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

Department of Computer Science,
University of York, UK
{jw,smp,louis,paige,fiona}@cs.york.ac.uk

Abstract. This paper describes a novel approach to the derivation of model-driven engineering (MDE) models using metaheuristic search, and illustrates it using a specific engineering problem: that of deriving computer game characters with desirable properties. The character behaviour is defined using a human-readable domain-specific language (DSL) that is interpreted using MDE techniques. We apply the search to the underlying MDE metamodels, rather than the DSL directly, and as a result our approach is applicable to a wide range of MDE models. An implementation developed using the Eclipse Modeling Framework, the most widely-used toolset for MDE, is evaluated. The results demonstrate not only the derivation of characters with the desired properties, but also the identification of unexpected features of the behavioural description language and the game itself.

1 Introduction

The search-based approach to deriving Model Driven Engineering (MDE) models described in this paper is generic, but was motivated by a specific engineering challenge encountered by the authors: that of deriving suitable game player characters in a computer game called 'Super Awesome Fighter' (SAF).

The SAF game was developed to illustrate MDE concepts to high-school students and is played between two human-specified fighters, or a human-specified fighter and a pre-defined (or 'non-player') opponent. A human player specifies their fighter's behaviour using a bespoke Fighter Description Language (FDL) which covers aspects such as the power of the fighter's kick, the reach of its punch, and whether the fighter punches, kicks, walks towards its opponent, or runs away in particular situations. The description of both fighters is provided to the game engine at the beginning of the game and interpreted using MDE tools. Play then proceeds automatically, and at each stage of the game, the game engine decides on the appropriate action for the fighters according to their FDL

M.B. Cohen and M. Ó Cinnéide (Eds.): SSBSE 2011, LNCS 6956, pp. 112–126, 2011.

descriptions. The winner is the fighter with the best 'health' at the end of the game.

For the game to be interesting, it should be challenging – but not impossible – for a human player to specify a winning fighter. For example, it should not be too easy for a human player to describe a fighter that beats all other opponents, both human and non-player. Similarly, the pre-defined non-player fighter should consistently win against the poorest human-specified fighters, but nonetheless should be beatable by the best fighters. We found that checking these requirements for the game play was difficult and time-consuming when fighters descriptions were investigated manually, and so were motivated to consider an automated search-based approach.

The paper makes two major contributions. Firstly, we demonstrate the use of Search-Based Software Engineering techniques in deriving fighters with particular properties, such as consistently beating all opponents. The search algorithm is a form of Grammatical Evolution (GE), which is a natural choice for this application since the FDL is an example of a context-free grammar to which GE is normally applied. The fighters derived by the search algorithm provide useful information for the software engineer in modifying the game play or the Fighter Description Language to ensure a suitable challenge for human players.

The second contribution is an implementation of the genotype-to-phenotype mapping process that is central to GE using the Eclipse Modeling Framework (EMF), a widely-used MDE toolset. Since the fighter description is implemented as a model using EMF, and the FDL is a concrete syntax for this model, the use of EMF model transformation technologies is a very convenient method of performing the mapping for this particular application. Moreover, we have implemented the mapping process in an automated and generic manner, enabling its use with other optimisation problems where solutions are expressed in terms of MDE models.

Related work that applies metaheuristic search to MDE models includes the use of particle swarm optimisation and simulated annealing in model transformation [7]; the evolutionary generation of behavioural models [4]; and the analysis of non-functional properties of architectures described using MDE [11]. However, such combinations of Search-Based Software Engineering and MDE are relatively rare, and our intention is that our generic implementation of GE using an appropriate toolset will facilitate further research in this area[1].

The paper is structured as follows. Section 2 describes relevant background material: Grammatical Evolution, MDE concepts and terminology, and details of the Fighter Description Language. The generic genotype-to-phenotype mapping process using EMF model transformation is explained in section 3. An empirical evaluation of the approach is described in section 4. Section 5 concludes the paper and outlines future work.

[1] The code for the GE implementation, the SAF game, and the results of the empirical work, are available from: `http://www-users.cs.york.ac.uk/jw/saf`

2 Background

This section explains the main features of Grammatical Evolution, relevant Model-Driven Engineering concepts and terminology, and the Fighter Description Language.

2.1 Grammatical Evolution

The technique of Grammatical Evolution (GE) was first described Ryan and O'Neill [15,14] 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 [15], deriving rules for foreign exchange trading [1], and the interactive composition of music [17].

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 1 illustrates the process using a simple grammar for naming pubs (bars).

The pub naming grammar is defined in BNF at the top of figure 1 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

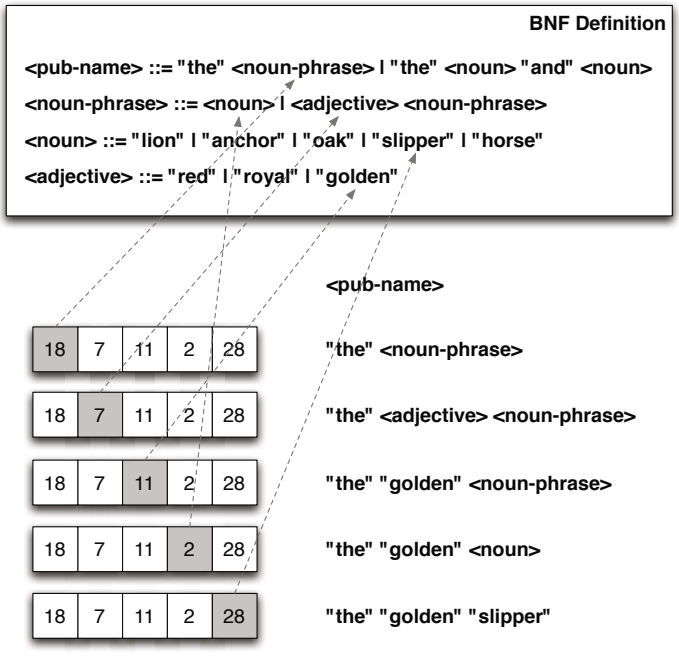

Fig. 1. An example of genotype-to-phenotype mapping in Grammatical Evolution

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 genotype consists of a sequence of integer values, each of which is termed a *codon*. 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 value of the first codon determines which choice to use by taking the value modulo the number of choices. In this case the codon value is 18, there are 2 choices, $18 \mod 2 = 0$, 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 codon. The codon has a value of 7, there are 2 choices, $7 \mod 2 = 1$, 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 codon values than are present in the genotype, codon values are re-used starting at the first codon. 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 [12] and particle swarm optimisation [13], 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.

2.2 Model-Driven Engineering

A *model* can be thought of as an *abstraction* of a problem under scrutiny; the abstraction is typically created for a specific purpose. Models have been used in many engineering disciplines for years, yet in software engineering models have often played a secondary role – as documentation or a means of problem exploration [16]. *Model-driven engineering* (MDE) is a software development practice that treats models as first-class artefacts in the development process. MDE focuses on modelling the system at the level of the application domain, and via a series of automatic transformations, generating code.

Models in MDE are defined and constrained by their *metamodel* – another model that establishes the form a model can take; a metamodel can be thought of as the *abstract syntax* of the model [8]. A model that adheres to the concepts and rules specified in a metamodel is said to *conform* to that metamodel.

An important concept in MDE is model transformation [2]. Examples of common transformations are generating code from a model (a *model-to-text* transformation), generating a model from code (a *text-to-model* transformation), and transforming a model into one that conforms to a different metamodel (a *model-to-model* transformation). Other model management activities include validating models, migrating models to newer versions of their metamodel, merging models, and comparing models.

One of the most widely used modelling frameworks is the Eclipse Modeling Framework (EMF) [18], part of the Eclipse IDE[2]. EMF provides mechanisms for creating, editing and validating models and metamodels, as well as for generating code from models. EMF generates a Java implementation of metamodels where each of the metamodel's classes (called meta-classes) corresponds to a single Java class. This means that these classes can be instantiated to create models conforming to the metamodel. EMF can also create (tree-based or graphical) editors for models conforming to metamodels [18].

2.3 The Fighter Description Language

The fighting game, SAF, introduced in section 1 allows the behaviour of fighter characters to be defined in a bespoke domain-specific language, the *fighter description language* (FDL). Fighters in SAF are MDE models, which are described by the FDL. Figure 2 shows a simplified version of the metamodel for a SAF fighter.

A fighter (`Bot`) in SAF 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 9). These characteristics represent trade-offs: a more powerful strength characteristic limits the speed with which the fighter can move. 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`. `BehaviourRules` specify how the fighter should behave in certain `Conditions`. A rule is composed of 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 they want to either block high or block low, and the game will pick one of these at random.

We have defined FDL using Xtext [3], a tool and language that enables the creation of custom languages based upon metamodels. Xtext generates a text-to-model parser for languages, meaning that the concrete syntax of FDL can automatically be parsed into a model and used with the SAF game. Furthermore, the Xtext grammar definition language also conforms to a metamodel and it is this metamodel that allows us to perform grammatical evolution over FDL (and other languages defined using Xtext). Listing 1 illustrates the syntax of FDL using an example fighter[3].

[2] Eclipse website: `http://www.eclipse.org`

[3] The FDL grammar is available from: `http://www-users.cs.york.ac.uk/jw/saf`.

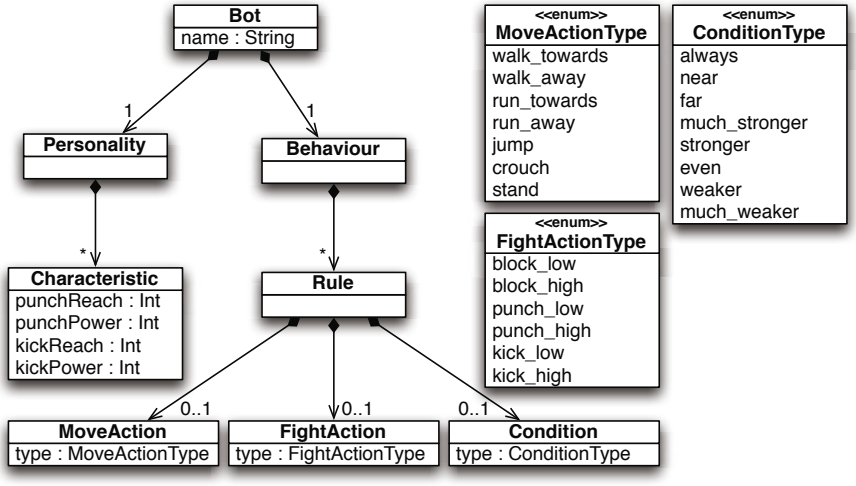

Fig. 2. The simplified metamodel for the character behaviour language used in SAF

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

Listing 1. An example character defined using FDL

Our choice of using Xtext was due to our familiarity with it, and our approach can be implemented for any equivalent grammar definition language that has a metamodel, such as EMFText [5].

The next section shows how the Xtext grammar defined for FDL can be used with metaheuristic search in order to discover the set of behaviours that define a good fighter.

3 Genotype to Phenotype Transformation

In this section we explain the process of mapping the genotype (integer string) to the phenotype (fighter) using model transformations. The inputs to the process are the genotype and the definition of the Fighter Description Language grammar. The latter is initially defined as a text file, but the mapping utilises

a more convenient form of the grammar expressed as a model that conforms to the Xtext metamodel.

3.1 Creating the Genotype Model

The first step in the process is to turn the genotype into a model representation in order to perform the model transformation. Figure 3 illustrates the metamodel of our genotype. A `Genotype` is composed on a number of `Codons`. A `Codon` has one attribute – its `value`, and one reference – a pointer to the `next` codon in the chromosome.

The integer string from the search algorithm is used to create a model that conforms to this metamodel. A `Codon` class is created for each codon, and its `value` attribute is set to the value of the codon. Each `Codon`'s `next` reference is set to the successive codon, with the final codon in the chromosome pointing back to the first. This creates a cycle, meaning that genotype wrapping is handled automatically by traversing the references. An example model that conforms to this metamodel is shown as part of figure 4.

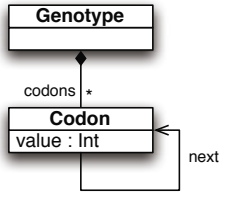

Fig. 3. The metamodel for a Genotype

3.2 Transliterating the Phenotype

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) [10], a general purpose model management language that is part of the Epsilon model management platform [9]. Figure 4 is an overview of this transformation process.

As grammar definitions written in Xtext conform to a metamodel, we can define the transformation at the metamodel level, enabling the mapping process to be applied to any Xtext grammar. 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, codons from the genotype model are used to select which reference 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.

To illustrate this approach, consider the rule in the fragment of the FDL grammar shown in listing 2. This rule specifies the characteristics of a fighter, and contains four choices. When parsed into a model conforming to the Xtext metamodel, it takes the shape of figure 5.

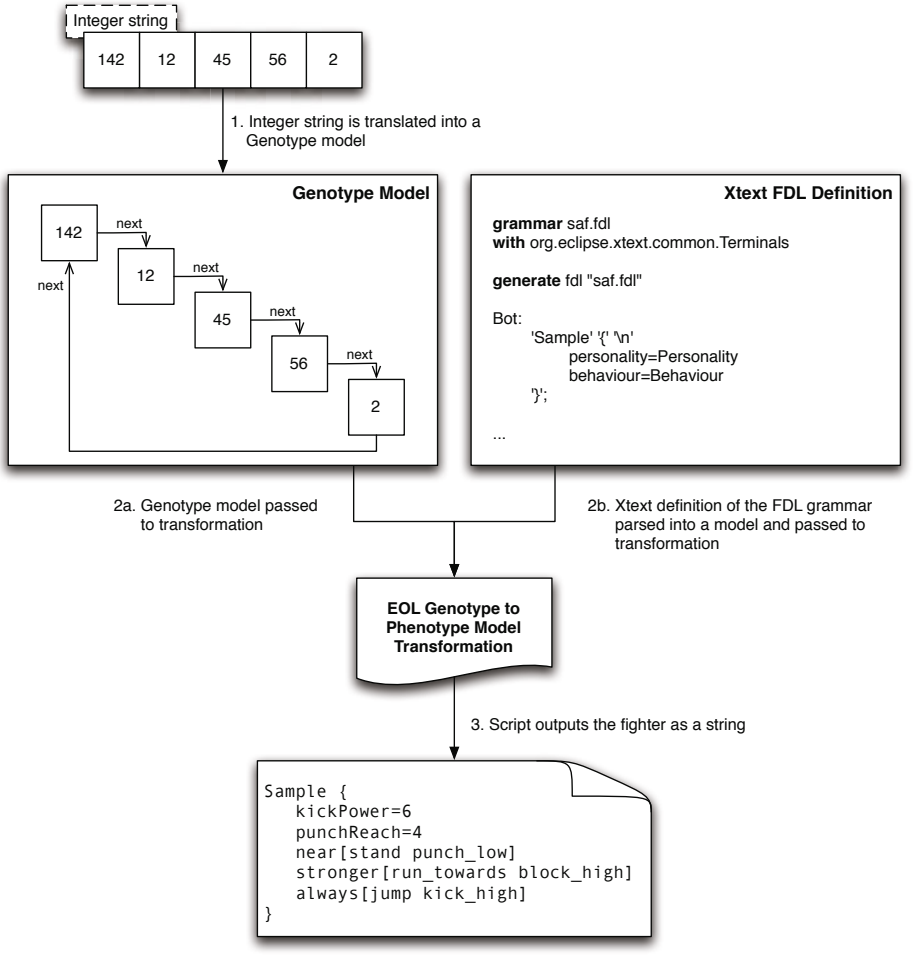

Fig. 4. The process followed in order to transform the genotype into the phenotype

When this rule is reached during the transformation, the current codon's value identifies which alternative to execute by taking the codon's value modulo the number of choices. If the first alternative is chosen, the keywords `punchReach` and = will be added to the output string, and the next codon in the genotype model will select the `NUMBER` to assign to the selected characteristic. The execution can then traverse back up the reference chain and execute the next production rule in sequence, or terminate. If the user-defined number of genotype wrappings is reached during the execution, the transformation aborts. Otherwise, the transformation results in a string that conforms to the grammar of interest – in our case, a fighter.

```
Characteristic:
  'punchReach' '=' value=NUMBER '\n' | 'punchPower' '=' value=
     NUMBER '\n' |
  'kickReach' '=' value=NUMBER '\n'  |'kickPower' '=' value=
     NUMBER '\n' ;
```

Listing 2. The Xtext grammar rule for defining characteristics of a fighter

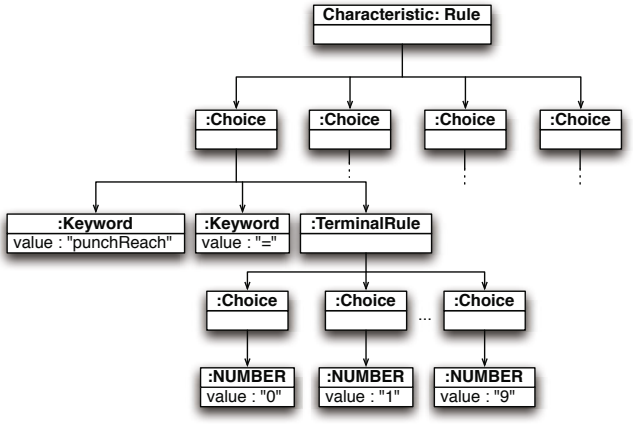

Fig. 5. A section of the FDL grammar model demonstrating how choices are made during the transformation process

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 return to the specific application described in the introduction where the phenotype is the SAF game Fighter Description Language (FDL), a concrete syntax for model of the fighter's capabilities and behaviour.

The objective of this evaluation is to understand the feasibility of the proposed search-based approach in the context of the original motivating problem: that of investigating the requirements for interesting and challenging SAF game play. These requirements are expressed as two experimental questions:

EQ1. Is it possible to specify unbeatable fighters? If so, it may be necessary to amend either the game play or restrict the fighter description language to limit the possibility of a human player specifying such a fighter.

EQ2. 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.

Since this is an initial evaluation of feasibility, no explicit consideration of efficiency (speed and resource usage) of the approach is made. We plan to make a detailed evaluation of efficiency as part of our future work.

4.1 Empirical Method

To answer these two questions, our GE implementation is applied to the problem by pairing the genotype-to-phenotype mapping described above with a suitable metaheuristic search algorithm. The primary 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.

Fitness Metric. An assessment of the properties of "unbeatable" (EQ1) and "wins 80% of its fights" (EQ2) requires a set of opponents to be defined. It is not practical to test how the a candidate fighter performs against *all* possible opponents, and so we created a 'panel' of representative opponents by asking our colleagues to specify what they believed would be winning fighters. (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 then 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, and so each candidate fighter fights each opponent a number of times so that a more accurate fitness can be estimated.

The fitness of a candidate fighter 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. It is calculated as:

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

where n_{opps} is the number of opponents in the panel; n_{fights} the number of fights with each opponent; ρ 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^{th} fight against the o^{th} opponent, and 0 otherwise. The proportion of fights to win, ρ, is set to 1 for experiments on EQ1, indicating an optimal fighter must win all fights against all opponents in the panel, and is set to 0.8 for experiments on EQ2. Fighters with lower fitnesses are therefore better since they are closer to winning the desired proportion of fights.

Algorithm Settings and Implementation. The algorithm settings, including the parameters used in the genotype-to-phenotype mapping and in the fitness calculation, are listed in table 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

Table 1. Parameter settings for the genetic algorithm, genotype-to-phenotype mapping, and fitness metric

parameter	setting
number of codons	20
codon value range	0–32767
population size	20
maximum number of generations	50
initialisation method	random codon values
selection method (for reproduction)	tournament, size 2
reproduction method	single point crossover
mutation method	integer mutation (random value)
mutation probability (per codon)	0.1
number of elite individuals	2
maximum wrappings (during mapping)	10
number of opponents (n_{opps})	7
number of fights (n_{fights})	5

settings, for example the use of integer mutation, was made with reference to existing GE studies, such as [6].

The genetic algorithm was a bespoke implementation in Java since this is the language used in the interface to the genotype-and-phenotype mapping. A bespoke implementation was chosen as this was throught 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[4], 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 entirely new random population at each generation (apart from the elite individuals which are retained unchanged in the next generation).

Response. Four experiments were performed: one for each combination of question (EQ1 or EQ2) and algorithm (genetic algorithm or random search). Within 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 EQ1 or EQ2) fighter. Since the fitness metric is noisy as a result of the stochastic `choose` construct in the FDL, 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 ρ causes the term $\rho\, n_{\text{opps}}\, n_{\text{fights}}$ in the fitness function to be non-integer.

[4] ECJ website: `http://cs.gmu.edu/eclab/projects/ecj/`

Table 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.

experimental question	search algorithm	proportion successful
EQ1 ($\rho = 1.0$)	genetic algorithm	0.67 (0.50 – 0.83)
EQ1 ($\rho = 1.0$)	random search	0 (0 – 0.12)
EQ2 ($\rho = 0.8$)	genetic algorithm	0.97 (0.88 – 1.0)
EQ2 ($\rho = 0.8$)	random search	0.47 (0.31 – 0.66)

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

Listing 3. Example of an 'unbeatable' fighter description found by the genetic algorithm

4.2 Results and Analysis

Table 2 summarises the results of the four experiments[5]. The 'proportion successful' column is the fraction 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 (chosen since it is typically a conservative estimate of the interval).

For EQ1, 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 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, and therefore the game play or the FDL may need to be re-engineered to avoid such fighters.

For EQ2, 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 4.

[5] Detailed results are available from: `http://www-users.cs.york.ac.uk/jw/saf`.

```
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(
        punch_low punch_high)]
5   always[crouch kick_low]
6 }
```

Listing 4. Example of an 'challenging' fighter description found by the genetic algorithm

An unintended, but very useful, outcome of these experiments was that search process exposed some shortcomings in the Fighter Description Language that were not discovered by human players. 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. 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 5. Example of an 'unbeatable' fighter description that illustrates language shortcomings

Further examples of language shortcomings are illustrated in the description shown in listing 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 EQ1 in table 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 EQ2, 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.

5 Conclusions and Future Work

In this paper we have presented a novel application of Grammatical Evolution to the derivation of fighting game characters that possess desirable properties. The genotype-to-phenotype mapping process uses model transformation technologies from the Eclipse Modeling Framework, facilitating the implementation for this specific application, as well as enabling the same approach to be used on other optimisation problems where the solutions are expressed as MDE models. The range of potential applications include not only other domain specific languages that conform to a metamodel, but also more general models.

We intend to continue this work in a number of directions. Firstly, in the context of this specific application, 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.

More generally, we aim to improve the generality of the genotype-to-phenotype mapping process. A first step is to accommodate cross-referencing, whereby multiple parts of the phenotype must refer to the same instantiated element; this feature was not required for the SAF Fighter Definition Language. This enhancement would permit the mapping to be used with any Xtext grammar definition to generate concrete instances. We also intend to investigate the use of *bi-directional* model transformation, enabling a reverse phenotype-to-genotype mapping in addition to current genotype-to-phenotype: this would be useful for search problems in which an additional local search is performed on the phenotype as part of a hybrid search algorithm, or an invalid phenotype is repaired, and such changes need to be brought back in to the genotype.

References

1. Brabazon, A., O'Neill, M.: Evolving technical trading rules for spot foreign-exchange markets using grammatical evolution. Computational Management Science 1, 311–327 (2004)

2. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
3. Efftinge, S., Voelter, M.: oAW xText: A framework for textual DSLs. In: Proc. Workshop on Modelling, Eclipse Con. (2006)
4. Goldsby, H.J., Cheng, B.H.C.: Avida-MDE: a digital evolution approach to generating models of adaptive software behavior. In: Proc. Genetic and Evolutionary Computation Conf., GECCO 2008, pp. 1751–1758 (2008)
5. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
6. Hugosson, J., Hemberg, E., Brabazon, A., O'Neill, M.: Genotype representations in grammatical evolution. Applied Soft. Computing 10, 36–43 (2010)
7. Kessentini, M., Sahraoui, H., Boukadoum, M.: Model Transformation as an Optimization Problem. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008)
8. Kleppe, A.: A language description is more than a metamodel. In: Fourth Int'l Workshop on Software Language Eng. (October 2007)
9. Kolovos, D.S., Rose, L.M., Paige, R.F.: The Epsilon book (2010) (unpublished)
10. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
11. Li, R., Chaudron, M.R.V., Ladan, R.C.: Towards automated software architectures design using model transformations and evolutionary algorithms. In: Proc. Genetic and Evolutionary Computation Conf (GECCO 2010), pp. 1333–1340 (2010)
12. O'Neill, M., Brabazon, A.: Grammatical differential evolution. In: Proc. 2006 Int'l Conf. Artificial Intelligence (ICAI 2006), pp. 231–236 (2006)
13. O'Neill, M., Brabazon, A.: Grammatical swarm: The generation of programs by social programming. Natural Computing 5, 443–462 (2006)
14. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Trans. Evol. Comput. 5(4), 349–358 (2001)
15. Ryan, C., Collins, J.J., Neill, M.O.: Grammatical Evolution: Evolving Programs for an Arbitrary Language. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) EuroGP 1998. LNCS, vol. 1391, pp. 83–96. Springer, Heidelberg (1998)
16. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
17. Shao, J., McDermott, J., O'Neill, M., Brabazon, A.: Jive: A Generative, Interactive, Virtual, Evolutionary Music System. In: Di Chio, C., Brabazon, A., Di Caro, G.A., Ebner, M., Farooq, M., Fink, A., Grahl, J., Greenfield, G., Machado, P., O'Neill, M., Tarantino, E., Urquhart, N. (eds.) EvoApplications 2010. LNCS, vol. 6025, pp. 341–350. Springer, Heidelberg (2010)
18. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. The Eclipse Series, 2nd edn., Addison-Wesley, Reading (2009)