Abstract. We explore the application of genetic algorithms in model transformations that can be understood as pattern-based refinements. In MDA (Model Driven Architecture), such transformations can be exploited for deriving a PIM model from a CIM model. The approach uses design patterns as the basis of mutations and exploits various quality metrics for deriving a fitness function. A genetic representation of models and their transformations is proposed, and the genetic transformation process is studied experimentally. The results suggest that genetic algorithms provide a feasible vehicle for model transformations, leading to convergent and reasonably fast transformation process. However, more work is needed to improve the quality of the individual models produced by the technique.

Key words: Model transformation, genetic algorithm, model driven architecture, software design

ACM CCS Categories and Subject Descriptors: D.2.11 Software architectures; D.2.2 Design tools and techniques.

1. Introduction

MDA (Model Driven Architecture) [13] reflects a long-standing trend in software engineering to raise the abstraction level of software system description, together with automated support for transforming high-level descriptions into lower-level ones and eventually into executable code. MDA shares many motivations for developing high-level programming languages, like the ability to use the same high-level descriptions to produce executable code for multiple platforms. In MDA, this is realized by dividing the high-level descriptions (i.e., models) into different layers, CIM (Computation-Independent Model), PIM (Platform-Independent Model) and PSM (Platform-Specific Model). Basically, the CIM level corresponds to a requirements model, while the PSM level corresponds to a detailed design model. The PIM level contains an intermediate model introducing basic platform-neutral solution structures.

Received November 7, 2008.
However, in spite of the high expectations and years of intensive research, MDA is still far from its original vision. If the concepts of the application domain are well-understood, it is possible to map these concepts into solution mechanisms in a systematic way, and build automated transformation support based on this mapping (e.g., [8][10][20]). However, if the domain is not precisely fixed, the transformations require more creative design knowledge which is hard to incorporate in the transformation. This is particularly problematic at the higher abstraction levels, like in transformations from the CIM level to the PIM level.

The interpretations of the role and purpose of CIM vary significantly. It is in some cases considered as a requirements model [9][13], capturing the requirements the user has on the system. MDA Guide [13] points out that CIM is sometimes called "a domain model" and "a vocabulary" that is familiar to the practitioners of the domain in question.

Here we adopt the view that a CIM model is a domain model of the system to be built, represented in a particular way. As customary in object-oriented software architecture design, we assume that the architectural model of the system can be derived by refining a functional model (in this case an abstract domain model) with quality-driven solutions (here patterns).

In this paper, we explore techniques to produce a solution-level (architectural) model (PIM) on the basis of a requirement-level (domain) model (CIM) automatically using genetic algorithms. In genetic algorithms [11], a "good" solution is found through simulated evolution, where individual solution generations are subject to mutation and crossover operations, competing with each other according to their "goodness", as defined by the so-called fitness function. In our case, the mutations are interpreted as specific design solutions (called patterns in this paper) applied in the transformation from the CIM level to the PIM level. The quality requirements are assumed to be encoded in the fitness function. The CIM is first used to randomly generate an initial population of solution models, this population is then submitted to evolution through hundreds of generations, and the result of the transformation is obtained as the best solution model of the last generation.

Genetic algorithms have a number of important benefits in this context. First, the level of domain-independency can be freely decided by adopting different sets of mutation patterns. If the patterns consist of, say, design patterns [6], the technique is completely domain-independent. If the patterns are intended for a particular domain, the technique is tuned for that domain, accordingly. Similarly, the level of expertise is essentially determined by the selection of patterns. In this paper we apply a small set of general-purpose design patterns as mutations. Second, the quality requirements, which are often ignored in model transformations, are naturally taken into account in this approach, since they are encoded in the fitness function. In principle this approach allows any quality requirements that can be evaluated using some computation on the models. Some quality requirements may require additional information to be associated with the models for sensible evalua-
tion, though. Here we will concentrate on two general quality requirements, efficiency and modifiability.

Third, the genetic approach is able to produce solutions that no human designer could come up with, being free of human prejudices. Since the solution proposed by the genetic process is guaranteed to have high quality in terms of the fitness function, it is realistic to expect that in some cases the genetic process could actually outperform a human designer.

The contributions of this paper consist of a technique to apply genetic approaches in model transformations from requirement level to solution level, and of the experimental evaluation of the technique. In particular, we are interested to study the genetic representation of models, the behavior of the genetic process, and the quality of the resulting solution models.

2. Related Work

2.1 Model Transformations

A wide range of model transformation approaches have been presented in the literature. Template-based approaches, such as XSLT [21], graph grammar based approaches like GReAT [1], VIATRA [20] and Triple Graph Grammars [17], are typically used for model-to-text but also for model-to-model transformations. A template usually binds certain aspects of the source and target metamodels and is instantiated in the actual transformation, as many times as the template matches can be found. Relational approaches, like MTF [5] and QVT [14], focus on specifying relations between model structures. VIATRA, GReAT and MTF are examples of declarative languages that specify what should be transformed, instead of how.

The approaches and tools proposed are often applicable only in relatively narrow domains. The underlying reason for this is that automatic transformation solutions with hard-wired metamodel mappings are usually aimed for. Such solutions do not take user decisions or other learning aspects into account. Instead, they always provide the same solution when applied for the same source model. In our approach, the level of domain-independence depends on the mutation patterns chosen. In this paper a sample application of our approach for a domain-independent transformation is demonstrated. In addition, our approach does not assume any hard-wired transformations.

A transformational pattern system that allows gradual and interactive model transformations is presented by the World Wide Web Consortium [21]. This approach allows variation management and human made decisions during the model transformations. The transformation specification consists of transformational patterns and assembly rules. The former describe how transformation rules are implemented, and the latter describe how the individual patterns relate and which patterns are applied to which source model elements. Flexibility and reusability is achieved through the assembly rules. This approach, as ours, aims at more flexible transformations and increasing the level of transformation reuse. However, while it achieves flexibility
by allowing user decisions during the semi-automatic transformations, our approach achieves flexibility with application of genetic algorithms.

While different PIM-to-PSM and PSM-to-code transformation approaches have been proposed, much less work has been reported in the literature on CIM-to-PIM transformations. This may be partly due to the different interpretations of the role of CIMs. Also, practical MDD-fashion transformations seem to work the better the lower the abstraction level is. The reason for this is obvious: the source models themselves are already fine-grained and well-defined, allowing the use of rather straightforward transformations. The transformations from CIMs to PIMs, on the other hand, are less intuitive and require more problem solving; the CIM essentially represents the problem domain and PIM the solution domain.

Zhang et al. [22] point out that since CIM-to-PIM transformations have been less practiced, converting CIM to PIM depends much on designers’ personal experience or creativity. In our approach, creativeness is achieved through application of genetic algorithms. Zhang et al. propose a feature-oriented approach for CIM-to-PIM transformations. In that approach features are considered as key elements of CIM and components as key elements of PIM. Zhang et al. [22] further propose an approach to create components by clustering responsibilities that are operationalized from features.

Rodriguez et al. [16] consider CIM-to-PIM transformations in the domain of business process modeling. They interpret business process models to be CIMs. The transformation rules are defined using QVT, and their purpose is to generate UML classes and use cases that will be part of the PIM of an information system. Our approach instead is not tied to any specific domain.

2.2 Genetic Algorithms

A genetic algorithm maintains a population of possible solutions. In our problem a population is a set of possible architectures which undergoes an evolutionary process imitating the natural biological evolution. In each generation better individuals have greater possibilities to survive and reproduce, while worse individuals have greater possibilities to die and to be replaced. It is believed that this process leads to a combination of the properties of the better individuals, which constitutes a good solution to the problem in question.

To operate with a genetic algorithm, one needs an encoding of possible solutions, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation. We do not explain these concepts in detail here, because we assume that the reader is familiar with the basics of genetic algorithms, as given, e.g., by Michalewicz [11].

Genetic algorithms are widely used in problems related to software engineering [4], but most of the genetic algorithms presented in the literature solve more restricted problems than that of ours. Typical problems consid-
erated are the software clustering problem [7] and systems refactoring [12][18]. However, system clustering considers software architecture only from the decomposition perspective, and software refactoring aims at structural fine-tuning of software architecture, whereas our approach strives for automating the entire architecture design process.

Architectural transformations apply bigger modifications to the system than simple refactoring operations. An example of architectural transformation is the introduction of design patterns in the architecture. Amoui et al. [2] have earlier applied genetic algorithms for finding the optimal sequence of design pattern transformations to increase the reusability of a software system.

The present work is based on our previous system [15]. Similar to Amoui et al. [2] we use high-level structural units as the basis of mutations in a genetic process. We have also applied the supergene idea of Amoui et al. [2], to be discussed in Section 3, as a starting point for representing the architecture. However, there are several differences. First, in principle, we do not consider any specific quality criterion, but we are interested in the overall quality of the architecture. Second, we aim at the synthesis of the architecture starting from requirement-level information, rather than at improving an existing architecture. Third, we do not restrict to design patterns, but also consider more high-level architectural solutions, i.e., architectural styles [19].

We do not need any formalism (like graph grammars) to model the transformations since it is sufficient to use the supergene notations to specify the relations between the entities of the architecture.

After adopting the supergene concept, we proceed with the genetic algorithm mainly in the standard manner. However, we use dynamic mutation probabilities when introducing certain architectural solutions to the individuals of the search space. The rationale behind this policy is that favoring fundamental solutions in the earlier stages of evolution leads to a stronger core architecture that can be more easily refined at later stages by lower-level solutions.

3. Genetic Model Transformations

In this section we describe our approach to genetically handle the transformations that are needed to achieve a PIM from a CIM in MDA. We begin with a set of responsibilities (requirements) that can be given some relative values regarding modifiability and efficiency. Using the information given on the dependencies between the responsibilities, this set is then formed into a responsibility dependency graph. Furthermore, a domain-model for the system is given. The graph is encoded as a chromosome, which is then subjected to the genetic algorithm (implemented with Java). The algorithm transforms the CIM to a PIM through the implementation of a set of architectural patterns to the given model, and produces a UML class diagram as the result. Fig. 1 depicts the process from CIM to PIM in our
Our CIM model consists of two parts, a responsibility graph and a domain model (see Fig. 1). A responsibility graph gives the functional requirements of the system in terms of responsibilities. A responsibility is either a task to be carried out by the system (or some part of it), or a data item that has to be managed by the system (or some part of it). Each node in the graph represents a responsibility, and each directed edge represents a dependency between the two responsibilities. Here a dependency means that the source responsibility relies on the target responsibility. In order to evaluate the system after the model has been subjected to transformation, some attributes of the responsibilities are also given, such as variability (the probability of future changes), parameter size and time consumption. The values for these attributes may be sometimes hard to determine at the requirements analysis stage, but the more accurately these can be estimated, the better will be the result. The given values for the attributes are relative, rather than absolute.

In our work, we have used an e-home as an example system. It contains 52 responsibilities and 90 dependencies between them. A part of the responsibility dependency graph, depicting the drape control component of the example system, is given in Fig. 2, where some sample properties (variability, parameter size, time consumption) are marked in the nodes. The drapeState node is marked with a thicker circle, as it is a data manager responsibility. The calculateOptimalDrape responsibility is a good example of how and why the certain attributes are evaluated: its variability is 3, as the optimal drape position can be calculated differently in different houses and according to different preferences. As it is a heavy operation, also its parameter size and time consumption are among the highest values of those

3.1 Requirements Model
shown here. Responsibilities with such high attribute values play an important role when constructing quality architecture as their placement has a bigger impact on the quality value.

![Diagram](image)

**Fig. 2:** A fragment of the responsibility dependency graph

The domain model part of CIM is constructed on the basis of the responsibility graph by assigning the responsibilities to classes. The division to classes is based on the data responsibilities: a responsibility is placed in the same class as the data it handles (either directly or through another responsibility, in the case where it does not use any data directly). The process actually follows the traditional object-oriented method of extracting classes and their dependencies on the basis of a functional description of the system.

![Diagram](image)

**Fig. 3:** Domain model of example system

This initial model thus gives the system a basic structure by separating subsystems into components. Associations for the model are derived directly from the dependency graph. However, this structure may change, as the class division may be altered through application of architectural patterns. The initial model is presented as a component diagram in Fig. 3. For simplicity, we have only used classes and associations. As can be seen, the initial model is very simplistic – the genetic algorithm is given a logical
starting point with a preliminary decomposition. No decisions are made regarding the architectural styles or the finer details of the architecture.

3.2 Genetic Model Representation

In a traditional chromosome representation, each chromosome consists of several genes, each of which has one field. A supergene, however, has several fields to store data in. Taking this idea as a starting point, it is quite straightforward to place all information regarding one responsibility into one supergene, i.e., each responsibility is represented as a supergene in the chromosome. This also makes it easier to keep the model consistent, as no responsibility can be left out of the model at any point, and there is no risk of breaking the dependencies between responsibilities.

There are two kinds of data regarding each responsibility $r_i$. Firstly, there is the basic information given as input. This contains the responsibilities $R_i = \{r_{i1}, r_{i2}, \ldots, r_{ik}\}$ depending on $r_i$, its name $n_i$, type $d_i$, frequency $f_i$, parameter size $p_i$, execution time $t_i$, call cost $c_i$, call variability $cv_i$ and variability $v_i$. Secondly, there is the information regarding the responsibility $r_i$’s place in the architecture: the class(es) $C_i = \{C_{i1}, C_{i2}, \ldots, C_{iv}\}$ it belongs to, the interface $I_i$ it implements, the dispatcher $D_i$ it uses, the responsibilities $RD_i \subset R_i$ that call it through the dispatcher, the design patterns $P_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ it is a part of and the predetermined model class $MC_i$. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons. Fig. 4 depicts the structure of a supergene $SG_i$. The actual chromosome is formed by simply collecting all supergenes.

![Fig. 4: Supergene](image)

Although the basic architecture is easy to keep coherent (i.e., the structure given by the model is not disturbed as the dependencies between responsibilities stay intact), the design patterns produce challenges at the chromosome level, as careless operations can easily break existing patterns and make the architecture incoherent. Thus, in order to quickly check the legality of an operation, a Pattern field is located in every supergene. This field contains information about the patterns with which the responsibility is involved, and thus if a new pattern is introduced, it is fast to check whether it contradicts the existing patterns. The initial population is made by first creating the desired number of individuals with the basic structure given in the CIM. A random pattern is then inserted into each individual, as a population should not consist of clones. In addition, a special case is left in the population where no pattern is initially inserted.
3.3 Mutation and Crossover Operations

As discussed, the model transformations are made by implementing patterns in the model. As we only deal with the refinements needed to transform a CIM to a PIM, we do not need to take into consideration the actual implementation of the patterns. The implementation is usually platform-specific, and thus belongs to the PSM. The patterns we have chosen include very high-level architectural styles [19] (dispatcher and client-server), medium-level design patterns [6] (façade and mediator), and low-level design patterns [6] (strategy and proxy). This ensures a variety of "building material" for the genetic algorithm, and the different levels are also used with dynamic mutations at different stages of the development. The mutations are implemented in pairs of introducing a pattern or removing a pattern. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. The mutations are the following:

- introduce/remove message dispatcher
- communicate/remove communication through dispatcher
- introduce/remove server
- introduce/remove façade
- introduce/remove mediator
- introduce/remove strategy
- introduce/remove proxy.

Apart from introducing the dispatcher, all the other mutations are implemented at supergene level by updating the information in the dispatcher connection or pattern field, as well as the interface and class fields when needed. However, as each pattern concerns more than one responsibility, all the supergenes involved in the implemented pattern must be mutated accordingly. For example, the proxy pattern is introduced by constructing a new ProxyPattern instance with data of the technical class and interface needed by the proxy. The pattern fields of the responsibility called through the proxy and the responsibilities calling the proxy are then updated so that they now contain the new ProxyPattern instance.

As adding a dispatcher, however, does not consider any particular responsibility (supergene), it cannot be implemented as a gene level mutation, but must affect the whole system. This means that the mutation must be implemented at chromosome level, where a dummy gene is added to the existing chromosome. In this gene, only the field containing information of the dispatcher has a value different from 0 or null. When a chromosome contains such a gene which "carries" the dispatcher, its supergenes can later on be subjected to the mutation where individual responsibilities are made to communicate through the dispatcher.

The crossover operation is implemented as a traditional one-point crossover. However, as patterns always concern more than one responsibility, it is possible that during a mutation or crossover, an existing pattern may be broken.
Because of this, after each mutation and crossover operation, the resulting chromosome(s) are subjected to a corrective operation, which ensures that the model stays coherent. In the case of the crossover, it is decided that the left side of the offspring is always legal, and the adjustments are made to the right side of the crossover point. In addition to ensuring that the patterns present in the system stay coherent and "legal", the corrective function also checks that the model conforms to certain architectural laws. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher), and state some basic rules regarding architectures, e.g., no responsibility can implement more than one interface. These laws ensure that no anomalies are brought to the model.

Mutation and crossover indexes are chosen randomly. An exception is made in case of patterns which favor highly variable responsibilities (dispatcher communication and strategy) and proxy, which favors responsibilities that have a relatively high call variability value. In these cases, the chances of each particular responsibility to be subjected to such mutations increases linearly in relation to the respective property value, e.g., if responsibility \( r_i \) has a variability value 3, it is three times more likely to be subjected to a strategy or dispatcher mutation than responsibility \( r_j \) with variability value 1.

The actual mutation probabilities are given as input. Selecting the mutation is made with a "roulette wheel" selection [11], where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover are also included in the wheel. Crossover is treated as a mutation in the selection, as the probability of being subjected to crossover increases in relation to how high an individual's fitness is in the population. When the crossover is included in the same wheel as the mutations, it also leads to smaller mutation slices, as they need to accommodate to fit in the larger crossover slice. Thus, including crossover and mutation in the same wheel increases the chances that highly fit individuals remain untouched throughout generations. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

We have also adopted the idea of dynamic mutation probabilities. The three different levels of patterns can be further used in mutation probabilities by favoring the high-level patterns in the beginning of evolution, as the basic structure of the model is being defined. We then move on to favor the medium-level patterns, and in the very end of evolution, the low-level patterns are favored to ensure as high-tuned a solution as possible. In practice, the probabilities of high-level pattern introductions are decreased after 1/3 of the generations have passed, and the probabilities of medium-level pattern introductions are increased respectively. The same procedure is done between medium-level pattern introductions and low-level pattern
introductions after another 1/3 of the generations.

3.4 Fitness Function

Selecting an appropriate fitness function is probably the most demanding task with any genetic algorithm application when there is no clear value to measure from the solutions. In the case of software architecture, evaluation is even more difficult. In real world, evaluation of software architecture is almost always done manually by human designers, and metric calculations are only used as guidelines. Also, no two architects may ever agree on a unique quality for certain architecture, as evaluation is bound to be subjective, and different values and backgrounds will influence the outcome of any evaluation process. However, for the genetic algorithm to be able to evaluate the architecture, a purely numerical fitness value must be calculated.

In a fully automated approach, no human interception is allowed, and thus the fitness function needs to be based on metrics. The selection of metrics may be as arguable as the evaluations of two different architects on a single software architecture. The reasoning behind the selected metrics in this approach is that they have been widely used and recognized to accurately measure some quality aspects of software architecture. Hence, the metrics are chosen so that they measure quality aspects that can be seen as "most agreed upon" in the real world, and singular values can be seen as accurate as possible. However, the combination of metrics and multiple optimization is another problem entirely, as not many quality values can be optimized simultaneously.

The fitness function is based on software product metrics, most of which are from the metrics suite introduced by Chidamber and Kemerer [3]. These metrics, especially coupling and cohesion, have been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear "sub-fitnesses" for modifiability and efficiency, both of which are measured with a positive and negative metric. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the fitness function into sub-functions answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the architecture. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. When $w_i$ is the weight for the respective sub-fitness $sf_i$, the fitness function $f(x)$ for chromosome $x$ can be expressed as

$$f(x) = w_1 \times sf_1 - w_2 \times sf_2 + w_3 \times sf_3 - w_4 \times sf_4 - w_5 \times sf_5.$$
Here, $sf_1$ measures positive modifiability, $sf_2$ negative modifiability, $sf_3$ positive efficiency, $sf_4$ negative efficiency and $sf_5$ complexity. The sub-fitness functions are defined as follows ($|X|$ denotes the cardinality of $X$):

\[
 sf_1 = |\text{interface implementors}| + |\text{calls to interfaces}| + \\
 (|\text{calls through dispatcher}| \times \\
 \sum (\text{variabilities of responsibilities called through dispatcher})) - \\
 |\text{unused responsibilities in interfaces}| \times 10,
\]

\[
 sf_2 = |\text{calls between responsibilities in different classes that do not happen through a pattern}|,
\]

\[
 sf_3 = \sum (|\text{depending Responsibilities within same class}| \times \\
 \text{parameterSize} + \\
 \sum (|\text{used Responsibilities in same class}| \times \text{parameterSize} + \\
 |\text{depending Responsibilities in same class}| \times \text{parameterSize}))
\]

\[
 sf_4 = \sum \text{ClassInstabilities} + (|\text{dispatcherCalls}| + |\text{serverCalls}|) \times \sum \text{callCosts},
\]

\[
 sf_5 = |\text{classes}| + |\text{interfaces}|
\]

The multiplier 10 in $sf_1$ means that having unused responsibilities in an interface is almost an architecture law, and should be more heavily penalized.

### 3.5 Selection

A selection operation is needed as the size of the population should be the same at the start of each generation, but through crossover the amount of individuals grows. Selecting the individuals for each generation is made with the roulette wheel method. Each individual is given a "slice" on the wheel. The size of the "slice" is based on how high the individual’s fitness is in the population. Thus, the "slices" are not proportioned according to raw numerical fitness differences, but based on order of fitnesses. In addition to roulette wheel selection an elitist approach is used, where 10 individuals with the best fitnesses are kept after each generation.

This approach takes into consideration that the quality difference between two individuals may not be the same as would appear when examining the raw numerical difference between the fitness values. By using elitism as an addition to the roulette wheel selection, the development of fitness values is also more secured.
4. Experiments

In this section we present the results achieved through experiments with our approach. The consistency of the fitness development of each test run has been showed in our previous research [15], and thus the fitness curves presented here can reliably be calculated as averages. We have first calculated the average fitness of the 10 best individuals of each generation, thus achieving the fitness development curve of the fittest individuals in each run. The actual fitness curves are then achieved by calculating the average development of five test runs. The mutation probabilities used for the test runs are same in all the experiments discussed here, and were set after exhaustive testing.

As using a model to give a basic structure for each solution greatly affects the amount of variation between individuals, we first experimented how the population size affected the fitness curve. As can be seen in Fig. 5, with a population of 50 the fitness curve actually turns downwards already after some 200 generations, while with populations of 100 and 150 the development is more stable. Thus the forthcoming tests were made with a population of 100, as it ensured a further developing fitness curve by having more variability in the population. Increasing the population size even further to 150 did not seem necessary, as the fitness values were not noticeably better than those achieved with a population of 100.

![Fig. 5: Different population sizes](image_url)

Another common variable to all genetic algorithms in addition to population size is the number of generations the algorithm runs through. We tested our algorithm with 1000 generations, and the resulting fitness curve is depicted in Fig. 6. As can be seen, the fitness curve continues to steadily achieve higher values up until around 625 generations, after which its development plunges. This would indicate that running the algorithm with an exceptionally large number of generations would not be beneficial. The following tests are all made with either 250 or 500 generations, when we can be sure that the fitness curve has not passed its optimum and started to...
descend.

![Fig. 6: 1000 generations](image1)

In Figures 5 and 6 the fitness curve indicates the development of the overall fitness, i.e., modifiability, efficiency and complexity combined. As has been discussed in Section 3, balancing the different sub-fitnesses is a demanding optimization task, and thus it is interesting to see how weighting one quality attribute over another will affect the development of their individual fitness curves. In Fig. 7 we have separated the fitness curves for modifiability and efficiency from the overall fitness.

![Fig. 7: Modifiability weighted](image2)

In this test, modifiability was valued 10 times higher than efficiency, which results in the modifiability curve increasing quite rapidly, while the efficiency curve does not develop at all, but achieves quite low values throughout the generations. This is expected, as when modifiability is valued, solutions with high efficiency values do not survive to next generation, and are definitely not in the top of any generation. A similar test was also made where efficiency was valued 10 times higher than modifiability; the fitness curves for this experiment are shown in Fig. 8.
The fitness curves achieved while valuing efficiency are quite different from those achieved when weighting modifiability: in the case of efficiency, neither the efficiency nor the modifiability curve develops at all. Valuing efficiency can only be seen in the different ranges of values, as the efficiency curve achieves high positive values while the modifiability curve never goes above 0. The reason for the stable efficiency curve can be found in the initial population: as the initial population only contains one pattern per model, and even a solution where there are no patterns, these models achieve such high efficiency values that it is simply not possible to top them as each time a pattern is added, the efficiency of the solution decreases.

In addition to studying the effects of weighting the different quality attributes, we have studied the effect of the dynamic mutation probabilities, discussed in Section 3, to the fitness curve. Fitness curves achieved with dynamic and static mutation probabilities, when both quality attributes are valued equally high, are shown in Fig. 9. The development of the overall
fitness is quite similar, and the tests with static mutations actually achieve slightly higher fitness values toward the end.

However, when modifiability is weighted over efficiency, we can see that the modifiability fitness clearly benefits from the dynamic mutation probabilities, as shown in the curve in Fig. 10, where the modifiability sub-fitness curves are shown from experiments with dynamic and static mutation probabilities.

As these experiments show, the modifiability of the solutions can quite
straightforwardly be affected by weighting the modifiability sub-fitness functions over the efficiency sub-fitnesses. The implemented dynamic mutation probabilities also benefit the modifiability fitness, which makes their usage justified.

The achieved solutions illustrate the creative power of the genetic algorithm: there is a high degree of variability between the solutions, and different patterns are implemented fearlessly. However, although some good decisions are found in parts of each solution, no single solution is achieved that would instantly impress an expert as having overall top quality.

Fig. 11 illustrates a proposed solution. In this solution, the dispatcher is central as many components communicate through it. A large component is turned into a server, and the user interface further uses a proxy to call the responsibilities provided by the server. Strategies are also used for several highly variable responsibilities (the responsibility and its class are given in parentheses).

5. Conclusions

We have presented a novel approach for model transformations from CIM to PIM using a genetic algorithm that implements design patterns to the given domain model and gives a solution model as a UML class diagram. Our work differs from that of Amoui et al. [2] by using the model-driven approach, not having a ready-made architecture to start from and not storing information of used patterns as a sequence of transformations.

The results show that promising solutions can be achieved, as the quality of solutions improves over generations and the actual design choices seen in the class diagrams are sensible in many cases. The achieved solutions prove that a conceivable PIM level architecture can be achieved from very high-level requirements, when only given a domain model as a guideline of the architecture’s structure.

The biggest limitations in our approach at the moment are the small number of applied patterns and the inconsistency of quality in proposed solutions. The inconsistency of quality is most likely due to the difficulty of finding a suitable compromise between the selected quality estimators. Also, the implementation of dynamic mutations should be studied further, as it may also have a negative impact if such design patterns that are not possible to implement in the architecture are favored for a long time.

6. Future Work

Our plans for future work include implementing a simulated annealing algorithm for comparison, a multi-objective fitness function to achieve a more balanced architecture in terms of different quality values, and studying different ways of applying the dynamic mutations.

Implementing the simulated annealing algorithm will show whether other
search-based techniques could also be used for model transformations, or whether the genetic algorithm is the only one that is "sophisticated enough" to handle the intricate details regarding software architectures. If the simulated annealing approach provides good results, it may be considered whether a combination of the two algorithms could be used to achieve optimal results.

Presumably, a multi-objective fitness function will better address the problem of evaluating a software architecture. As discussed, it is difficult to optimize several quality aspects at the same time. A multi-objective fitness function will enable to better choose a solution that achieves the best compromise between different quality measures. In addition, while further improving mutations and the fitness evaluation will most probably improve the quality of results, more design patterns are also to be added in order to gain access to a wider collection of design choices.

References


