Search-Based Model-Driven Engineering

James R. Williams, Simon Poulding, Richard F. Paige, Dimitrios S. Kolovos, Fiona A. C. Polack
Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK.
{jlw,smp,dkolovos,paige,fiona}@cs.york.ac.uk

Abstract—Model-Driven Engineering (MDE) is a software engineering approach that treats models as the primary development artefacts. Models are manipulated, using model management operations (MMO), in order to perform engineering tasks such as comparison, validation, and code generation. We propose that many of the challenges faced in MDE could be addressed using Search-Based Software Engineering (SBSE) techniques: the two approaches, MDE and SBSE, are natural partners. However, a significant barrier preventing widespread application of SBSE to MDE is a suitable representation of MDE models that is amenable to metaheuristic search algorithms while also applicable across a wide range of MDE problem domains. This paper presents such a representation and describes an implementation using a widely used MDE toolset. We illustrate the utility of the representation by applying it to the white-box analysis of three MMOs. Furthermore, we discuss other areas of MDE which could benefit from the application of SBSE techniques, and highlight research opportunities for combining the two fields.

1 INTRODUCTION

Model-Driven Engineering (MDE) [31], [9] is a software engineering practice which uses disparate models of systems and software to (often automatically) produce an end product. Models are treated as the primary development artefacts and are manipulated by model management operations (MMO), such as model comparison, model merging, model validation, and perhaps most commonly, model transformation. Since MDE aims to raise the level of abstraction at which developers work, it faces its own unique set of challenges alongside many of those found in traditional software engineering.

Search-based software engineering (SBSE) [14], [6], [13] 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. In this situation, metaheuristic search algorithms may be used to efficiently locate (near) optimal solutions by assessing how close candidate solutions are to solving the problem and using this information to guide the search over the space of all possible solutions. 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 wide variety of software engineering problems [14].

In this paper we motivate the case for applying SBSE techniques to challenges faced in MDE. Since the solution to many of these challenges takes the form of a model, we propose a search-amenable representation that encodes the important features of the model space and therefore permits the search algorithms typically used in SBSE to be applied to these MDE challenges. The paper is structured as follows. Section 2 overviews MDE and SBSE, and illustrates the areas of MDE that could benefit from SBSE techniques. Section 3 presents our proposed representation of MDE models. Section 4 discusses our implementation of the representation for a widely used MDE toolset, and illustrates its utility by using it to perform coverage analysis on three MMOs. Section 5 outlines many opportunities where SBSE techniques can provide many benefits to MDE. Finally, section 6 concludes by inspecting the limitations of the representation.

2 SEARCH-BASED AND MODEL-DRIVEN

This section overviews the key concepts in MDE and SBSE before illustrating how SBSE can help to address some of the challenges faced in MDE.

2.1 Model-Driven Engineering

MDE aims to improve important aspects of software engineering, such as productivity, maintainability, and interoperability. Hutchinson et al. empirically evaluate the use of MDE in industry and provide evidence for some of these perceived benefits, whilst noting the need for appropriate supporting processes and techniques [17]. MDE prescribes models to be first class citizens in the software development process [31], [9]. The goal of MDE is to model systems at the level of the application domain, and apply a series of automatic transformations to produce code. Raising the level of abstraction can reduce the complexity of the problem, whilst also allowing domain experts to become the primary developers of systems; software engineers produce the modelling languages and infrastructures that support the domain experts [31], [32].
Models in MDE are said to conform to a metamodel – another model that describes the concepts that a model can use, and any constraints over those concepts. Metamodels can be viewed as the grammar, or abstract syntax, for modelling languages. The concrete syntax of a model is commonly implemented graphically, but can be expressed textually as well, and there are many existing tools that aid in the development of concrete syntaxes (e.g. GMF\(^1\) and Graphiti\(^2\) for graphical, and Xtext\(^3\) for textual syntaxes).

To clarify these concepts, we present an illustrative example of a metamodel and model. Figure 1 shows a metamodel that describes a simple language for modelling the placement of animals in a zoo, and Figure 2 shows an example model conforming to this metamodel, represented as an object diagram.

Disparate models have different uses, which can be broadly categorised into two classes: development models and runtime models [9]. Development models may be used for abstractly modelling the implementation before generating code via a series of automated transformations. Alternatively, they may simply be used for problem illustration or analysis. Runtime models capture information about an executing system, enabling software to be dynamically adapted.

There are many kinds of model transformations; the two common classes being model-to-model and model-to-text transformations [7]. Other kinds of model management operation include: comparison, e.g. to discover inconsistencies between two versions of a model; merging, e.g. to combine different views of a system into one model; validation, e.g. to ensure a model satisfies specific domain constraints. It is the combination of these operations into intricate workflows that enables large and complex systems to be developed more effectively.

### 2.2 Search-Based Software Engineering

In SBSE, software engineering problems are couched as optimisation problems, where the search space is the set of all possible solutions to the problem. The rationale behind SBSE is the observation that it is often easier to determine whether one solution is better than another, than to construct an optimal solution to a problem [14], [6], [13]. In order to successfully reformulate a software engineering problem as an optimisation problem, three tasks need to be considered:

1) Define a characterisation, or representation, of solutions that is amenable to search techniques (applicable where the native format of solutions is particularly complex) and can be easily mapped to the native format. The representation (commonly strings of bits or integers) is known as the genotype, and the native format is known as the phenotype.

2) Define a set of search operators – ways in which solutions are varied in order to effectively traverse the solution space.

3) Define an objective function to calculate the “fitness” of candidate solutions – a measure of how close the solution is to solving the desired problem.

The choice of representation and search operators are key to the efficacy of the search [29].

Metaheuristic search algorithms focus on identifying (near) optimal solutions to a problem by searching over the solution space – the set of possible solutions presented for consideration [6]. Metaheuristic search algorithms can be categorised into two kinds [6]: local search, which focuses on improving a single solution; and population-based search, which focuses on improving the fitness of a population of solutions.

#### 2.2.1 Local Search

The most basic local metaheuristic search algorithm is hill-climbing: the starting candidate solution is chosen at random, and its fitness is compared to each of its neighbours (those solutions deemed “close” to the current solution though application of a single search operator). If a fitter neighbour is found, it becomes the current candidate solution, and each of its neighbours is examined. In this way the search “climbs” the fitness landscape until no adjacent solution is fitter than the current candidate. Hill-climbing

---

1. www.eclipse.org/modeling/gmp
2. www.eclipse.org/graphiti
3. www.eclipse.org/Xtext
work well in simple fitness landscapes, but cannot escape from a local optimum in the case of a more rugged fitness landscape. This means that the selected solution may not be the global optimum. Extensions of the algorithm, such as random restart hill climbing, or techniques such as simulated annealing overcome the problem of local optima by allowing less-fit solutions to be adopted, with some probability, to allow the search to move away from local optima [6], [22].

2.2.2 Population-Based Search

Other metaheuristic search techniques focus on improving a population of solutions, as opposed to a single solution, whilst attempting to identify a globally optimal solution. The most widely known population-based algorithm is the genetic algorithm (GA) [6], [10], a type of evolutionary algorithm. Inspired by biological evolution, GAs select a population of candidate solutions which are combined and mutated to develop a new population. A solution is referred to as an individual, and is typically represented as a bit-string or a sequence of integers. Each component bit or integer is called a gene. Each gene encodes one or more aspects of the solution’s native form, the phenotype. When seeking an optimal solution, a GA is executed for many generations, each of which produces a new population. The fittest chromosomes from the previous generation are selected and, in order to maintain the diversity necessary to avoid local optima, are mutated and combined with other fit solutions. Often the fittest solutions, the elite, are kept unaltered in the new population in an attempt to increase the population’s overall fitness. 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.

2.3 Connecting SBSE with the Challenges Faced in MDE

The observation that premises SBSE is inherent in MDE. Developing an optimal model of a domain is 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, indicating that utilising SBSE methods could prove fruitful.

France and Rumpe [9] enumerate three categories of challenges faced in MDE: modelling language challenges; separation of concerns challenges; and, model manipulation and management challenges. We feel that SBSE techniques can be a positive influence in tackling, in particular, challenges in the latter category.

To present an overview of MDE issues and tasks related to model manipulation and management where SBSE techniques may help, we separate them into two groups and describe them here. Existing work that utilises SBSE techniques to address the problems below, will be presented in section 5.

2.3.1 Discovering and Validating Models

Modelling of any kind is subject to different levels of uncertainty, such as errors of measurement or interpretation, incomplete information, and poor or partial understanding of the domain [30]. When MMOs 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. There are many cases where we wish to find an optimal model of a certain domain. For instance, taking the example in Figure 2, we might want to find an optimal allocation of animals to cages such that no animal shares a cage with one that eats its – this is a variation of the bin packing problem. Model and metamodel discovery and validation, therefore, are key challenges in MDE.

2.3.2 Discovering and Validating MMOs

The uncertainty involved with developing models is also prevalent in the definition of MMOs. For example, one of the challenges found in MDE is that of metamodel evolution [15]. When a metamodel is updated to a new version, 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 1 abstract, and define two concrete subclasses, Mammal and Reptile. The model in Figure 2 no longer conforms to the metamodel, as abstract classes cannot be instantiated. We migrate models by applying a number of transformation rules, called a migration strategy, but it may not be obvious what the best strategy is. One solution may be to simply delete the Animal objects – this would result in a model that conforms to the new version of the metamodel, but has lost information. A better solution would be to convert the two Animal objects into instances of the Mammal class. Furthermore, there may exist numerous valid, semantic-preserving, migration strategies, some of which may be more efficient than others with respect to non-functional properties.

Validating a migration strategy (or any form of MMO) is a challenging problem. Where viewed from the perspective of testing traditional software, it can be seen that the test inputs (i.e., models) are extremely complex and this increases the difficulty of test data selection and the definition of an oracle [1], [2]. We believe that the body of knowledge from the search-based software testing field could be very useful in addressing this problem, as much work has looked at ways to improve traditional software testing [23], [24].
2.4 Summary and Motivation

This section has given an overview of MDE and some of the challenges faced by practitioners. For a comprehensive review of the state of MDE and its challenges, see [9], [2], [34]. If these challenges were couched as search problems, many of them (e.g. testing MMOs, discovering/optimising models) would share the same phenotypical form – models. Therefore, if we had a generic representation for models, we could use the wealth of existing search practices to help solve MDE problems. The next section introduces a generic, search-amenable representation for MDE models.

3 A Canonical Representation for Models

Clark et al. identify the process of reformulating a software engineering problem as a search problem as (a) finding a characterisation that enables solutions to the problem to be represented in a simple manner which is amenable to search; and (b), defining an objective function that measures how close a candidate solution is to solving the problem under scrutiny [6]. The objective function is normally specific to a single problem domain. However, there is an opportunity to define a canonical representation for MDE models that is suitable for use across a range of problem domains and is amenable to many of the search algorithms typically used in SBSE.

This section introduces our generic representation for MDE models, describing the structure of an individual, the genotype-phenotype mapping, the search operators and properties specific to the representation.

3.1 A Linear Representation for Models

Rothlauf [29] defines desirable characteristics of a representation for genetic and evolutionary algorithms, including: not encoding infeasible solutions; having low redundancy – reducing the number of elements which are unused in the mapping to a solution; and aiming for high locality – meaning that a small change to part of an individual will also result in a small change to the solution it represents.

Models in MDE are represented as object graphs. A direct representation, where the search space is the solution space (or equivalently using GA terminology, the genotype is the phenotype), would ensure an efficient encoding. However, significant work would be required to define custom search techniques and devise custom evolutionary operators to be used by the search algorithm. Furthermore, models can get very large, which could make search impractical due to the vast memory requirements in cases of large models and large populations. To alleviate these problems we propose using a linear representation of models. A linear genotype enables the use of many linear genotype-based search algorithms, such as genetic algorithms, evolutionary strategies, simulated annealing and hill climbing. This allows MDE practitioners to determine whether different algorithms perform better for different problems, and make use of the wealth of existing research in this area.

This section presents our linear representation of MDE models. Firstly, we introduce the finitisation model, which is used to declare the data available in the mapping process. We then explain the structure of an individual and describe the mapping from integer string to model. Finally, we discuss some of the interesting characteristics of the representation.

3.1.1 Metamodel Data Finitisation

When designing metamodels, it is common to use unbounded data types for attributes, or to assign one-to-many multiplicities to references instead of enforcing an upper bound. This results in an infinitely large model space: there are an infinite number of models that can conform to the metamodel. In order to make the model space amenable to search, we require the user to finitise the data in their metamodel, similar to the creation of the terminal set used in Genetic Programming (GP) [27]. For instance, in our zoo example, we may wish to restrict the sizes of cages to be integers between 5 and 50, and not the entire range of integers. We have defined a metamodel (not shown for brevity) that allows practitioners to capture finitisation information in a model. Metamodel attributes can be finitised with a list or a range of possible values, and both meta-classes and references can have a maximum scope defined. Scoping a meta-class restricts the number of its instances that can appear in a model, and scoping a reference enforces an upper bound on the number of objects assigned to that reference. Furthermore, it is also possible to finitise entire data types with a list or range of possible values.

One requirement of the finitisation model, is that the user needs to define a root object type – the container object in which all model elements are situated. In MDE models, it is usual for the model to have a top-level container object (for example, a package in a UML class diagram), and it is not always possible to automatically infer from the metamodel which class this root object should be an instance of.

Finally, a user can choose to ignore certain meta elements (such as specific classes or attributes) if they are not of interest to the task at hand. As with the terminal set in GP, a user may specify different metamodel data finitisations depending on the problem being addressed.

3.1.2 The Genotype

The representation we have defined is linear; more precisely, an individual in our representation is a structured string of integers. As shown in figure 3,
Fig. 3: The structure of an individual in the representation. Segments are the building block for representing models as integers. (a) is the feature selector node, and (b) is the feature value node.

this string is divided into a number of segments, with each segment representing a single object in the model being represented. The first node in a segment, the class node, identifies which meta-class is instantiated by the object being represented. Successive nodes, the feature nodes, define the values of features from that meta-class. Feature nodes are grouped into pairs: the first member of the pair, the feature selector node, identifies a feature of the meta-class; the second member of the pair, the feature value node, specifies the value that should be assigned to that feature in the object being represented.

3.2 Genotype-Phenotype Mapping

The steps taken to transform an individual into a model (the mapping from genotype to phenotype) are presented below, and illustrated with a simple example shown in figure 4.

The first two steps are performed by the user to configure the search:

1) Assign identifiers to features in the metamodel:
In order to reference meta-elements (i.e. elements in the metamodel), we assign identifiers to them (see figure 4a). Each instantiable meta-class is given an identifier, and each of its meta-features is also assigned an identifier, unique to that meta-class. 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. Furthermore, we declare one class to be the root class. All objects created during the mapping will be directly or indirectly contained by a single instance of the root class.

2) Finitise any infinite data: In order to represent the value(s) of a feature, we define a finitisation model for the metamodel (see figure 4b). We also assign identifiers to each of the data values in order to reference them in the mapping.

The final two steps take place for every individual evaluated by the search algorithm. These steps map

Fig. 4: The steps taken to instantiate an object from our integer-based representation. Meta-element identifiers are displayed in grey circles.
an individual into a candidate solution which can then be analysed by the relevant fitness function.

3) **Map each segment to a model object:** A concrete model object is created by mapping a segment using the definition in figure 3, and the metaclass identifiers defined in figures 4a and 4b. Figure 4c shows how we resolve the segment’s class node (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 attribute to 5, and its quantity attribute 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).

4) **Construct references:** When the feature selector node maps to a reference feature, we take note and move to the next feature pair. This is because not all objects will have been instantiated, and so we are unable to assign the references. Once all segments (ignoring references) have been mapped to objects, and those objects instantiated, we can then assign the references. To assign a reference, we select the set of objects whose type is the same as the reference target, and then take the feature value modulo the size of the list to select one of those objects. If the reference’s upper bound has already been reached, we ignore that feature pair.

Insisting that models have a single root container object (Step 1) requires us to make a decision as to which one of those objects should become the root object. The mapping process may create multiple objects of the root’s type and the reference construction phase may distribute objects evenly amongst these root objects – selecting one of the root objects to keep would mean removing the other roots and all objects that they contain, creating a lot of redundancy in the individual. To address this, our transformation always creates a single object as the root, irrespective of the individual. Furthermore, the source metamodel is automatically analysed to determine whether the root object is allowed (directly or indirectly) to contain objects of its own type. If not, the root meta-class is not given an identifier and therefore cannot be instantiated. One drawback of this solution, is that, as the root object doesn’t have a corresponding segment, we cannot assign values to any attributes it may possess.

### 3.3 Search Properties and Genetic Operators

Here we briefly describe the search-specific characteristics of the representation – the genetic operators and method of initialising the search population, including a discussion about the size of individuals. As mentioned, due to being based on a linear genotype, our representation can be used by many linear genotype-based search algorithms. Therefore, each search technique would make use of the appropriate initialisation method or specific genetic operators required.

#### 3.3.1 Initialisation

In the general case, the initial search population of a population-based algorithm (such as a genetic algorithm) may be generated by assigning random values to each gene. Certain applications, however, may require a different method. For example, it may be beneficial to seed the population with variants on an existing model should the goal be to optimise a specific model. Similarly, for a local 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:

$$|I| = \sum_{i=1}^{n} |S_i|$$

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

$$|S_i| = 1 + (2 \times \#fp)$$

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 differ between segments. An alternative would be to use a fixed number of feature pairs, however this would introduce a high level of redundancy in segments which represent meta-classes with few features.

Currently, the number of feature pairs and number of segments per individual are specified by the user as a range. The initialisation process randomly selects a value from these ranges for each segment and each individual created. In the future these values could be inferred by analysing the metamodel. Individuals are not required to be the same length, to allow for a much more diverse population and to more realistically represent the solution space. Future work is planned to examine the effect of individual length on the results of different problems.

#### 3.3.2 Standard Genetic Operators, Adapted

The linear representation permits standard genetic operators, such as a single-point crossover, to be used. However, the structure of chromosomes 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. Therefore, in order to reduce this, we allow crossover to occur only between segments. This allows entire objects to be copied over to the child being created – only the feature pairs that represent associations will be affected. Figure 5 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 be redundant.

The mutation operator also differs slightly from the usual mutation operator used with linear genotypes. Figure 6 demonstrates how mutating different types of genes causes different amounts of variation in the phenotype. Mutating the class node has the most significant effect on the phenotype, whereas mutating either the feature selector node or feature value node has a lesser effect. Therefore, in our representation, each gene type has a different probability of being mutated which is specified by the user. In the future, we plan to investigate whether using different mutation probabilities for the different gene types would improve the effectiveness of the search algorithm.

### 3.3.3 Custom Genetic 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. There is a need for the ability to lengthen and shorten individuals, and so we include two new operators to do so: (i) Segment Creator inserts a randomly created segment into a random position in an individual; (ii) Segment Destroyer deletes a randomly selected segment from an individual;

As with the standard genetic operators, we plan to thoroughly examine the effects of each of the operators on a number of case studies.

### 3.4 Redundancy

During the mapping process, we introduced the notion of ignoring feature pairs (when references could not be satisfied) and discussed how we would be forced to ignore parts of the model if multiple root objects were allowed. Any part of the genotype that does not appear in the phenotype is said to be non-coding or redundant. Too much redundancy creates an inefficient representation, which cannot encode as much information as its potential [29], so we aim to minimise this. There is the belief, however, that redundancy is a good thing, as mutations to the integer string will activate new areas of the solution space [29], [25]. For example, changing the value of a class node may activate a previously redundant feature pair in the production of that model object. This section looks at the two structural areas where redundancy manifests – entire segments and individual feature pairs.
3.4.1 Non-Coding Segments

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.

3.4.2 Non-Coding Feature Pairs

Where a feature pair relates to a reference, it can become non-coding if no referenceable objects have been instantiated, or if the reference has reached its upper bound. With respect to attributes, 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 repair the feature pair to select a different feature. The latter solution might introduce more clashes with other feature pairs in the segment. The choice between the first and second solution should not cause any problems with a metaheuristic search technique, providing the choice is consistent. Any feature pairs that are ignored are said to be non-coding feature pairs. Our current implementation (section 4) uses the second solution.

Summary

This section presented a generic approach to encoding models for use with metaheuristic search techniques. We devised a linear genotype to encode models, meaning that it can be used with many well-known metaheuristic search algorithms, and we extended the standard set of linear-genotype-related genetic operators with model-specific operators. We now present our implementation of the representation for a widely used modelling platform.

4 IMPLEMENTATION FOR ECORE

We have implemented the representation and finitisation metamodel for Ecore – the metamodelling language of the Eclipse Modeling Framework [33]. Ecore automatically assigns each meta-element an identifier, which could be used in the mapping process. However, Ecore even 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 take the set of instantiable classes, 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.

Our implementation includes the reverse transformation – a phenotype-to-genotype mapper – to allow for population seeding as previously discussed. The search framework and mapping transformations are written in the Epsilon Object Language (EOL) [19] – a general purpose model management language. A bespoke implementation was chosen to provide a flexible basis for exploration with the representation, and also because it allows us to treat all key artefacts as models. Figure 7 shows the metamodel for the search population, and illustrates how we have captured the genotype defined in the previous section as a metamodel. Our genotype-phenotype mapping transformations are, therefore, written as model-to-model transformations.

To utilise the representation and search framework, one is simply required to implement a fitness function to evaluate candidate solutions, and specify the metamodel and associated finitisation model for use in the search. Fitness functions are written in either EOL or Java. This allows the user to utilise the most appropriate means to evaluate candidate solutions. For example, users might wish pass the model into an external program for analysis.

We now illustrate the use of the representation in a typical application – performing coverage analysis of

Fig. 7: The metamodel for the search population.
that models contain a number of very specific objects, is a challenging coverage problem as it requires appropriate finitisation models. EuGENia, in particular, is a complicated, real world model transformation. EuGENia takes a metamodel defined in Ecore and automatically creates the models required to generate a graphical model editor in GME, which are usually created by hand. For each of the three metamodels, we have defined appropriate finitisation models. EuGENia, in particular, is a challenging coverage problem as it requires that models contain a number of very specific objects.

To illustrate that our representation enables the use of multiple metaheuristic search algorithms, we compare three: a GA, hill climbing and random search. The GA has population of 10, the hill climbing selects the best neighbour from a random sample of 4 mutations, and random search is implemented as a GA with the mutation probability set to 1.0 (two elites are kept in each generation, as with the GA, to not punish random search). The GA and random search are executed for 10 generations, and hill climbing for 25, so that all three algorithm make an equivalent number of evaluations. As this experiment is just for proof-of-concept to illustrate the feasibility and practicality of the representation, no tuning was made to encourage the search to achieve full coverage. Furthermore, our population size is conservative to ensure that we do not brute force the solutions.

The fitness function is defined as the number of newly uncovered statements minus the number of newly covered statements, and we attempt to minimise this value, taking the fittest individual at the termination of the algorithm. We repeat this process up to 4 more times, attempting to discover models that provide coverage of previously uncovered areas. Variable-length individuals are used, allowing models to contain anywhere between 3 and 100 objects to demonstrate that our representation can handle scale. Table 1 shows the mean results over ten repetitions of the algorithm. The number of metamodel classes given in the table is presented as “X (Y)”, where X is the total number of classes, and Y is the number of concrete classes.

We do not attempt a detailed analysis of the results, but note only that it was straightforward to apply a simple generic search framework, designed for linear genotypes, to a complicated MDE problem (discovering models) through the use of our proposed representation and its associated mapping from genotype to phenotype. Without this representation, a custom search framework operating directly on the models themselves might have been necessary, as would have been the use of domain-specific genetic operators.

The next section discusses future extensions to the representation and highlights interesting research opportunities for combining SBSE and MDE.

5 Research Opportunities

This section further motivates the case for utilising SBSE methods in MDE. Firstly, section 5.1 discusses some future work related to the representation presented in the previous section. Sections 5.2, 5.3 and 5.4 examine different areas in MDE that could benefit from SBSE techniques, and present existing work that attempts to do so.

5.1 Exploring the Representation Further

The representation we presented in the previous section has a number of areas that demand further investigation. A thorough investigation of the effects of the search operators is required. Analysing each operators’ effect, in relation to their probability of application, over a number of distinct MDE problems would provide guidance to users, helping them understand how to best tailor the representation to their problem. We also plan on exploring the relationship between the size of a metamodel and the length of the operators’ effect, in relation to their probability of application, over a number of distinct MDE problems would provide guidance to users, helping them understand how to best tailor the representation to their problem. We also plan on exploring the relationship between the size of a metamodel and the length of individuals used in the search. The aim for many applications would be to use the smallest individual possible to achieve to the search goal, as this will reduce the size of the search space.

Finally, there are some issues with our representation that need to be addressed. For instance, currently we do not enforce lower bounds on references or enforce the assignment of required fields. This could be achieved by injecting segments or feature pairs into individuals, however this may negatively impact the effectiveness of the search method being used. Another concern is that, as the generation of the root object is external to the genotype, we are unable to assign any values to its attributes. This is, however, a task that can currently be performed by the appropriate fitness functions, or manually by the
user. Furthermore, we do not ensure that the encoded models are valid with respect to constraints defined externally to the metamodel. Metamodels can only define structural constraints and use other languages (such as the Object Constraint Language [12]) to define extra constraints, such as class invariants. Users are currently required to perform this validation during the fitness function and assign fitnesses accordingly.

5.2 Discovering and Validating Models
As mentioned in section 2.3, SBSE methods could help to address some of the problems associated with developing and validating models. France and Rumpe [9] propose that the utilisation of models at runtime is an important research area, and search-based optimisation techniques could provide a way to optimise models at runtime, as has been attempted in the field of online evolution of robot controllers [3].

Previously, we have used grammatical evolution to search over the model space to discover interesting textual models [35]. (The representation used in [35] suffered many flaws which we have addressed in the representation presented in this paper, including the ability to specify data using a finitisation model, and to cross-reference other objects in the model.) Goldsby and Cheng [11] also use search to discover models of interest. They use a GA to generate resilient system behaviour models (form in state diagrams) based on a user-defined set of possible behaviours. An individual in their search space is a sequence of instructions mapping to possible pre-defined transitions and guards. To evaluate individuals, they translate the behaviour models into Promela [16] and perform model checking with respect to a set of temporal properties.

Harman [14] cites sensitivity analysis as being an important topic for future work in the SBSE domain. We have successfully used our representation of MDE models to apply sensitivity analysis to the results of a complex, multi-objective decision making tool [36], which itself uses search to discover solutions and has been applied to the Next Release Problem [4].

Recent work by Cadavid et al. [5] demonstrates a user-driven approach to validating the boundaries (relationships and multiplicity constraints) of a metamodel to ensure that a metamodel is neither under-constrained (allows invalid models) over-constrained (disallows valid models) with respect to the domain. They search over a model space specified by a set of user-defined model fragments – ranges of values for certain properties defined in the metamodel – to obtain a set of models which are dissimilar and cover the set of model fragments. Domain experts can then examine the set of models to detect any faults in the metamodel.

5.3 Discovering and Validating MMOs

The proof-of-concept experiments illustrated in Table 1 show that we did not manage to gain much coverage of the EuGENia transformation. This is not a flaw of the representation – it is capable of expressing the required model information – instead it is due to the vastness of the search space and the algorithm used to explore it. An interesting side effect of performing coverage analysis on EuGENia was that it highlighted the need for robustness testing of MMOs – an area in which Harman claims SBSE can help [14]. EuGENia expects that its input models satisfy certain, unspecified, preconditions and any failure to meet these preconditions causes EuGENia to fail gracefully. Applying robustness testing to EuGENia, or any MMO, would allow transformation developers to discover the implicit preconditions of their transformation, and guard against them by creating explicit preconditions, or better failure responses.

There is much work in the field of search-based software testing [23], [24] that could be reused in MDE. There have, however, been few attempts to date that use this wealth of knowledge. Fleurey et al. [8] use an evolutionary algorithm to optimise an existing set of test models, with respect to metamodel coverage.

With respect to the discovery of MMOs, Kessentini et al. [18] present a search based approach that uses example source and target models to learn model transformations. Their approach applies metaheuristic search to “transformation fragments” – mappings between elements in example source and target models – in an attempt to match the best fragment to each construct in a given source model. The set of matched fragments then creates the target model. This approach relies heavily on the quality of the transformation fragments. If the fragments do not give a good representation of source models, the approach will not be able to find good matches, and thus will not be able to create the target model. Furthermore, the transformation fragments are verbose and have to be written manually.

Genetic Programming may prove very useful for evolving model transformations (and other MMOs).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Metamodel</th>
<th>MMO</th>
<th>GA Coverage</th>
<th>HC Coverage</th>
<th>Random Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowchart</td>
<td>6 (5)</td>
<td>8</td>
<td>19</td>
<td>66.3%</td>
<td>89.7%</td>
</tr>
<tr>
<td>Triangle</td>
<td>2 (2)</td>
<td>4</td>
<td>21</td>
<td>96%</td>
<td>96%</td>
</tr>
<tr>
<td>EuGENia</td>
<td>20 (15)</td>
<td>81</td>
<td>631</td>
<td>10.8%</td>
<td>12.1%</td>
</tr>
</tbody>
</table>

TABLE 1: Naïve coverage analysis of three MMOs to illustrate the practicality of our representation.
For example, the transformation language constructs could make up GP’s function set, and the associated metamodel(s) would provide the terminal set (as transformations are generally written at the metamodel level, explicit information about model data is not required). Alternatively, as transformations are commonly a sequence of rule applications, a technique such as N-Gram GP [26] could be applied.

5.4 Other Application Areas and Challenges

Non-functional properties of models and MMOs, such as performance, human understandability, and the level of coupling and cohesion, should also be investigated. Existing work in the SBSE field has looked at analysing coupling and cohesion of class diagrams [28], however these works do not address domain-specific models, in which traditional software metrics may not directly apply. Li et al. [21] show how evolutionary algorithms can be used to suggest alternative software architectures based on non-functional requirements.

An open challenge in the SBSE world, which applies to MDE, is the issue of landscape visualisation [14]. The complex nature of models makes the landscape incredibly difficult to visualise. Furthermore, to allow the union of MDE and SBSE to progress, we will require a number of benchmark problems for which to compare solutions against. Research into evolutionary algorithms has been around for decades and provides many benchmarks for SBSE methods to tackle. Not all of these benchmarks will map naturally to MDE, and so we need to develop a useful repository of benchmark problems for this field.

6 CONCLUSION

MDE provides an effective way to develop large-scale, complex systems thanks to its advocacy for abstraction and decomposition. The complexity of MDE models, however, introduces many challenges for MDE practitioners. The solution to a large number of MDE problems can be represented as models, and so in this paper we have proposed a generic linear transformation approach to generate models automatically. To allow the union of MDE and SBSE to progress, we will require a number of benchmark problems for which to compare solutions against.

Acknowledgements. This research was supported by the EPSRC, through the Large-Scale Complex IT Systems project, EP/F001096/1.

REFERENCES