# Searching for Model Migration Strategies

James R. Williams
Department of Computer Science
University of York
York, UK
jw@cs.york.ac.uk

Richard F. Paige
Department of Computer Science
University of York
York, UK
paige@cs.york.ac.uk

Fiona A. C. Polack
Department of Computer Science
University of York
York, UK
fiona@cs.york.ac.uk

## ABSTRACT

Metamodels, like many software artefacts, are subject to evolution. If a metamodel evolves, models that previously conformed to the metamodel may become non-conformant, and must be *migrated* to reestablish conformance. Manually migrating models can be tedious and error prone, and a number of solutions have arisen to aid in the creation of a *migration strategy* - a model transformation which automates the migration. This paper asks whether we can make use of research in the field of *Search-Based Software Engineering* to automatically discover migration strategies, and discusses the challenges involved. In particular, we explore how tools that provide *coupled evolution* of metamodels and models provide a suitable platform for which to apply search techniques.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

## 1. INTRODUCTION

Metamodels define the language in which models are written. Metamodels *evolve* as the requirements of the language change, and as language developers improve their understanding of the ways that people create and use models. For example, the UML metamodel has undergone numerous revisions and evolutions over two decades, including the addition of new concepts, structural changes, as well as conceptual revisions. When a metamodel evolves, the changes made can be *additive*, where concepts are added, *destructive*, where concepts are deleted, or *constructive*, where the structure is revised so that concepts are repositioned elsewhere in the metamodel [7].

Changes to a metamodel can affect the conformance of models and so these models must be *migrated* to conform to the evolved metamodel. This is analogous to other software engineering artefacts whose consistency should be maintained, e.g. design documents with the implementation. Manually migrating each model is tedious and prone to error and any tool which automates the migration needs to correctly migrate an arbitrary set of models [16]. Conformance may simply mean deleting any elements from the model which are no longer valid, but this may produce semantically different models, and so any *migration strategy* needs to consider the semantics of the model it is migrating [8].

*Search-Based Software Engineering* (SBSE) is predicated on the fact that it is often easier to determine whether one solution to a problem is better than another, than it is to develop an optimal solution to that problem [6, 2]. Software engineering problems can be reformulated as an optimisation problem, and *metaheuristic algorithms* search over the space of possible solutions to the problem. SBSE techniques have been successfully applied to many aspects of software engineering in the last decade. However, determining an efficient search-amenable representation of a problem can be a challenge, in particular in determining a metric to state how 'good' a solution is and how it compares to an alternative solution.

This paper argues that model migration can be successfully couched as an optimisation problem and in doing so can reduce the effort demanded in developing migration strategies. We propose that *operator-based co-evolutionary* approaches to model migration offer a promising medium to which SBSE techniques can be applied in order to automatically discover model migration strategies from just two versions of a metamodel. In section 2 we briefly overview background material. Section 3 examines the different ways in which model migration might be couched as a search problem by analysing the different approaches to model migration. Section 4 further details how operator-based co-evolutionary approaches might be reformulated as an optimisation problem, whilst highlighting the challenges involved in such a feat. Finally, section 5 concludes.

## 2. BACKGROUND

This section briefly describes the tools developed to support model migration and outlines SBSE in further detail.

### 2.1 Approaches to Model Migration

Rose et al. [18] define three categories of approaches to developing model migration strategies. Firstly, there is *manual specification*, which requires the metamodel developer

to encode the migration strategy explicitly, either in a general purpose language or a model-to-model transformation language [18]. This approach gives the metamodel developer fine-grained control over the migration strategy, but the work of integrating the strategy with the modelling framework is a challenging specialist activity [18]. The second approach, *operator-based co-evolution*, allows users to evolve their metamodel using pre-defined operators (small transformations applied to the metamodel), each of which has an associated model migration action. Therefore, the migration strategy is created automatically whilst evolving the metamodel. Finally, *metamodel matching* analyses the metamodel and the metamodel *history* (the set of changes made) to infer the migration strategy [18]. This approach is superficially simple, but it can be challenging to produce the correct, semantic-preserving strategy [18].

## 2.2 Search-Based Software Engineering

Three things are required in order to reformulate a software engineering problem as an optimisation problem:

1. Define a *characterisation*, or *representation*, of solutions that is amenable to search techniques. This is important because many software engineering problems make use of artefacts with a complex native format (e.g., graphs) that is not easily and directly amenable to search. It must be possible to easily map this *characterisation* to the native format of interest.

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. Fitness is usually measured against the native format of the candidate solution.

The choice of representation and search operators plays a crucial role in the efficacy of the search algorithm [19].

*Metaheuristic algorithms* are a type of *stochastic optimisation* technique [12]. The goal of these algorithms is to optimise the best known solution through the iterative application of stochastic search operators. Arguably, the most widely known metaheuristic search algorithm is the *genetic algorithm* (GA) [5]. GAs act upon a *population* of candidate solutions that are combined and mutated (mimicking biological evolution) in an attempt to improve the overall fitness of the population, whilst seeking a globally optimal solution. GAs are executed for a number of *generations*, each of which produces a new variant population and each solution's fitness is evaluated. The GA terminates when a pre-determined maximum number of generations have been evaluated, or until a desired fitness is obtained, or until the population converges and no further improvement in fitness is detected.

A similar metaheuristic approach, tailored specifically towards discovering computer programs, is *genetic programming* (GP) [15]. GP also mimics biological evolution, but each of the solutions examined are programs – users define the set of terminal nodes and language operators that can be used in the programs created by GP. Other metaheuristic algorithms include: hill climbing and simulated annealing (see [2] for a brief overview) and ant colony optimisation [3].

# 3. MODEL MIGRATION AS A SEARCH PROBLEM

A migration strategy is, in MDE parlance, an in-place model-to-model transformation. Searching for model transformations is relatively understudied. Kessentini et al [11] present a metaheuristic approach to learning model transformations based on example source and target models. Their approach applies metaheuristic search to "transformation fragments" – specific 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 produces the target model. This approach relies heavily on the quality of the transformation fragments; if the fragments do not adequately cover all parts of the source metamodel , 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, requiring a lot of manual effort in creating the training set.

The particular representation used by SBSE techniques is often problem specific; certain software engineering problems are more suited to a particular representation or metaheuristic technique. We address the three categories of approaches to model migration presented in section 2.1 and discuss how they might be couched as a search problem.

## 3.1 Manual Specification

A number of task-specific languages have been devised to aid model migration (e.g. *Epsilon Flock* [17]). These languages provide an opportunity to use a technique such as GP to generate migration strategies. The information contained in both the evolved and original metamodels could be used for the terminal nodes for the programs created. However, languages like Epsilon Flock are rule-based and it is not immediately obvious how many rules would be required to perform the migration, which may prove difficult in adequately guiding the search algorithm towards a good solution.

Furthermore, how can we provide a useful fitness metric to feedback into the search algorithm? Applying the candidate solution to example unmigrated models and checking their conformance with the evolved metamodel would not be enough to guide the search algorithm. This is because conformance is not a strong enough measure of correctness – as previously mentioned, a migration strategy could simply delete every element in a model in order to make it conform to the evolved metamodel. Therefore it would be required that we measure the fitness against a training set of example pairs of unmigrated and manually migrated models, and devise a fitness score based on the similarities between the manually migrated models and those migrated by the candidate solution. This approach, however, as with Kessentini et al's approach [11] requires a lot of manual effort to create a training set that provides a useful heuristic, which defeats

the goal of automating migration.

## 3.2 Metamodel Matching

Metamodel matching approaches examine two versions of a metamodel to create a *difference model* or *matching model* – a model which captures the two metamodels' equivalences and differences – from which the migration strategy is inferred. One issue with this approach is that metamodel matching is often achieved through the use of a combination of matching heuristics (called a *matching strategy*), the best combination of which is problem specific [4]. It would be possible to use SBSE techniques to search over the combinations of heuristics to determine which matching strategy is most suited to a particular problem, however in order to evaluate a matching strategy one needs to compare the match results against the actual answer, requiring much effort in manually determining the changes between the two versions of the metamodel. On unfamiliar problems, without a target matching goal, it would be very challenging to evaluate a matching strategy and therefore difficult to have confidence in the migration strategy produced.

## 3.3 Operator-Based Co-Evolution

Operator-based co-evolution differs from the other two approaches in that the migration strategy is automatically produced from the sequence of operators which were applied to evolve the metamodel. However, if a metamodel was not evolved using one of the operator-based tools such as *Edapt* [7] (previously known as *COPE*), it can be difficult to reverse engineer the evolution [16]. The sequence of operators, nevertheless, provides a structured medium to search over and so if we are able to discover the correct sequence of operators for the metamodel evolution, we obtain the migration strategy for free.

Working at the metamodel level also allows us to utilise a fitness function that does not require a large training set of migrated models. The fitness function of a candidate solution could be calculated by applying the candidate sequence of operators to the original version of the metamodel and comparing the result with the evolved version of the metamodel. For example, the number of matching elements could act as a fitness score. Alternatively, as metamodel structure is not the top priority, we could make use of a *semantic differencing* technique (e.g. [14, 13]) to compare the evolved metamodel with that produced from the candidate solution. Providing the candidate solution results in a metamodel that is semantically equivalent to the evolved metamodel, the migration strategy produced should be suitable. Care is still needed to ensure that the migration strategy is preserving the semantics of the models, but that could be addressed, for example, by minimising the number of destructive operators in the sequence.

One caveat of operator-based approaches is that they heavily rely on the quality of the library of operators. Herrmannsdoefer et al [10] present an extensive catalogue of coupled evolution operators which have been implemented in their Edapt tool. A large library of operators, however, can increase the complexity of applying the evolution [18] as the different combinations of operators can produce the same result in the metamodel, yet produce very different effects on the model migration strategy. A search-based approach would potentially be able to evaluate these alternative paths and, with a well-defined fitness function, help to produce the optimal migration strategy.

## 3.4 Other Considerations

Searching for a migration strategy would also allow metamodel developers to discover strategies that are (near) optimal with respect to non-functional properties. For example, if a migration is particularly large, the execution time of applying the migration to a model may be important, or memory consumption may be a concern. Or perhaps human comprehension is desirable. Furthermore, it may not be the case that the shortest solution (in whichever approach taken) is the most efficient and so a multi-objective fitness function may be required.

## 3.5 Summary

This section has examined how the different approaches to model migration could be reformulated as an optimisation problem, and highlighted some of the challenges involved in defining a fitness function to quantify the quality of a candidate solution. Our current view is that operator-based co-evolution (using a tool such as Edapt) offers the most promising approach to applying SBSE techniques. The next section examines how this could be achieved in more detail.

# 4. SEARCHING OVER COUPLED OPERATORS

Operator-based co-evolutionary approaches to model migration offer a unique benefit over manual specification and metamodel matching in that migration strategies are automatically produced from the sequence of operators (as opposed to being manually specified or selected using heuristics). This means that a training set of migrated models is not required to analyse candidate solutions. Furthermore, if we are able to automatically discover the sequence of operators that evolve a metamodel from one version to the next, we also obtain the model migration strategy. As with metamodel matching approaches, this would still suffer the issue of preserving the semantics of models but this could be overcome through the use of a clever fitness function.

In this section we look in depth at Edapt[1] [7, 8, 10] – an Eclipse-based tool for operator-based co-evolution which has been successfully applied to many real world problems – and discuss how it could be used in tandem with SBSE techniques to automatically discover model migrations in cases where we have only a single pair of source and target metamodels.

## 4.1 Edapt

Edapt is an Eclipse-based tool that supports metamodel developers with evolving their EMF-based metamodels. It provides an extensive catalogue of co-evolutionary operators [10] with which developers can work and provides a number of useful user interface tools to support the evolution. Edapt stores the changes made to a metamodel within another model called the *history* model. In doing this, Edapt allows any changes to easily be undone. Figure 1 shows a simplified version of the Edapt history metamodel. Releases

[1] `http://www.eclipse.org/edapt`


**Figure 1: A simplified version of the Edapt history metamodel**

allow developers to release a version of their metamodel with the associated model migration strategy, and contain a list of Changes made to the history model by an operator in that release.

There are times when the evolution operators provided by Edapt cannot capture the necessary model migration that would be required. In these instances, Edapt allows users to specify custom migrations in Java. In this paper we do not consider custom migrations.

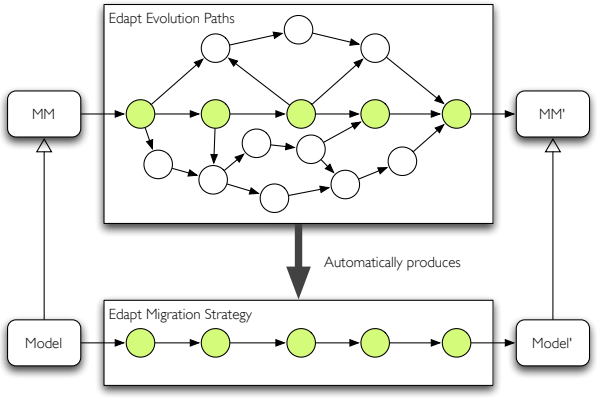


**Figure 2: Illustration of Edapt, highlighting the fact that there may be many evolutions paths. Circles represent the application of Edapt operators.**

Figure 2 illustrates how Edapt allows developers to apply operators to their metamodels in order to evolve a new version, and automatically obtain the model migration strategy. Moreover, there may be many sequences of operators (*evolution paths*) that evolve the metamodel from the previous version to the new, however some of these may contain superfluous operators or paths which do not preserve model semantics. For instance, if the path deletes a metaclass and then recreates an exact copy of it, the associated migration strategy will delete all model objects conforming to the deleted metaclass and cannot reintroduce them. Depending on the requirements of the user, an evolution path which contains superfluous operators may still be acceptable as the important factor is ensuring models migrate correctly.

## 4.2 Searching for the Edapt History Model

To discover the correct sequence of operators that apply to a metamodel we need to search over the space of possible history models for the given problem. I.e. the candidate solutions in this search problem take the form of models. Creating arbitrary models is challenging [1]. In previous work [20] we have produced a method which makes it possible to search over the space of models that conform to a given metamodel. Using this approach would enable us to

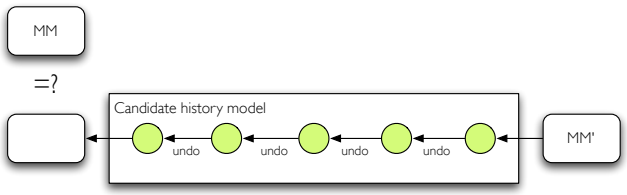


**Figure 3: Rolling back the changes in a candidate history model and comparing the resulting metamodel against the original version.**

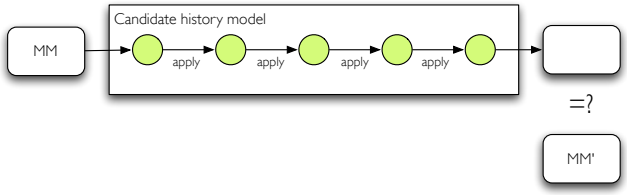


**Figure 4: Applying the changes in a candidate history model sequentially to the original version of the metamodel and comparing the resulting metamodel against the evolved version.**

search for an (near) optimal history model. Furthermore, the information required by our method can automatically be extracted directly from the two versions of the metamodel, and so would require no extra work from the user.

The task of discovering an optimal history model presents an interesting optimisation challenge. Firstly, the sequence of operators in each candidate solution (history model) needs to be applied to the original metamodel in order to calculate a fitness score (calculating the score is discussed in the next section). Edapt's history model only references elements contained in the latest version of the evolved metamodel, but stores information about any deleted elements in order to allow them to be recreated if an operation is undone. This leaves us with two options for how to apply the history model's operators to determine the quality of the solution:

1. Rollback the changes in the history model against the evolved metamodel in an attempt to obtain the original version. I.e. instead of *e*volution, the task becomes *de*volution. (see figure 3)
2. Apply the operations in sequence to the original metamodel in an attempt to obtain a metamodel resembling the evolved metamodel. (see figure 4)

The latter approach will introduce situations where attempting to apply an operation would cause Edapt to throw an exception (for example if the search algorithm produces an operation which references an element that was yet to be created in the un-evolved metamodel). The greatest difficulty in any search-based technique is defining a fitness function that can provide a quantitative score for each solution that is able to guide the search in a meaningful way (see section 4.3). It may be the case the the search algorithm has discovered the optimal solution but with an extraneous operation

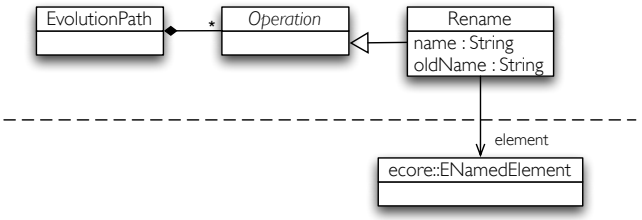


**Figure 5: A simplified version of the *SimpleEdapt* metamodel. Each of Edapt's operators is captured by the metamodel as an Operation. Here we present only the Rename operation, which stores a reference to the element in the evolving metamodel which it affected, as well as the new and old names of the element.**

in the middle that causes an exception to be thrown. Giving a particularly bad score to that particular solution would be unproductive and hinder the search, so perhaps it would be worthwhile to simply ignore the exception-throwing operation and move onto the next whilst calculating the fitness.

A second challenge regarding discovering history models is that an Operation object only captures parts of the change made to the metamodel. In order to allow for operations to be undone, Change objects consist of both the Operation and a set of primitive changes made by that operation, including information which has been deleted or the original values of elements which has been modified. Therefore, in order to discover the optimal history, these primitive changes also need to be discovered by the search algorithm. This kind of model is hugely complicated to efficiently discover through the use of search, due to its multiple interdependencies between objects. To overcome this we propose using an abstraction of Edapt's history metamodel, *SimpleEdapt* (figure 5), to concisely capture the information required by each of the Edapt operations, and a model-to-model transformation to produce the more complex Edapt history model, against which the fitness is evaluated.

As operators are executed in a sequence, some operators will affect the changes made by earlier operators. For example, one operator in the sequence may introduce a new class, which a later operator deletes. If this occurs, the creation operator object in the history model will now be referencing an object in the evolving metamodel which no longer exists. To keep the history model consistent, Edapt updates the creation operator object with a reference to a new object in the history model which stores information about the class which has been deleted. This information is then used to recreate the class in the metamodel if the delete operator is revoked using Edapt's undo feature. This information is captured by the *SimpleEdapt* metamodel and so can be found using search.

A third challenge is the notion of intermediate state. It is common for elements to be created during metamodel evolution which do not appear in the final evolved version of the metamodel (for example, two references may be created and then merged). Therefore, the candidate history models need to also be able to create these intermediate states and so extra information which does not appear in either the original or evolved metamodel needs to be injected into the process. This can be addressed by allowing a small amount of random data to be accessible by the search algorithm used to create the history models.

## 4.3 Fitness Function

The goal of a fitness function is to provide an effective way to indicate how 'close' a candidate solution is to the optimal solution. Once the candidate history model has been applied (either sequentially or rolled back) to obtain a new version of the metamodel, we need to compare this new metamodel against the expected metamodel (either the evolved metamodel or the original) and provide a score that is indicative of the candidate history model's quality.

One option would be to calculate the number of structural differences between the target metamodel and the derived metamodel. Determining an effective scoring mechanism, however, is not trivial. Should one count the number of meta-elements that exactly match? Should fuzzy or partial matching be used? Should certain matched meta-elements, such as classifiers, be weighted more than others? The heuristics used by metamodel matching approaches to model migration may prove fruitful here.

An alternative to structural differencing might be to use semantic differencing [13, 14] instead. Our goal is not to find a structural transformation between two models. Instead, it is to find a transformation that results in a semantically equivalent model which has the same migration path as the desired target model. Therefore, it may be irrelevant if the target metamodel labels a particular meta-class as abstract where the derived metamodel does not; semantically they might be equivalent. However, it is unclear at this stage whether semantic differencing techniques would be able to provide a suitable metric with which to compare other candidate solutions against.

As mentioned in section 3.4, finding a correct migration path may not be the only goal. Other, non-functional, properties may be of interest; therefore, perhaps a multi-objective fitness function is desirable. For example, it is probable that a shorter migration strategy is more likely to be more efficient than a longer one.

Finally, it may be that a combination of measurements is required to accurately score candidate solutions – hybrid semantic and structural comparisons, for example.

## 4.4 Future Plans

We plan to continue investigating the idea of searching over Edapt operators for migration strategies. We have implemented the SimpleEdapt metamodel and have written transformations from SimpleEdapt to the Edapt history model for a small set of Edapt operators. Currently, we are testing this method on the exemplar Petri net evolution found in much of the model migration literature (e.g. see [17, 4, 16]). If this proves successful, we plan to move on to larger case studies such as Netbeans Java (in [4]), and GMF (in [16, 9]), and provide a detailed comparison of our approach against existing approaches.

For each of the case studies, we shall analyse the two approaches to applying the operators in the candidate history model (section 4.2) and compare and contrast numerous measures of fitness (section 4.3).

# 5. CONCLUSION

This paper has argued the case for utilising search-based software engineering practices to help with the automatic discovery of model migration strategies. We have examined the three categories of approaches for model migration and argued that operator-based co-evolutionary approaches offer the most promising platform from which to apply SBSE techniques. The problem of reformulating model migration as an optimisation problem is both challenging and interesting. A particularly challenging problem is defining an effective fitness function for evaluating candidate solutions. We are currently working on exploring and evaluating the ideas proposed in this paper and feel that the challenges presented here are not impossible, and if overcome would provide a useful new approach in the metamodel developer's toolkit.

# 6. ACKNOWLEDGEMENTS

The authors would like to thank Markus Herrmannsdoerfer and Louis Rose for their advice and contributions to the early ideas that have been turned into this paper.

# 7. REFERENCES

[1] B. Baudry, S. Ghosh, et al. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2009.

[2] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B.Mitchell, S.Mancoridis, K. Rees, M. Roper, et al. Reformulating software engineering as a search problem. In *IEEE Software*, volume 150, pages 161–175, 2003.

[3] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[4] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1989.

[6] M. Harman and B. F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[7] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *23rd European Conference on Object-Oriented Programming*. Springer, 2009.

[8] M. Herrmannsdoerfer and M. Koegel. Towards semantics-preserving model migration. In *International Workshop on Models and Evolution*, 2012.

[9] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: the history of gmf. In *Proceedings of the Second international conference on Software Language Engineering*, SLE'09, pages 3–22, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 163–182, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. B. Omar. Search-based model transformation by example. *Software and Systems Modeling*, September 2010.

[12] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[13] S. Maoz, J. Ringert, and B. Rumpe. CDDiff: Semantic differencing for class diagrams. In M. Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 230–254. Springer Berlin / Heidelberg, 2011.

[14] S. Maoz, J. O. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In *Proceedings of the 2010 international conference on Models in software engineering*, MODELS'10, pages 194–203, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[16] L. M. Rose, M. Herrmannsdoerfer, J. R. Williams, D. S. Kolovos, K. Garcés, R. F. Paige, and F. A. C. Polack. A comparison of model migration tools. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 61–75, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model migration with Epsilon Flock. In *Proceedings of the International Conference on Model Transformation*, Malaga, Spain, June 2010. Springer.

[18] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack. An analysis of approaches to model migration. In *Proceedings of the Joint Workshop on Model-Driven Software Evolution and Model Co-Evolution and Consistency Management*, Denver, U.S.A, October 2009.

[19] F. Rothlauf. *Represenatations for Evolutionary and Genetic Algorithms*. Springer Berlin Heidelberg New York, second edition, 2006.

[20] J. R. Williams, R. F. Paige, D. S. Kolovos, and F. A. C. Polack. Search-based model driven engineering. Technical Report YCS-2012-475, Department of Computer Science, University of York, 2012.