set of labelled transition systems	-	Evaluation is wrt the parameters to the model generator
[145]	D	Hill climbing	Metamodel	Set of test models	Set of test models	Mutate probabilities	Coverage of model transformation

**Table A.1:** *Papers addressing the discovery of models.*

## *A.2 Discovering Model Management Operations*

Paper Type	Algorithm	Input	Encoding	Output	Operators	Fitness
[85]	DT Particle swarm optimisation	Transformation (source model, target model, mapping); Mapping expressed as predicates	Integer string: each position in the string represents a source model element, each allele selects a mapping block to assign to that construct Predicate sequences	Set of transformation blocks	Standard velocity and position update equations	Similarity match on construct's predicate against all predicates in selected mapping block
[86]	DT Genetic algorithm	A test case: source model, target model, traceability links	Predicate sequences	Set of traces, each with an associated 'risk'	Standard crossover and mutation	Risk is calculated using an immune system-inspired sequence matching algorithm Calculates the percentage of exact matches
[43]	DT Genetic programming	Example source and target models; source and target metamodels	Tree structures that represent JESS rules	Set of JESS transformation rules	Crossover exchanges entire rules; Rule mutation: add/remove rule randomly; Construct mutation: add/remove/modify constructs	
[10]	DR Simulated annealing and genetic algorithm (comparison)	Source and target models, represented as predicates; Exhaustive list of possible refactorings	Vector of string arrays. Each array is a refactoring definition: [id, operator, parameters]	Sequence of refactorings	Single-point crossover; swap a number of refactorings; Randomly mutate a random number of refactorings	Application of refactorings to source model. Calculate predicate-similarity of result with known target
[44]	DT Genetic programming	As above	As above	As above	As above	As above
[88]	DM Genetic algorithm (comparison against random search and simulated annealing)	Initial model; set of revised models; computed list of operations	Fixed length genotype, each gene represents one operation. Genotype length = number of operations	Sequence of operations that minimises the number of disabled operations	Crossover: standard; Mutation: select two genes and swap	Calculate number of disabled operations (preconditions not satisfied). Pairwise combination to reduce complexity
[114]	DT NSGA-II; P-NSGA-II	Source model; target model;	Binary tree representation of transformation rules	Set of transformation rules	Tree crossover; function or terminal mutation	Maximise transformation correctness and target model quality; minimise rule complexity

**Table A.2:** Papers addressing the discovery of model management operations. DT: discovering transformations; DR: discovering refactorings; DM: discovering merges.



# Code Listings

# B

## B.1 Fighter Description Language Grammar

### B.1.1 EMFText Grammar

```
1 SYNTAXDEF fantastic
2 FOR <fighter>
3 START Bot
4
5 OPTIONS {
6   usePredefinedTokens = "false";
7   reloadGeneratorModel = "true";
8   tokenspace = "0";
9 }
10
11 TOKENS {
12   DEFINE NAME $('A'..'Z'|'a'..'z'|'o'..'g'|'_'|'-' )+ $;
13   DEFINE NUMBER $('o'..'g')+ $;
14
15   DEFINE CHARACTERISTIC $('punchReach'|'punchPower'|'
16     kickReach'|'kickPower') $;
17
18   DEFINE CONDITION $('always'|'near'|'far'|'much_stronger'|'
19     stronger'|'even'|'weaker'|'much_weaker') $;
20
21   DEFINE MOVE_ACTION $('run_towards'|'run_away'|'jump'|'
22     crouch'|'stand'|'walk_towards'|'walk_away') $;
23
24   DEFINE FIGHT_ACTION $('block_low'|'block_high'|'punch_low
25     '| 'punch_high'|'kick_low'|'kick_high') $;
26
27   DEFINE WHITESPACE $(' '|'\t'|'\f') $;
28   DEFINE LINEBREAK $('r\n'|'\r'|'\n') $;
29 }
30
31 TOKENSTYLES {
32   "NAME" COLOR #000000, BOLD;
33   "NUMBER" COLOR #2A00FF;
34
35   "CONDITION" COLOR #00bboo, BOLD;
36   "and" COLOR #00bboo, BOLD;
37   "or" COLOR #00bboo, BOLD;
38
39   "CHARACTERISTIC" COLOR #7F0055, BOLD;
40   "MOVE_ACTION" COLOR #7F0055, BOLD;
41   "FIGHT_ACTION" COLOR #7F0055, BOLD;
42 }
43
44 RULES {
45   Bot ::= name[NAME] "{" linebreaks[LINEBREAK] personality
46     behaviour "}";
```

```

40 Personality ::= (characteristics linebreaks[LINEBREAK])*;
41
42 Characteristic ::= type[CHARACTERISTIC] "=" value[NUMBER];
43
44 Behaviour ::= (rules linebreaks[LINEBREAK])*;
45
46 Rule ::= condition "[" moveAction " " fightAction "]" ;
47
48 AtomicCondition ::= type[CONDITION] ;
49
50 AndCondition ::= parts " " "and" " " parts ;
51
52 OrCondition ::= parts " " "or" " " parts ;
53
54 MoveAction ::= type[MOVE_ACTION] | "choose(" (type[
55     MOVE_ACTION] " ")+ type[MOVE_ACTION] ")" ;
56
57 FightAction ::= type[FIGHT_ACTION] | "choose(" (type[
58     FIGHT_ACTION] " ")+ type[FIGHT_ACTION] ")" ;
59 }

```

**Listing B.1:** *The EMFText grammar definition for FDL.*

### B.1.2 Xtext Grammar

```

1 grammar org.xtext.example.mydsl1.MyDsl with org.eclipse.
  xtext.common.Terminals
2
3 generate myDsl "http://www.xtext.org/example/mydsl1/MyDsl"
4
5 Model:
6   'Sample' '{' '\n'
7   personality=Personality
8   behaviour=Behaviour
9   '}';
10
11 Personality:
12   (characteristics+=Characteristic)+;
13
14 Characteristic:
15   'punchReach' '=' value=NUMBER '\n' |
16   'punchPower' '=' value=NUMBER '\n' |
17   'kickReach' '=' value=NUMBER '\n' |
18   'kickPower' '=' value=NUMBER '\n' ;
19
20 Behaviour:
21   (rules+=BehaviourRule)+
22   always=AlwaysRule;
23
24 AlwaysRule:
25   'always' '[' action=Action ']' '\n' ;
26
27 BehaviourRule:
28   condition=Condition '[' action=Action ']' '\n' ;
29
30 Condition:
31   type+=ConditionType | (type+= ConditionType ' ' ('and' | '
32   or') ' ' type+=ConditionType);
33
34 Action:
35   (moveAction+=MoveAction | 'choose' '(' (moveAction+=
36     MoveAction ' ')* moveAction+=MoveAction ')') ' '
37     (fightAction+=FightAction | 'choose' '(' (
38       fightAction+=FightAction ' ')* fightAction
39       +=FightAction ')');

```

```

36
37
38 terminal MoveAction:
39   'run_towards' | 'run_away' | 'jump' | 'crouch' | 'stand'
40   | 'walk_towards' | 'walk_away';
41
42 terminal FightAction:
43   'block_low' | 'block_high' | 'punch_low' |
44   'punch_high' | 'kick_low' | 'kick_high';
45
46 terminal ConditionType :
47   'near' | 'far' | 'much_stronger' |
48   'stronger' | 'even' | 'weaker' | 'much_weaker';
49
50 terminal NUMBER :
51   '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

```

**Listing B.2:** *The Xtext grammar definition for FDL.*

## B.2 Delivery Man Problem

### B.2.1 Hill Climbing Algorithm

```

1  import "platform:/resource/jw.research.crepe.casestudies.
   util/finModelUtil.eol";
2  import 'platform:/plugin/jw.research.crepe/pancake/algs/
   hillClimbing.eol';
3
4  // Utility variables
5  var t = 0; // iteration counter
6  var random = setupRandomSeed();
7  var maxAllele = 1000;
8
9  // Necessary for genotype to phenotype mapping
10 var unassignedReferences;
11 var modelObjects;
12 var instantiableClasses;
13 var objectSegmentMap;
14 var featureFeaturePairMap;
15 var featureCollection;
16 var fitFunctionIsEol = true;
17
18 // Setup finitisation model
19 var fin = new FIN!MetamodelFinitisation;
20 fin.rootObject = MM!EClass.all.selectOne(c | c.name == "
   Route");
21 fin.finitisations.add(createEObjectFeatureListFinitisation(
   selectFeatureByName("Stop", "city"), MAP!City.all));
22 fin.ignores.add(MM!EClass.all.selectOne(c | c.name == "
   Product"));
23 fin.ignores.add(selectFeatureByName("Stop", "goods"));
24
25 // Configure segments based on #cities
26 setOrCreateConfigurationProperty("population.segments.
   quantity.max", MAP!City.all.size());
27 setOrCreateConfigurationProperty("population.segments.
   quantity.min", MAP!City.all.size());
28
29 // Now search
30 for (seed in Sequence
   {724377,7199439,4096091,7163541,2691539,8282485,4824792,
   6231867,6579662,6256102}) {
31   random = new Native("java.util.Random")(seed.asLong());

```

```

32 seed.print();
33 t = 0; // Reset generation count
34 commence();
35 }
36
37 operation doEvaluateFitness(candidate : MM!EObject) : Real {
38   MOD.getResource.getContents().clear();
39   MOD.getResource.getContents().add(candidate);
40   MOD.store();
41
42   // First of all punish solutions which don't include all
43   // cities (i.e. aren't valid)
44   var missingCities = 0;
45   for (city in MAP!City.all) {
46     if (candidate.stops.select(s | s.city == city).size() ==
47         0) {
48       missingCities = missingCities + 1;
49     }
50   }
51
52   if (missingCities <> 0) {
53     return missingCities;
54   }
55
56   // Now calculate the distance score for valid solutions
57   var distance = 0.0;
58   var isComplete = false;
59   for (i in Sequence{0..candidate.stops.size()}) {
60     var stop = candidate.stops.at(i).city;
61     var nextStop;
62     // Need to get back to the start
63     if (i + 1 >= candidate.stops.size()) {
64       nextStop = candidate.stops.at(0).city;
65       isComplete = true;
66     } else {
67       nextStop = candidate.stops.at(i+1).city;
68     }
69
70     distance = distance + MAP!Distance.all.selectOne(d |
71       (d.city1 == stop and d.city2 == nextStop) or
72       (d.city1 == nextStop and d.city2 == stop)).
73       distance;
74
75     if (isComplete) break;
76   }
77
78   var fitness = 1.asDouble() - (1.asDouble()/distance.
79     asDouble());
80   return fitness;
81 }

```

**Listing B.3:** Solving the delivery man problem using hill climbing.

## B.2.2 Random Search Algorithm

```

1 import "platform:/resource/jw.research.crepe.casestudies.
2   util/finModelUtil.eol";
3 import 'platform:/plugin/jw.research.crepe/pancake/algs/
4   hillClimbing.eol';
5
6 // Utility variables
7 var t = 0; // iteration counter
8 var random = setupRandomSeed();
9 var maxAllele = 1000;
10
11 // Necessary for genotype to phenotype mapping
12 var unassignedReferences;

```



```

11 var modelObjects;
12 var instantiableClasses;
13 var objectSegmentMap;
14 var featureFeaturePairMap;
15 var featureCollection;
16 var fitFunctionIsEol = true;
17
18 // Setup finitisation model
19 var fin = new FIN!MetamodelFinitisation;
20 fin.rootObject = MM!EClass.all.selectOne(c | c.name == "
    Route");
21 fin.finitisations.add(createEObjectFeatureListFinitisation(
    selectFeatureByName("Stop", "city"), MAP!City.all));
22 fin.ignores.add(MM!EClass.all.selectOne(c | c.name == "
    Product"));
23 fin.ignores.add(selectFeatureByName("Stop", "goods"));
24
25 // Configure segments based on #cities
26 setOrCreateConfigurationProperty("population.segments.
    quantity.max", MAP!City.all.size());
27 setOrCreateConfigurationProperty("population.segments.
    quantity.min", MAP!City.all.size());
28
29 // Now search
30 for (seed in Sequence
    {724377,7199439,4096091,7163541,2691539,8282485,4824792,
    6231867,6579662,6256102}) {
31     random = new Native("java.util.Random")(seed.asLong());
32     seed.print();
33     t = 0; // Reset generation count
34     commence();
35 }
36
37 operation doEvaluateFitness(candidate : MM!EObject) : Real {
38     MOD.getResource.getContents().clear();
39     MOD.getResource.getContents().add(candidate);
40     MOD.store();
41
42     // First of all punish solutions which don't include all
43     // cities (I.e. aren't valid)
44     var missingCities = 0;
45     for (city in MAP!City.all) {
46         if (candidate.stops.select(s | s.city == city).size() ==
47             0) {
48             missingCities = missingCities + 1;
49         }
50     }
51     if (missingCities <> 0) {
52         return missingCities;
53     }
54
55     // Now calculate the distance score for valid solutions
56     var distance = 0.0;
57     var isComplete = false;
58     for (i in Sequence{0..candidate.stops.size()}) {
59         var stop = candidate.stops.at(i).city;
60         var nextStop;
61         // Need to get back to the start
62         if (i + 1 >= candidate.stops.size()) {
63             nextStop = candidate.stops.at(0).city;
64             isComplete = true;
65         } else {
66             nextStop = candidate.stops.at(i+1).city;
67         }
68
69         distance = distance + MAP!Distance.all.selectOne(d |
            (d.city1 == stop and d.city2 == nextStop) or

```

```

70         (d.city1 == nextStop and d.city2 == stop)).
           distance;
71
72     if (isComplete) break;
73 }
74 var fitness = 1.asDouble() - (1.asDouble()/distance.
   asDouble());
75 return fitness;
76 }
77
78 /**
79  * Override default impl: Just one neighbour – a random one.
80  */
81 operation SEARCH!Individual createNeighbours() : Sequence {
82     var neighbours = new Sequence;
83     neighbours.add(randomlyCreateIndividuals(1).at(0));
84     return neighbours;
85 }

```

**Listing B.4:** Solving the delivery man problem using random search.

## B.3 Metamodel Analysis

### B.3.1 Analysis Program

```

1  /**
2  * Metamodel Statistics: Calculates some simple structural
   statistics about the input metamodel.
3  */
4
5  // Metamodel name
6  var modelName = MM.getAliases().at(0);
7
8  // Num meta-classes
9  var numMetaClasses = EClass.all.size();
10 var numConcreteMetaClasses = EClass.all.select(c|not c.
   abstract).size();
11
12 // Num meta-features
13 var numMetaFeatures = EStructuralFeature.all.size();
14 var numReferences = EReference.all.size();
15 var numContainmentReferences = EReference.all.select(r|r.
   containment).size();
16 var numNonContainmentReferences = EReference.all.select(r|
   not r.containment).size();
17 var numAttributes = EAttribute.all.size();
18
19 // Mean num meta-features per meta-class
20 var avgMetaFeaturesPerClass = numMetaFeatures.asDouble() /
   numMetaClasses.asDouble();
21 var avgReferencesPerClass = numReferences.asDouble() /
   numMetaClasses.asDouble();
22 var avgAttributesPerClass = numAttributes.asDouble() /
   numMetaClasses.asDouble();
23
24 // Median num meta-features per meta-class
25 var medMetaFeaturesPerClass = EClass.all.collect(c|c.
   eStructuralFeatures.size()).sortBy(i|i).median();
26 var medAllMetaFeaturesPerClass = EClass.all.collect(c|c.
   eAllStructuralFeatures.size()).sortBy(i|i).median();
27 var medReferencesPerClass = EClass.all.collect(c|c.
   eReferences.size()).sortBy(i|i).median();
28 var medAllReferencesPerClass = EClass.all.collect(c|c.
   eAllReferences.size()).sortBy(i|i).median();

```

```

29 var medAttributesPerClass = EClass.all.collect(c|c.
    eAttributes.size()).sortBy(i|i).median();
30 var medAllAttributesPerClass = EClass.all.collect(c|c.
    eAllAttributes.size()).sortBy(i|i).median();
31
32 // Featureless classes
33 var numClassesWithNoImmediateFeatures = EClass.all.select(c|
    c.eStructuralFeatures.size()==0).size();
34 var numAbstractClassesWithNoImmediateFeatures = EClass.all.
    select(c|c.abstract and c.eStructuralFeatures.size()==0)
    .size();
35 var numConcreteClassesWithNoImmediateFeatures = EClass.all.
    select(c|not c.abstract and c.eStructuralFeatures.size()
    ==0).size();
36 var numClassesWithNoFeaturesAtAll = EClass.all.select(c|c.
    eAllStructuralFeatures.size()==0).size();
37 var numAbstractClassesWithNoFeaturesAtAll = EClass.all.
    select(c|c.abstract and c.eAllStructuralFeatures.size()
    ==0).size();
38 var numConcreteClassesWithNoFeaturesAtAll = EClass.all.
    select(c|not c.abstract and c.eAllStructuralFeatures.
    size()==0).size();
39
40 // Average attributes per class (excluding featureless
    classes)
41 var avgFeaturesExclFeatureless = numMetaFeatures.asDouble()
    / EClass.all.select(c|c.eAllStructuralFeatures.size()>0)
    .size().asDouble();
42 var avgReferencesExclFeatureless = numReferences.asDouble()
    / EClass.all.select(c|c.eAllStructuralFeatures.size()>0)
    .size().asDouble();
43 var avgAttributesExclFeatureless = numAttributes.asDouble()
    / EClass.all.select(c|c.eAllStructuralFeatures.size()>0)
    .size().asDouble();
44
45 // Reference upper bounds
46 var refUpperOne = EReference.all.select(r | r.upperBound ==
    1).size().asDouble() / numReferences.asDouble();
47 var refUpperMany = EReference.all.select(r | r.upperBound ==
    -1).size().asDouble() / numReferences.asDouble();
48 var refUpperN = EReference.all.select(r | r.upperBound <> 1
    and r.upperBound <> -1).size().asDouble() /
    numReferences.asDouble();
49
50 // Print as CSV
51 var result = modelName;
52 for (s in Sequence(numMetaClasses, numConcreteMetaClasses,
    numMetaFeatures, numReferences, numAttributes,
53     numContainmentReferences,
    numNonContainmentReferences,
54     avgMetaFeaturesPerClass, avgReferencesPerClass,
    avgAttributesPerClass,
55     medMetaFeaturesPerClass,
    medAllMetaFeaturesPerClass,
56     medReferencesPerClass, medAllReferencesPerClass,
57     medAttributesPerClass, medAllAttributesPerClass,
58     numClassesWithNoImmediateFeatures,
    numClassesWithNoFeaturesAtAll,
59     numAbstractClassesWithNoImmediateFeatures,
    numConcreteClassesWithNoImmediateFeatures,
60     numAbstractClassesWithNoFeaturesAtAll,
    numConcreteClassesWithNoFeaturesAtAll,
61     avgFeaturesExclFeatureless,
    avgReferencesExclFeatureless,
    avgAttributesExclFeatureless,
62     refUpperOne, refUpperMany, refUpperN
63     }) {
64     result = result + "," + s;

```

```

65 }
66
67 result.println();
68
69 /**
70  * Calculates the median of a list of numbers.
71  */
72 operation Sequence median() : Real {
73   if (self.size() == 1) return self.at(0);
74
75   if (modulo(self.size(), 2) == 1) {
76     return self.at(self.size()/2);
77   } else {
78     var lower = self.at((self.size()/2)-1);
79     var upper = self.at(self.size()/2);
80     return (upper + lower) / 2.0;
81   }
82 }
83
84 /**
85  * Calculates the remainder of a modulo b.
86  */
87 operation modulo(a:Integer, b:Integer) : Integer {
88   return a - (a/b).floor() * b;
89 }

```

**Listing B.5:** *The EOL program used to analyse the structural properties of metamodels.*

## B.4 Metamodels

### B.4.1 EG-MOF

```

1 @namespace(uri="eg_mof", prefix="eg_mof")
2 package EG_MOF;
3
4 class ExecutionGraph {
5   !ordered val Node[#executionGraph] node;
6   !ordered ref Node[#nested] nodeNested;
7 }
8
9 class Node {
10  !ordered ref Node[#successor] predecessor;
11  !ordered ref Node[#predecessor] successor;
12  !ordered ref ExecutionGraph[#node] executionGraph;
13  !ordered ref ExecutionGraph[#nodeNested] nested;
14 }
15
16 class Start extends Node { }
17
18 class End extends Node { }
19
20 class Control extends Node { }
21
22 class Basic extends Node { }
23
24 class Branch extends Control { }
25
26 class Loop extends Control { }
27
28 class Fork extends Control { }
29
30 class Join extends Control { }

```

```

31 class Acquire extends Control { }
32
33 class Release extends Control { }
34
35 class Split extends Control { }
36

```

**Listing B.6:** *The EG-MOF metamodel defined in Emfatic notation.*

## B.4.2 Java

```

1 @namespace(uri="http://Java/1.0", prefix="Java")
2 package Java;
3
4 class Model {
5     attr PrimitiveTypes.String name;
6     val Element[*]#owningModel ownedMember;
7 }
8
9 abstract class Element {
10     !unique !ordered attr PrimitiveTypes.String[1] name;
11     ref Model#ownedMember owningModel;
12 }
13
14 class Package extends Element {
15     !ordered val Class[*]#~package classes;
16     !ordered val Enumeration[*]#~package enumerations;
17     !unique !ordered attr PrimitiveTypes.Boolean[1] isImported
18     ;
19 }
20
21 abstract class ClassMember extends Element {
22     !unique !ordered attr PrimitiveTypes.Boolean[1] isStatic;
23     !unique !ordered attr PrimitiveTypes.Boolean[1] isPublic;
24     !ordered ref Class[1]#members owner;
25     !ordered ref Type type;
26 }
27
28 class Field extends ClassMember {
29     !unique !ordered attr PrimitiveTypes.String initializer;
30 }
31
32 abstract class Type extends Element {
33 }
34
35 class Class extends Type {
36     !unique !ordered attr PrimitiveTypes.Boolean[1] isAbstract
37     ;
38     !unique !ordered attr PrimitiveTypes.Boolean[1] isPublic;
39     !unique !ordered attr PrimitiveTypes.Boolean[1]
40     isInterface;
41     !ordered ref Class[*] superClasses;
42     ref Class[*] actualTypeParameters;
43     !ordered ref Package[1]#classes ~package;
44     !ordered val ClassMember[*]#owner members;
45 }
46
47 class Method extends ClassMember {
48     !unique !ordered attr PrimitiveTypes.String[1] body;
49     val MethodParameter[*]#method parameters;
50     val Statement[*] statements;
51     ref Class[*] exceptions;
52 }
53
54 class PrimitiveType extends Type {
55 }

```

```

53 class Enumeration extends Type {
54   !ordered ref Package[1]#enumerations ~package;
55   val EnumerationLiteral[*]#enumeration enumerationLiterals;
56 }
57
58 class EnumerationLiteral extends Element {
59   !ordered ref Enumeration[1]#enumerationLiterals
60   enumeration;
61 }
62
63 class MethodParameter extends Element {
64   !ordered ref Type[1] type;
65   !ordered ref Method[1]#parameters method;
66 }
67
68 class Statement {
69   attr PrimitiveTypes.String[1] name;
70   attr PrimitiveTypes.String[1] body;
71 }
72
73 @namespace(uri="http://JavaPrimitiveTypes", prefix="
74   PrimitiveTypes")
75 package PrimitiveTypes {
76   datatype String : java.lang.String;
77
78   datatype Integer : java.lang.Integer;
79
80   datatype Boolean : boolean;
81 }

```

**Listing B.7:** *The Java metamodel defined in Emfatic notation.*

### B.4.3 ATOM

```

1 @namespace(uri="atom", prefix="atom")
2 package ATOM;
3
4 class ATOM {
5   !unique !ordered attr String[1] title;
6   !unique !ordered attr String[1] ~id;
7   !unique !ordered attr String subtitle;
8   !unique !ordered attr String rights;
9   !unique !ordered attr String icon;
10  !unique !ordered attr String logo;
11  !ordered val Link[+] links;
12  !ordered val Date[1] lastUpdate;
13  !ordered val Generator generator;
14  !ordered val Category[*] categories;
15  !ordered val Author[+] authors;
16  !ordered val Contributor[*] contributors;
17  !ordered val Entry[*]#atom entries;
18 }
19
20 class Entry {
21  !unique !ordered attr String[1] title;
22  !unique !ordered attr String[1] ~id;
23  !unique !ordered attr String rights;
24  !unique !ordered attr String summary;
25  !ordered val Link[+] links;
26  !ordered val Source source;
27  !ordered val Date published;
28  !ordered val Date[1] lastUpdate;
29  !ordered val Content content;
30  !ordered val Category[*] categories;

```

```

31  !ordered val Author[+] authors;
32  !ordered val Contributor[*] contributors;
33  !ordered ref ATOM[1]#entries atom;
34  }
35
36  class Source {
37    !unique !ordered attr String ~id;
38    !unique !ordered attr String icon;
39    !unique !ordered attr String logo;
40    !unique !ordered attr String rights;
41    !unique !ordered attr String title;
42    !unique !ordered attr String subtitle;
43    !ordered val Link[*] links;
44    !ordered val Date lastUpdate;
45    !ordered val Generator generator;
46    !ordered val Contributor[*] contributors;
47    !ordered val Category[*] categories;
48    !ordered val Author author;
49  }
50
51  class Content {
52    !unique !ordered attr String type;
53  }
54
55  class InLineXHTMLContent extends Content {
56  }
57
58  class InLineOtherContent extends Content {
59  }
60
61  class OutOfLineContent extends Content {
62    !unique !ordered attr String[1] src;
63  }
64
65  class Generator {
66    !unique !ordered attr String uri;
67    !unique !ordered attr String version;
68  }
69
70  class Category {
71    !unique !ordered attr String[1] term;
72    !unique !ordered attr String scheme;
73    !unique !ordered attr String label;
74  }
75
76  class Link {
77    !unique !ordered attr String[1] href;
78    !unique !ordered attr String rel;
79    !unique !ordered attr String type;
80    !unique !ordered attr String hreflang;
81    !unique !ordered attr String title;
82    !unique !ordered attr int lenght;
83  }
84
85  class Person {
86    !unique !ordered attr String[1] name;
87    !unique !ordered attr String uri;
88    !unique !ordered attr String email;
89  }
90
91  class Author extends Person {
92  }
93
94  class Contributor extends Person {
95  }
96
97  class Date {
98    !unique !ordered attr int[1] day;

```

```

99 !unique !ordered attr int[1] month;
100 !unique !ordered attr int[1] year;
101 !unique !ordered attr int[1] hours;
102 !unique !ordered attr int[1] minutes;
103 !unique !ordered attr int[1] seconds;
104 }

```

**Listing B.8:** *The ATOM metamodel defined in Emfatic notation.*

#### B.4.4 DslModel

```

1 @namespace(uri="dslmodel", prefix="dslmodel")
2 package DSLModel;
3
4 class Model {
5     !unique !ordered attr String domainModel;
6     val ModelElement[*] contents;
7 }
8
9 abstract class Element {
10     !unique !ordered attr String type;
11     !unique !ordered attr String ~id;
12 }
13
14 class ModelElement extends Element {
15     !ordered ref EmbeddingLink[1]#elements parentLink;
16     !ordered val Property[*]#owner properties;
17     !ordered val EmbeddingLink[*]#owner embeddinglinks;
18     !ordered val ReferenceLink[*]#owner referencelinks;
19 }
20
21 class ModelElementLink extends ModelElement {
22     !ordered ref ReferenceLink[*]#modelElement links;
23 }
24
25 class EmbeddingLink extends NamedElement {
26     !ordered ref ModelElement#embeddinglinks owner;
27     !ordered ref ModelElement[*]#parentLink elements;
28 }
29
30 class ReferenceLink extends Element {
31     !ordered ref ModelElement#referencelinks owner;
32     !ordered ref ModelElementLink[1]#links modelElement;
33     !ordered val Role[2]#owner roles;
34 }
35
36 abstract class NamedElement {
37     !unique !ordered attr String name;
38 }
39
40 class Property extends NamedElement {
41     !ordered ref ModelElement[1]#properties owner;
42     !ordered val Value[1] value;
43 }
44
45 class Role extends NamedElement {
46     !ordered ref ModelElement[1] element;
47     !ordered ref ReferenceLink[1]#roles owner;
48 }
49
50 abstract class Value {
51 }
52
53 class IntegerValue extends Value {
54     !unique !ordered attr int value;
55 }

```



```
56
57 class DoubleValue extends Value {
58     !unique !ordered attr double value;
59 }
60
61 class BooleanValue extends Value {
62     !unique !ordered attr boolean value;
63 }
64
65 class StringValue extends Value {
66     !unique !ordered attr String value;
67 }
```

**Listing B.9:** *The DslModel metamodel defined in Emfatic notation.*



# *VI*

## *References*



# Bibliography

- [1] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Kozi-olek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.
- [2] 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. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
- [3] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In *Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735, pages 436–450, Nashville, USA, 2007. Springer.
- [4] Thorsten Arendt, Pawel Stepien, and Gabriele Taentzer. Emf metrics: Specification and calculation of model metrics within the eclipse modeling framework. In *Proceedings of the 9th Belgian-Netherlands software eVOLution seminar (BENEVOL 2010)*, December 2010.
- [5] Marco Autili, Vittorio Cortellessa, Davide Di Ruscio, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. EA-GLE: engineering software in the ubiquitous globe by leveraging uncertainty. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11*, pages 488–491, New York, NY, USA, 2011. ACM.
- [6] Zahra Naji Azimi. Comparison of metaheuristic algorithms for examination timetabling problem. *Journal of Applied Mathematics and Computing*, 16(1-2):337–354, 2004.
- [7] Anthony J. Bagnall, Victor J. Rayward-Smith, and Ian Whitley. The next release problem. *Information and Software Technology*, 43(14):883–890, 2001.
- [8] Andras Balogh and Daniel Varro. Advanced model transformation language constructs in the VIATRA2 framework.

- In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1280–1287, New York, NY, USA, 2006. ACM Press.
- [9] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Software Testing, Verification & Reliability*, 15(2):73–96, June 2005.
- [10] Ameni. ben Fadhel, Marouane. Kessentini, Philip. Langer, and Manuel. Wimmer. Search-based detection of high-level model changes. In *Proceedings of 28th IEEE International Conference on Software Maintenance*, pages 212–221. IEEE Computer Society, 2012.
- [11] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003.
- [12] Daniel Boutros. Gamasutra: Difficulty is difficult: Designing for hard modes in games. [http://www.gamasutra.com/view/feature/3787/difficulty\\_is\\_difficult\\_designing\\_.php](http://www.gamasutra.com/view/feature/3787/difficulty_is_difficult_designing_.php), August 2012.
- [13] Michael Bowman, Lionel C. Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transactions on Software Engineering*, 36(6):817–837, 2010.
- [14] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.
- [15] Anthony Brabazon and Michael O’Neill. Evolving technical trading rules for spot foreign-exchange markets using grammatical evolution. *Computational Management Science*, 1:311–327, 2004.
- [16] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan and Claypool, 2012.
- [17] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. In *Computer*, volume 20, pages 10–19. IEEE Computer Society Press, April 1987.
- [18] Frank R. Burton, Richard F. Paige, Louis M. Rose, Dimitrios S. Kolovos, Simon Poulding, and Simon Smith. Solving acquisition problems using model-driven engineering. In *Proceedings of the 8th European conference on Modelling Foundations and Applications, ECMFA’12*, pages 428–443, Berlin, Heidelberg, 2012. Springer-Verlag.

- [19] Frank R. Burton and Simon Poulding. Complementing metaheuristic search with higher abstraction techniques. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [20] Juan Cadavid, Benoit Baudry, and Benoit Combemale. Empirical evaluation of the conjunct use of MOF and OCL. In *Proceedings of 1st International Workshop on Experiences and Empirical Studies in Software Modelling*, October 2011.
- [21] Juan Cadavid, Benoit Baudry, and Houari Sahraoui. Searching the boundaries of a modeling space to test metamodels. In *Proceedings of 5th IEEE International Conference on Software Testing, Verification and Validation*, pages 131–140, Washington, DC, USA, 2012. IEEE Computer Society.
- [22] Asli Celikyilmaz and I. Burhan Turksen. *Modeling Uncertainty with Fuzzy Logic*, volume 240 of *Studies in Fuzziness and Soft Computing*. Springer, 2009.
- [23] Betty H.C. Cheng. Harnessing evolutionary computation to enable dynamically adaptive systems to manage uncertainty. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [24] Betty H.C. H. Cheng, Rogério. Lemos, Holger. Giese, Paola. Inverardi, and Jeff. Magee. Software engineering for self-adaptive systems: A research roadmap. In Rogério Lemos, Holger Giese, HausiA. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, 2009.
- [25] S. C. Chiam, K.C. Tan, C.K. Goh, and A. Al Mamun. Improving locality in binary representation via redundancy. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 38(3):808–825, 2008.
- [26] Francisco Chicano, Marco Ferreira, and Enrique Alba. Comparing metaheuristic algorithms for error detection in java programs. In *Proceedings of the Third international conference on Search based software engineering, SSBSE'11*, pages 82–96, Berlin, Heidelberg, 2011. Springer-Verlag.
- [27] Hyun Cho and Jeff Gray. Design patterns for metamodels. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE'11, AOOPEs'11, NEAT'11, and VMIL'11, SPLASH'11 Workshops*, pages 25–32, New York, NY, USA, 2011. ACM.

- [28] John A. Clark, Jose J. Dolado, Mark Harman, Robert M. Hierons, Bryan F. Jones, M. Lumkin, Brian S. Mitchell, Spiros Mancoridis, K. Rees, Marc Roper, and Martin J. Shepperd. Reformulating software engineering as a search problem. In *IEEE Software*, volume 150, pages 161–175, 2003.
- [29] C. J. Clopper and E. S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.
- [30] Thelma E. Colanzi and Silvia R. Vergilio. Representation of software product line architectures for search-based design. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [31] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
- [32] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, 1993.
- [33] DBLP. Computer science bibliography. [www.dblp.org](http://www.dblp.org), 27 August 2013.
- [34] Kalyanmoy Deb, Amrit Pratap, Samir Agrawal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–187, 2002.
- [35] Osmar Marchi dos Santos, Jim Woodcock, and Richard F. Paige. Using model transformation to generate graphical counter-examples for the formal analysis of xUML models. In *Proceedings of 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 117–126. IEEE Computer Society, 2011.
- [36] Simon Easterbrooke, Marsha Chechik, et al. xCheck: A model check for multi-valued reasoning. In *Proceedings of 25th International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, USA, May 2003.
- [37] Sven Efftinge and Markus Voelter. oAW xText: A framework for textual DSLs. In *Eclipsecon Summit Europe*, 2006.
- [38] Maged Elaasar and Lionel C. Briand. An overview of UML consistency management. Technical Report SCE-04-18, Carleton University, August 2004.
- [39] Ramin Etemaadi, Michael T. M. Emmerich, and Michel R. V. Chaudron. Problem-specific search operators for meta-heuristic software architecture design. In *Proceedings of*



*the 4th international conference on Search Based Software Engineering, SSBSE'12*, pages 267–272, Berlin, Heidelberg, 2012. Springer-Verlag.

- [40] Michalis Famelis, Shoham Ben-David, Marsha Chechik, and Rick Salay. Partial models: a position paper. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVA*, pages 1:1–1:4, New York, NY, USA, 2011. ACM.
- [41] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: towards modeling and reasoning with uncertainty. In *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012*, pages 573–583, Piscataway, NJ, USA, 2012. IEEE Press.
- [42] Michalis Famelis, Rick Salay, Alessio Di Sandro, and Marsha Chechik. Transformation of models containing uncertainty. In Jeff Gray and Antonio Vallecillo, editors, *Proceedings of the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems*, 2013.
- [43] Martin Faunes, Houari Sahraoui, and Mounir Boukadoum. Generating model transformation rules from examples using an evolutionary algorithm. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, pages 250–253, New York, NY, USA, 2012. ACM.
- [44] Martin Faunes, Houari Sahraoui, and Mounir Boukadoum. Genetic-programming approach to learn model transformation rules from examples. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations*, volume 7909 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin Heidelberg, 2013.
- [45] Nicolas Ferry, Vincent Hourdin, Stephane Lavirotte, Gaetan Rey, Jean-Yves Tigli, and Michel Riveill. Models at runtime: Service for device composition and adaptation. In *Proceedings of the 4th International Workshop on Models at Runtime*, 2009.
- [46] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering: testing model transformations. In *Proceedings of the 1st International Workshop on Model-Driven Engineering, Verification and Validation*, pages 29–40, 2004.
- [47] Association for Computing Machinery. ACM Digital Library. <http://dl.acm.org/>, 27 August 2013.
- [48] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, Boston, MA, USA, 1999.

- [49] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.
- [50] Mārtiņš Francis, Dimitrios S. Kolovos, Nicholas Matragkas, and Richard F. Paige. Adding spreadsheets to the mde toolkit. In *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems*, 2013.
- [51] Franck Fleurey, Benoit Baudry, Robert France and Sudipto Ghosh. A Generic Approach For Automatic Model Composition. In *Proceedings of 11th International Workshop on Aspect-Oriented Modeling*, Nashville, USA, September 2007.
- [52] Edgar Galván-López, James Mcdermott, Michael O’Neill, and Anthony Brabazon. Defining locality as a problem difficulty measure in genetic programming. *Genetic Programming and Evolvable Machines*, 12(4):365–401, December 2011.
- [53] Antonio García-Domínguez, Dimitrios Kolovos, Louis Rose, Richard Paige, and Inmaculada Medina-Bulo. EUnit: A unit testing framework for model management tasks. In *Proceedings of the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, volume 6981, pages 395–409, Berlin, Heidelberg, 2011. Springer-Verlag.
- [54] Adnane Ghannem, Marouane Kessentini, and Ghizlane El Boussaidi. Detecting model refactoring opportunities using heuristic search. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, CASCON ’11*, pages 175–187, Riverton, NJ, USA, 2011. IBM Corp.
- [55] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8(1):21–43, 2009.
- [56] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.
- [57] David E. Goldberg. *The Design of Innovation: Lessons from and for competent genetic algorithms*. Kluwer Academic Publishers, 2002.
- [58] Heather J. Goldsby and Betty H. C. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, pages 568–583, Berlin, Heidelberg, 2008. Springer-Verlag.

- [59] Heather J. Goldsby and Betty H. C. Cheng. Avida-MDE: a digital evolution approach to generating models of adaptive software behavior. *Proceedings of the 10th Annual conference on Genetic and Evolutionary Computation*, pages 1751–1758, 2008.
- [60] Network Working Group. The Atom Syndication Format, RFC 4287. <http://tools.ietf.org/html/rfc4287>, December 2005.
- [61] The Object Management Group. UML 2.0 OCL Specification. <http://www.omg.org/docs/ptc/03-10-14.pdf>.
- [62] Mark Harman. The current state and future of search based software engineering. In *Proceedings of 2007 Future of Software Engineering*, pages 342–357, 2007.
- [63] Mark Harman. Search based software engineering for program comprehension. In *15th IEEE International Conference on Program Comprehension*, pages 3–13, 2007.
- [64] Mark Harman and John Clark. Metrics are fitness functions too. In *Proceedings of the Software Metrics, 10th International Symposium, METRICS '04*, pages 58–69, Washington, DC, USA, 2004. IEEE Computer Society.
- [65] Mark Harman and Bryan F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.
- [66] Mark Harman, Jens Krinke, Jian Ren, and Shin Yoo. Search based data sensitivity analysis applied to requirement engineering. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation, GECCO '09*, pages 1681–1688, New York, NY, USA, 2009. ACM.
- [67] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, December 2012.
- [68] Mark Harman, Richard F. Paige, and James Williams. 1st International Workshop on Combining Modelling and Search-based Software Engineering (CMSBSE 2013). In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 1513–1514, Piscataway, NJ, USA, 2013. IEEE Press.
- [69] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO '07*, pages 1106–1113, New York, NY, USA, 2007. ACM.

- [70] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2009.
- [71] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE - automating coupled evolution of meta-models and models. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 52–76, Berlin, Heidelberg, 2009. Springer-Verlag.
- [72] Ernest Friedman Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.
- [73] John H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.
- [74] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, September 2003.
- [75] Jonatan Hugosson, Erik Hemberg, Anthony Brabazon, and Michael O’Neill. Genotype representations in grammatical evolution. *Journal of Applied Soft Computing*, 10(1):36–43, January 2010.
- [76] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11*, pages 633–642, New York, NY, USA, 2011. ACM.
- [77] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11*, pages 471–480, New York, NY, USA, 2011. ACM.
- [78] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.
- [79] Frederic Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of the Satellite Events at the 2005 International conference on Modeling Languages and Systems*, volume 3844 of *Lecture Notes in Computer Science*, pages 129–138, 2006.
- [80] Thomas E. Kammeyer and Richard K. Belew. Stochastic context-free grammar induction with a genetic algorithm using local search. *Foundations of Genetic Algorithms IV*, 1996.

- [81] Horst Kargl, Michael Strommer, and Manuel Wimmer. Measuring the explicitness of modeling concepts in meta-models. In *Proceedings of the 2006 Workshop on Model Size Metrics*, October 2006.
- [82] Hillol Kargupta. A striking property of genetic code-like transformations. *Complex Systems*, 11, 2001.
- [83] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
- [84] Marouane Kessentini, Philip Langer, and Manuel Wimmer. Searching Models, Modeling Search: On the Synergies of SBSE and MDE. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [85] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. *LNCS*, 5301:159–173, 2008.
- [86] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Example-based model-transformation testing. *Automated Software Engineering*, 18(2):199–224, 2011.
- [87] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar ben Omar. Search-based model transformation by example. *Software and Systems Modeling*, pages 1–18, 2010.
- [88] Marouane Kessentini, Wafa Werda, Philip Langer, and Manuel Wimmer. Search-based model merging. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference, GECCO '13*, pages 1453–1460, New York, NY, USA, 2013. ACM.
- [89] Donald Knuth, James Morris, Jr., and Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [90] Dimitrios S. Kolovos. Establishing correspondences between models with the Epsilon Comparison Language. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 146–157. Springer Berlin Heidelberg, 2009.
- [91] Dimitrios S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, 2009.
- [92] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In *Model Driven*

*Architecture — Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

- [93] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Detecting and repairing inconsistencies across heterogeneous models. In *1st IEEE Conference on Software Testing, Verification, and Validation*, pages 356–364, 2008.
- [94] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Transformation Language. In *Proceedings of the 2008 International Conference on the Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.
- [95] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Merging models with the Epsilon Merging Language (EML). In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin Heidelberg, 2006.
- [96] Dimitrios S. Kolovos, Louis M. Rose, Saad Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. Taming EMF and GMF using model transformation. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 211–225, Berlin, Heidelberg, 2010. Springer-Verlag.
- [97] Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Antonio Garcia-Dominguez. *The Epsilon book*. Unpublished., 2010.
- [98] Dimitrios S. Kolovos, Louis M. Rose, James R. Williams, Nicholas Matragkas, and Richard F. Paige. A lightweight approach for managing XML documents with MDE languages. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos, editors, *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin Heidelberg, 2012.
- [99] Christian Lange, Michel R. V. Chaudron, Johan Muskens, and H. M. Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In *Proceedings of 2nd Workshop on Consistency Problems in UML-based Software Development*, 2003.
- [100] Grzegorz Lehmann, Marco Blumendorf, Frank Trollmann, and Sahin Albayrak. Meta-modeling runtime models. In

Juergen Dingel and Arnor Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 209–223. Springer Berlin Heidelberg, 2011.

- [101] A.M. Leite da Silva, L.S. Rezende, L.M. Honório, and L.A.F. Manso. Performance comparison of metaheuristics to solve the multi-stage transmission expansion planning problem. *Generation, Transmission Distribution, IET*, 5(3):360–367, 2011.
- [102] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, pages 5–14, 2003.
- [103] Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at <http://cs.gmu.edu/~sean/book/metaheuristics/>.
- [104] Zhiyi Ma, Xiao He, and Chao Liu. Assessing the quality of metamodels. *Frontiers of Computer Science*, pages 1–13, 2013.
- [105] Philip K. McKinley, Betty H.C. Cheng, Andres J. Ramirez, and Adam C. Jensen. Applying evolutionary computation to mitigate uncertainty in dynamically-adaptive, high-assurance middleware. *Journal of Internet Services and Applications*, 3(1):51–58, 2012.
- [106] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.
- [107] Phil McMinn. Search-based software testing: Past, present and future. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, Washington, DC, USA, March 2011. IEEE Computer Society.
- [108] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
- [109] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, second edition, 1997.
- [110] Julian F. Miller. *Cartesian Genetic Programming*. Natural Computing. Springer-Verlag, Berlin, Heidelberg, 2011.
- [111] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April 2006.
- [112] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B.

- Langdon, Julian Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132. Springer Berlin Heidelberg, 2000.
- [113] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - a review of experiences from applying MDE in industry. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer Berlin Heidelberg, 2008.
- [114] Slim Bechikh Mohamed Mkaouer, Marouane Kessentini and Daniel Tauritz. Preference-based multi-objective software modelling. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [115] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michele Soria. Uniform random generation of huge metamodel instances. In *Proceedings of 5th European Conference on Model Driven Architecture – Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 130–145, Berlin, Heidelberg, 2009. Springer-Verlag.
- [116] Raymond H. Myers, Douglas C. Montgomery, and Christine M. Anderson-Cook. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley, 3rd edition, 2009.
- [117] Francisco Gomez Oliveira Neto, Robert Feldt, Richard Torkar, and Patricia Machado. Searching for models to test software technology. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [118] Linda Northrop et al. Ultra-large scale systems: The software challenge of the future. Technical report, Carnegie Mellon, June 2006.
- [119] The Object Management Group. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*, June 2001.
- [120] The Object Management Group. *Meta Object Facility (MOF) Core Specification*, January 2006.
- [121] The Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation v1.0*, April 2008.
- [122] The Object Management Group. *The Unified Modeling Language Specification*, August 2011.



- [123] Charles Ofria and Claus O. Wilke. Avida: a software platform for research in computational evolutionary biology. *Artificial Life*, 10(2):191–229, April 2004.
- [124] Mark O’Keeffe and Mel O Cinneide. Towards automated design improvement through combinatorial optimisation. In *Proceedings of the 1st Workshop on Directions in Software Engineering Environments*, 2004.
- [125] Michael O’Neill and Anthony Brabazon. Grammatical differential evolution. In *Proceedings of 2006 International Conference Artificial Intelligence*, volume 1, pages 231–236, Las Vegas, Nevada, USA, 2006. CSREA Press.
- [126] Michael O’Neill and Anthony Brabazon. Grammatical swarm: The generation of programs by social programming. *Natural Computing*, 5:443–462, 2006.
- [127] Michael O’Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions Evolutionary Computation*, 5(4):349–358, 2001.
- [128] Richard F. Paige. Case study : Airport crisis management. Large Scale Complex IT Systems Workshop, 2010.
- [129] Richard F. Paige, Phillip J. Brooke, Xiaocheng Ge, Christopher D.S. Power, Frank R. Burton, and Simon Poulding. Revealing complexity through domain-specific modelling and analysis. In Radu Calinescu and David Garlan, editors, *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539 of *Lecture Notes in Computer Science*, pages 251–265. Springer Berlin Heidelberg, 2012.
- [130] Richard F. Paige, Phillip J. Brooke, and Jonathan S. Ostroff. Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology*, 16(3), July 2007.
- [131] Richard F. Paige and Louis M. Rose. Lies, Damned Lies and UML2Java. <http://blog.jot.fm/2013/01/25/lies-damned-lies-and-uml2java/>, January 2013.
- [132] Massimiliano Penta. SBSE meets software maintenance: Achievements and open problems. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 27–28. Springer Berlin Heidelberg, 2012.
- [133] Marian Petre. UML in practice. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE ’13*, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.
- [134] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

- [135] Outi Raiha. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.
- [136] Andres J. Ramirez and Betty H. C. Cheng. Evolving models at run time to address functional and non-functional adaptation requirements. In *Proceedings of the 4th International Workshop on Models at Runtime*, 2009.
- [137] Andres J. Ramirez, Erik M. Fredericks, Adam C. Jensen, and Betty H.C. Cheng. Automatically RELAXing a goal model to cope with uncertainty. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 198–212. Springer Berlin Heidelberg, 2012.
- [138] Andres J. Ramirez, David B. Knoester, Betty H.C. Cheng, and Philip K. McKinley. Applying genetic algorithms to decision making in autonomic computing systems. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 97–106, New York, NY, USA, 2009. ACM.
- [139] Sam Ratcliff, David R. White, and John A. Clark. Searching for invariants using genetic programming and mutation testing. In *Proceedings of the 13th International Conference on Genetic and Evolutionary Computation*, pages 1907–1914, New York, NY, USA, 2011. ACM.
- [140] Mark Read. *Statistical and Modelling Techniques to Build Confidence in the Investigation of Immunology through Agent-Based Simulation*. PhD thesis, University of York, 2011.
- [141] Guenther Ruhe Reza Karimpour. Bi-criteria genetic search for adding new features into an existing product line. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [142] Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James R. Williams, Richard F. Paige, Fiona A. C. Polack, and Keiran J. Fernandes. Concordance: A framework for managing model integrity. In *Proceedings of 6th European Conference on Modelling Foundations and Applications*, Berlin, Heidelberg, June 2010. Springer-Verlag.
- [143] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. The Epsilon Generation Language. In *Proceedings 4th European Conference on Model Driven Architecture Foundations and Applications*, Berlin, Heidelberg, June 2008. Springer-Verlag.
- [144] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. Constructing models with the human-usable textual notation. In Krzysztof Czarnecki, Ileana

Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 249–263. Springer Berlin Heidelberg, 2008.

- [145] Louis M. Rose and Simon Poulding. Efficient probabilistic testing of model transformations. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [146] Olivia Rossi-Doria, Michael Sampels, Mauro Birattari, Marco Chiarandini, Marco Dorigo, Luca M. Gambardella, Joshua Knowles, Max Manfrin, Monaldo Mastrolilli, Ben Paechter, Luis Paquete, and Thomas Stützle. A comparison of the performance of different metaheuristics on the timetabling problem. In Edmund Burke and Patrick Causmaecker, editors, *Practice and Theory of Automated Timetabling IV*, volume 2740 of *Lecture Notes in Computer Science*, pages 329–351. Springer Berlin Heidelberg, 2003.
- [147] Franz Rothlauf. *Representations for Evolutionary and Genetic Algorithms*. Springer Berlin Heidelberg New York, second edition, 2006.
- [148] Franz Rothlauf. *Design of Modern Heuristics: Principles and Application*. Natural Computing. Springer Berlin Heidelberg New York, 2011.
- [149] Franz Rothlauf and David E. Goldberg. Pruefer numbers and genetic algorithms: A lesson on how the low locality of an encoding can harm the performance of gas. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature, PPSN VI*, pages 395–404, London, UK, UK, 2000. Springer-Verlag.
- [150] Franz Rothlauf and Marie Oetzel. On the locality of grammatical evolution. In *Proceedings of the 9th European conference on Genetic Programming, EuroGP'06*, pages 320–330, Berlin, Heidelberg, 2006. Springer-Verlag.
- [151] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution : Evolving programs for an arbitrary language. In *Proceedings 1st European Workshop on Genetic Programming (EuroGP'98)*, volume 1391 of *LNCS*, pages 83–96. Springer, 1998.
- [152] Rick Salay, Marsha Chechik, and Jan Gorzny. Towards a methodology for verifying partial model refinements. In *Proceedings of IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 938–945, Washington, DC, USA, 2012. IEEE Computer Society.

- [153] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering, FASE'12*, pages 224–239, Berlin, Heidelberg, 2012. Springer-Verlag.
- [154] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009.
- [155] A. Saltelli, K. Chan, and E. M. Scott, editors. *Sensitivity Analysis*. Probability and Statistics. Wiley, 2000.
- [156] Luis Emiliano Sanchez, Sabine Moisan, and Jean-Paul Rigault. Metrics on feature models to optimize configuration adaptation at run time. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [157] Mario Sánchez, Iván Barrero, Jorge Villalobos, and Dirk Deridder. An Execution Platform for Extensible Runtime Models. In Nelly Bencomo, Gordon Blair, Robert France, Freddy Muñoz, and Cedric Jeanneret, editors, *Proceedings of the 3rd International Workshop on Models at Runtime*, 2008.
- [158] Abdel Salam Sayyad, Josef Ingram, Tim Menzies, and Hany Ammar. Optimum feature selection in software product lines. In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMSBSE)*, 2013.
- [159] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.
- [160] Daniel Schneider and Martin Becker. Runtime models for self-adaptation in the ambient assisted living domain. In *Proceedings of the 3rd International Workshop on Models at Runtime*, 2008.
- [161] Tom Seaton, Julian F. Miller, and Tim Clarke. An ecological approach to measuring locality in linear genotype to phenotype maps. In *Proceedings of the 15th European conference on Genetic Programming, EuroGP'12*, pages 170–181, Berlin, Heidelberg, 2012. Springer-Verlag.
- [162] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.
- [163] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

- [164] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, pages 42–45, 2003.
- [165] Jianhua Shao, James McDermott, Michael O’Neill, and Anthony Brabazon. Jive: A generative, interactive, virtual, evolutionary music system. In *Applications of Evolutionary Computation*, volume 6025 of *LNCS*, pages 341–350. Springer, 2010.
- [166] Christopher L. Simons. Whither (away) software engineerings in sbse? In M. Harman, R. F. Paige, and J. R. Williams, editors, *Proceedings of the 1st International Workshop on Combining Modelling and Search Based Software Engineering (CMS-BSE)*, 2013.
- [167] Christopher L. Simons, Ian C. Parmee, and Rhys Gwynlyw. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Transaction Software Engineering*, 36(6):798–816, November 2010.
- [168] Steven. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London Limited, second edition, 2008.
- [169] Steven. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London Limited, second edition, 2008.
- [170] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, second edition, 2009.
- [171] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In Gregor Engels, Bill Opdyke, DouglasC. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2007.
- [172] Paul Suddaby. Unknown mode: Difficulty in open-ended game design. <http://gamedev.tutsplus.com/articles/game-design-articles/unknown-mode-difficulty-in-open-ended-game-design/>, March 2013.
- [173] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE ’07*, pages 295–304, New York, NY, USA, 2007. ACM.
- [174] U.S. Department of Defense. *The Department of Defence Architecture Framework Version 2.02*.

- [175] Eric Vépa, Jean Bézivin, Hugo Brunelière, and Frederic Jouault. Measuring model repositories. In *Proceedings of the 1st Workshop on Model Size Metrics*, October 2006.
- [176] Andrew Watson. A brief history of MDA. *Upgrade, the European Journal for the Informatics Professional*, 9(2), 2008.
- [177] Thomas Weigert. Practical experiences in using model-driven engineering to develop trustworthy computing systems. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, volume 1 of *SUTC '06*, pages 208–217, Washington, DC, USA, 2006. IEEE Computer Society.
- [178] Tony White, Jinfei Fan, and Franz Oppacher. Basic object oriented genetic programming. In *Proceedings of the 24th international conference on Industrial engineering and other applications of applied intelligent systems conference on Modern approaches in applied intelligence - Volume Part I, IEA/AIE'11*, pages 59–68, Berlin, Heidelberg, 2011. Springer-Verlag.
- [179] James R. Williams, Frank R. Burton, Richard F. Paige, and Fiona A. C. Polack. Sensitivity analysis in model-driven engineering. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 732–758, Berlin, Heidelberg, 2012. Springer-Verlag. Submitted to MODELS 2012.
- [180] James R. Williams, Dimitrios S. Kolovos, Fiona A. C. Polack, and Richard F. Paige. Requirements for a model comparison language. In *Proceedings of the 2nd International Workshop on Model Comparison in Practice, IWMCP '11*, pages 26–29, New York, NY, USA, 2011. ACM.
- [181] James R. Williams, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. Search-based model driven engineering. Technical Report YCS-2012-475, Department of Computer Science, University of York, 2012.
- [182] James R. Williams, Richard F. Paige, and Fiona A.C. Polack. Searching for model migration strategies. In *Proceedings of 2012 International Workshop on Models and Evolution*, Innsbruck, Austria, September 2012.
- [183] James R. Williams and Fiona A.C. Polack. Automated formalisation for verification of diagrammatic models. *Electronic Notes in Theoretical Computer Science*, 263:211 – 226, 2010.
- [184] James R. Williams and Simon Poulding. Generating models using metaheuristic search. In *Proceedings of the 4th York Doctoral Symposium on Computing*, York, UK, October 2011.

- [185] James R. Williams, Simon Poulding, Richard F. Paige, and Fiona A.C. Polack. Exploring the use of metaheuristic search to infer models of dynamic system behaviour. In *Proceedings of 8th International Workshop on Models at Runtime*, September 2013.
- [186] James R. Williams, Simon Poulding, Louis M. Rose, Richard F. Paige, and Fiona A. C. Polack. Identifying desirable game character behaviours through the application of evolutionary algorithms to model-driven engineering meta-models. In *Proceedings of the 3rd International Conference on Search Based Software Engineering (SSBSE'11)*, volume 6956 of *Lecture Notes in Computer Science*, pages 112–126, Berlin, Heidelberg, 2011. Springer-Verlag.
- [187] James R. Williams, Athanasios Zolotas, Nicholas Matragkas, Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. What do metamodels really look like? In *Proceedings of 3rd International Workshop on Experiences and Empirical Studies in Software Modelling (EESS-MOD 2013)*, 2013.
- [188] Peter Wyard. Context free grammar induction using genetic algorithms. In *Grammatical Inference: Theory, Applications and Alternatives, IEE Colloquium on*, pages P11/1 – P11/5, April 1993.
- [189] Tina Yu and Julian Miller. Neutrality and the evolvability of boolean function landscape. In Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G.B. Tetamanzi, and William B. Langdon, editors, *Genetic Programming*, volume 2038 of *Lecture Notes in Computer Science*, pages 204–217. Springer Berlin Heidelberg, 2001.
- [190] Tina Yu and Julian F. Miller. Finding needles in haystacks is not hard with neutrality. In *Proceedings of the 5th European Conference on Genetic Programming, EuroGP '02*, pages 13–25, London, UK, 2002. Springer-Verlag.
- [191] Yuehua Lin and Jeff Gray and Frederic Jouault. DSMDiff: A Differentiation Tool for Domain-Specific Models. *European Journal of Information Systems*, 16(4):349–361, August 2007. (Special Issue on Model-Driven Systems Development).
- [192] Hadar Ziv, Debra J. Richardson, and Rene Klösch. The uncertainty principle in software engineering. In *Proceedings of 19th International Conference on Software Engineering*, 1997.