Genetic Programming meets Model-Driven Development

Thomas Weise, Michael Zapf, Mohammad Ullah Khan, Kurt Geihs
University of Kassel
Wilhelmshöher Allee 73
34121 Kassel, Germany
weise|zapf|khan|geihs@vs.uni-kassel.de

Abstract

Genetic programming is known to provide good solutions for many problems like the evolution of network protocols and distributed algorithms. In such cases it is most likely a hardwired module of a design framework that assists the engineer to optimize specific aspects of the system to be developed. It provides its results in a fixed format through an internal interface. In this paper we show how the utility of genetic programming can be increased remarkably by isolating it as a component and integrating it into the model-driven software development process. Our genetic programming framework produces XMI-encoded UML models that can easily be loaded into widely available modeling tools which in turn posses code generation as well as additional analysis and test capabilities. We use the evolution of a distributed election algorithm as an example to illustrate how genetic programming can be combined with model-driven development. This example clearly illustrates the advantages of our approach – the generation of source code in different programming languages.

1 Introduction

Genetic programming is the automated generation of computer programs by artificial evolution. Among many other applications, it has successfully been used for creating protocols with minimum communication costs [9, 31, 6], software testing [8], and evolving hardware/software co-designs [7].

In our previous work [28] we applied genetic programming in the area of distributed computing. We were able to evolve proactive aggregation protocols for large-scale distributed systems and to find local algorithms for sensor networks that create a specified global behavior [26, 27].

In that former work, genetic programming was integrated as a fixed component into a software system, as is the case in most of the other current activities in this area. As a hardwired module, it provides its results through an application-internal interface or at least in a format that usually can only be used by exactly this system. Especially for the evolution of algorithms, such a restriction makes no sense. Algorithms are general, platform-independent descriptions of processes. It would thus be more reasonable if they were returned in an independent format. Analogous issues can be observed for many other applications of evolutionary algorithms.

In this paper we discuss how the results of genetic programming can be represented in a standardized format which allows them to be analyzed, transformed, and tested. In the Section 2 we give an introduction into the process of genetic programming and MDD, and how both can be combined. In Section 4 we introduce the evolution of a distributed election algorithm as an example for genetic programming. Based on this example we demonstrate how such automatically created algorithms can be stored as XMI-encoded UML models in Section 5. In Section 6 we apply standard means of model transformation by employing *MOFScript*, a model to text transformation language, to create source code from these models. 7 concludes the article with some prospects on future work.

2 Genetic Programming and MDD

In this section we will elaborate on genetic programming and model-driven development in general and then describe how both technologies can be combined – with a minimum amount of effort. Figure 1 illustrates the versatility of this new approach.

2.1 Monolithic Genetic Programming

In many cases of genetic programming, the internal representation of the solution candidates differs from the format in which the final results will be delivered. When breeding distributed algorithms, for instance, the results of the evolution should be delivered as C code. If we also

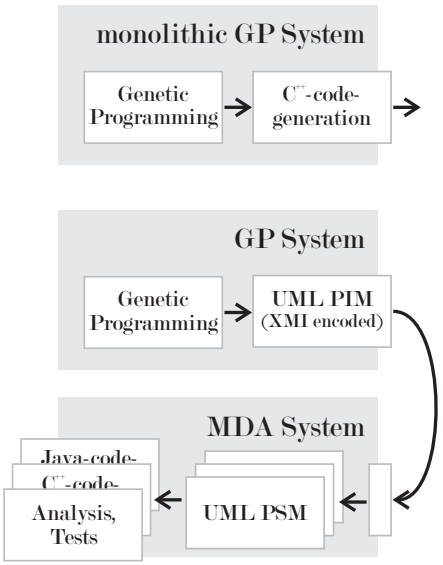

Figure 1. A monolithic genetic programming system with hard-wired output compared to genetic programming with XMI output.

breed the programs in C code, some difficulties arise, especially when performing multi-objective optimization that also takes non-functional criteria into consideration. Limiting the runtime of a process in terms of the exact number of executed instructions is practical impossible. Measuring the precise memory consumption is also complicated, since there is always some overhead caused by the libraries used. For testing a distributed algorithm, one single execution of the program does not suffice. Instead, multiple, message-exchanging instances have to run in parallel. Obtaining reproducible results then becomes impossible because of, for instance, the scheduling done by the operating system.

All these complications can easily be avoided when evolving the algorithms in an AST-like[1] representation, as introduced by Koza in his trailblazing work [16]. Genetic operators now can work on the trees with a much higher chance of producing useful offsprings. The algorithms do not need to be compiled anymore but can be interpreted directly, since the instructions in the AST can be executed on simple virtual machines. A maximum runtime of the algorithms being tested can easily be enforced by limiting the count of interpretation steps. Parallelization can be imitated by performing one interpretation step for each virtual machine in each simulated time unit.

There exist similarities to this situation in almost all applications of genetic programming. In each of these cases, a translation of the internal representation of the individuals into a desired output format takes place. The disadvantage of this direct translation is the fixed binding of the phenotypes of the evolutionary algorithms to one single applicable representation. Transforming the results into an intermediate format which can be processed by different tools would add great flexibility while only requiring little additional work.

2.2 Model-Driven Development

Model-Driven Development (MDD) [2] and the Model-Driven Architecture (MDA) [4, 22] guided by the OMG[2] are key technologies for software and system design. Developing applications starts with an elaborate modeling phase instead of writing program code from the start. The model is a simple and intuitive specification of the application and can be transformed into program code step-by-step: At the beginning, a platform-independent model (PIM) is created which only describes the semantics, abstracting from a specific technology. The PIM is transformed into a more fine-grained, platform-specific model (PSM) bound to a certain target technology. In the final step, the PSM is transformed to source code in a programming language which can then be compiled and deployed.

MDA recommends using the Unified Modeling Language (UML) [10] for model specification, the most popular and wide-spread modeling language in software technology. The syntax of UML is defined by the UML metamodel which in turn is an instance of the Meta-Object Facility (MOF) [15]. MOF models can be exchanged between different MDD tools in the standardized XML Metadata Interchange format (XMI) [1]. For UML models a special schema exists, allowing them to be serialized as XMI.

The basic idea is that these model transformations to a more specific platform may be performed semi-automatically by tools, preventing many programming errors and thus decreasing software development costs. There exist various methods to perform model transformations, spanning from QVT (Query, View and Transformation, model-to-model transformation) [14], MOF-Script (model-to-source code) [17] to direct XML transformations using XSLT style sheets.

2.3 Combining MDD and GP

After translating the results of genetic programming into XMI-encoded UML models, they can be imported into a wide range of MDD tools. It becomes now possible to use their code generation and transformation abilities to translate the models into implementations for a variety of target platforms and programming languages. Furthermore, it enables us to perform additional optimization, tests, and anal-

[1] Abstract Syntax Tree, see `http://en.wikipedia.org/wiki/Abstract_Syntax_Tree`

[2] Object Management Group, `http://www.omg.org/`

yses on the algorithms evolved using standardized software engineering methods.

Genetic programming will in most cases not create whole applications. Instead, it will just evolve a certain functionality which can, for example, be encapsulated in a module. It is quite possible that the application which this module will later be part of is also modeled in UML with a software design tool. Hence, it is possible to integrate the bred algorithms directly into its model, achieving a consistent view of the whole system in one common specification. The software engineer no longer needs to patch different application parts together.

It also becomes much easier to combine different genetically evolved algorithms. One example for such a situation would be that an algorithm is needed which transmits messages along the spanning tree of a sensor network if the sensors detect the occurrence of a specific event. Since sensor measures are always noisy, more than one adjacent node should detect the event before its occurrence can be regarded as confirmed. It is very unlikely that one could genetically evolve an algorithm that is able to perform this task as a whole. If we proceed according to the divide-and-conquer scheme and split the task into different problems, solving them stepwise, chances are good that we will be able to obtain a fully functional system. An algorithm could be evolved that automatically finds and maintains a spanning tree, using genotypes and phenotypes suitable for graph problems. Another algorithm can be grown that optimizes the detection of the event, using different genotypes, phenotypes, parameters and simulation methods. If both results are returned as XMI, they can be combined in a software design tool with very little effort.

3 Related Work

In the last two decades, the area of automated protocol generation using evolutionary algorithm has been visited by different researchers. Yamaguchi *et al.* concentrate on finding an optimum message exchange [30, 9] in order to reduce communication costs. Yamamoto and Tschudin were able to evolve protocols based on their experimental Fraglet model [25, 31]. Special applications like efficient broadcasting were topics of studies like those of Comellas and Giménez [6]. The evolution of distributed algorithms itself is indeed a new area. One important perspective on the topic is given by Qureshi [20, 21] who discusses the evolution of agents. He showed that communication behavior could be evolved along with agent cooperation and that it is possible that agents can "learn" how to communicate with already existing agents by genetic programming.

Currently, a lot of research is done on the software engineering sector in order to improve the model-driven development approach [2]. For the model-to-text transformation, many different approaches exist [3, 18] among which the MOFScript [17] transformation (envisaged to become an OMG standard) is one of the most promising. Interesting in the context of this work is the masters thesis of Jim [13] who describes how behavioral UML models can be translated to Java source. She also uses XMI to transport the model data but instead of applying MOFScript, an XSLT transformation performs the source generation.

With its current state of maturity, MDD gains value for applications with more scientific character like the design of multi-agent [19] or real-time systems [23]. No research however has yet been conducted on the integration of genetic programming into the model-driven software development process itself.

4 Evolving Distributed Algorithms

As already mentioned in the introduction, the focus of our work is put on utilizing genetic programming for distributed systems. In this section we give an introduction on how a desired behavior of a network can be translated into algorithms that run on its nodes by genetic programming [26]. After elaborating on the key aspects of this topic, we give a small example which we will later use for demonstration purposes.

Today we experience a growing demand for distributed systems of sensors [5]. These *sensor networks* are composed of a large number of sensor nodes, small devices that gather information about their environment and transmit it wirelessly. They are restricted in resources like memory size, processing speed, and – most importantly – battery power. The communication among them is not reliable, and the topology of their network is volatile. Distributed algorithms for sensor nodes should thus be robust and as efficient as possible.

To evaluate the fitness of such algorithms we simulate whole sensor networks. In our simulation, sensor nodes are represented as virtual machines with a fixed-size memory architecture, asynchronous I/O, and a Turing-complete instruction set (similar to those discussed in [24, 29]).

As in reality, many nodes (i.e. the virtual machines) run asynchronously in the simulation at approximately the same speed, which may differ from node to node and cannot be assumed to be constant. The communication is unreliable, and transmissions are broadcasted like radio waves that spread into all directions and are received by any node in range.

We apply multi-objective genetic programming since it allows us to optimize the algorithms for different aspects. In the functional objective functions, we perform a comparison of the observed behavior of the simulated network (running the evolved algorithms) with the desired global be-

```
called on startup        procedure_0
store 1st variable into  0: push mem[0]
output buffer            1: some useless operations used
                         2: to delay and, as a consequence,
                         3: reduce transmissions in the
                         4: simulated/evaluated time span
send output buffer       5: send
go back to start         6: goto 0

called asynchronously when
a message comes in       procedure_1
compare the known and
the received value       0: zf = (params[0] < mem[0])
if no improvement then exit 1: if zf then goto 3 // ≡ exit
exchange values          2: xchg params[0], mem[0]
```

Figure 2. One of the non-dominated solutions.

havior. The evolutionary algorithm hence transforms global behavior of a network into local behavior of single nodes, thus effectively creating emergence. Non-functional objective functions are added to foster the economical use of resources, especially for minimizing energy-expensive communication.

4.1 Evolving an Election Algorithm

Election means to select one node out of a group of nodes whereby at the end all nodes should have knowledge of the ID of this special node. In distributed systems, an election is performed, when e.g. a special node is needed to act as a communication relay or as a coordinator. For the purpose of this example, we assume that the active node with the maximum ID shall be selected.

In the simulations, we initialize all virtual machines with their own ID in the first memory cell. If an algorithm makes progress, the nodes should have assumed greater (valid) IDs after some time. A fully functional algorithm would accomplish that the first memory cells of all nodes contain the maximum ID. If the algorithm is also resource-friendly, it should reach this goal with as few transmissions as possible.

Therefore, we apply three objective functions: the first function, subject to maximization, is the aggregation of all valid IDs stored in the first memory cells of the nodes over all time steps (see equation 1).

$$f_1 = \sum_{\forall\, time} \sum_{\forall\, nodes\ n} \begin{cases} n.mem[0] & if\ valid(n.mem[0]) \\ 0 & otherwise \end{cases} \tag{1}$$

It is an indicator both for the functionality as well as the convergence speed of the evolved algorithms. The second and third objective functions both optimize non-functional aspects. They are used to minimize the number of messages sent and the instruction count of the algorithms.

One of the results obtained performing genetic programming guided by these objective functions is displayed in Figure 2. The algorithm consists of two parts: a procedure that is called when the node starts up (`procedure_0`) and an asynchronously called, interrupt-like routine which receives incoming messages (`procedure_1`). In this simple algorithm, the nodes constantly broadcast the greatest ID they have encountered in a loop, reducing the network traffic only by performing dummy work. By constantly sending (probably unnecessary) messages and thus not being optimal, the algorithm ensures that nodes started later will still take part in the election.

5 Creating a PIM

The example algorithm introduced in the previous section has been evolved in an internal, pseudo code-like representation – exactly as displayed in Figure 2. The next step is to transform this pseudo code into a suitable UML model and to create XMI-formatted output which will be used as input for creating source code.

We have to analyze which entities must be specified in order to describe an algorithm completely. In principle, each algorithm is constituted by three parts:

1. the data structures the algorithm works on,
2. the primitive instructions used that work on that data structures, and
3. the control flow (i.e. the sequence of primitive instructions).

While the control flow is a result of genetic programming, the data structures (except for the memory size) and the instruction semantics are predefined for the simulated virtual machines.

In the next subsections, we discuss how these three parts can be specified consistently using UML 2.

5.1 Control Flow Model

The control flow of an algorithm can easily be represented using an activity model in UML. In general, the single procedures of an algorithm are modeled as compound activities including a set of simple actions. Each of them corresponds either to the execution of a single instruction on the virtual hardware, or to a branch to another procedure. Transitions mark the sequence of the instructions inside the procedures and also denote unconditional jumps whereas conditional jumps are represented by decision nodes.

Since we evolve distributed algorithms, we also need means to model the sending and receiving of transmissions: Broadcasting a message is modeled by sending a signal. The asynchronous mechanism of receiving messages is modeled as a procedure running in parallel that contains

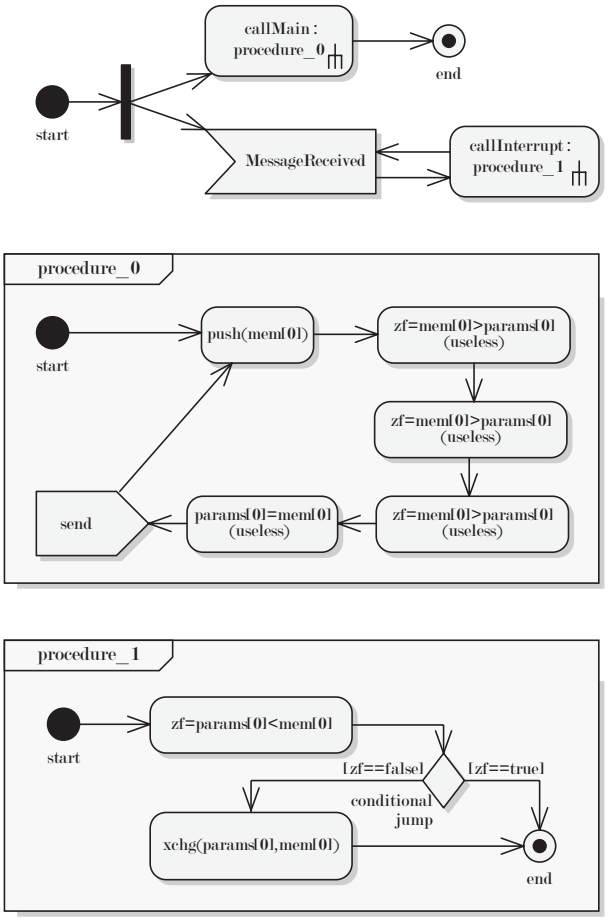

Figure 3. The control flow of the evolved election algorithm.

an infinite loop of a receive event action followed by a procedure call to the message handler. Figure 3 illustrates the control flow of the example algorithm from Section 4.1.

5.2 Data Model

The nodes of a distributed system as well as the virtual machines that we use to simulate our evolved algorithms can be regarded as instances of a class and are therefore modeled in a class diagram. They have a fixed-size memory, a stack and a flag register. Parameters for procedures (also integer numbers) are stored in an additional parameter list. The interrupt-like procedure, which is invoked whenever a message comes in, finds the message stored in this array. These data structures can be modeled as member variables: the memory `mem`, the parameter array `params` and the stack `stack` are lists of integer numbers, whereas the flag `zf` is a boolean value. It should be noted that modeling nodes as classes does not necessarily imply the application of object-

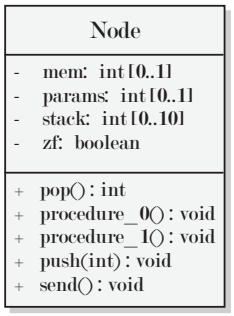

Figure 4. The data structure definition of the evolved election algorithm.

oriented programming. Classes are just means to specify data structures along with operations working on them. Figure 4 shows the class diagram of the example algorithm.

5.3 Modeling the Primitive Operations

Finally, the primitive operations used by the algorithms must be specified. The pseudo code used in Figure 2 contains three different types of operations:

1. Operations that modify the control flow like `goto` or procedure calls are already defined in the activity diagram of Section 5.1 – as transitions, decision nodes, or behavior-invoking actions.

2. Operations with trivial, self-defining semantics like arithmetic operations or value assignments require no additional specifications. The operations `zf`, `params`, and `mem` within `(zf=params[0]<mem[0])` are already defined as data structures in Section 5.2. Specifying the operators "`<`" and "`=`" would involve re-using either of them again or similar operators, and is therefore not practical.

3. Other operations like pushing something onto the stack (`push`) or transmitting all data currently on the stack as message (`send`) need to be defined more precisely. Their semantics can be specified as post-conditions using the Object Constraint Language OCL [12].

5.4 Using Profiles

UML 2 allows for customizing a metamodel using profiles. Profiles consist of stereotypes which extend metaclasses and may introduce additional constraints helping to verify model consistency.

Stereotypes are most effectively used when instances of the metamodel have an arbitrary number of node types. In

that case, node types can be specified by one or more stereotypes.

In our approach, the UML model which is created as a result of the artifical evolution comprises a limited and predefined set of node types. Hence, profile application has only limited benefits, so we rely on the base UML metamodel.

6 Transforming the UML Models

One of the main goals of the MDD-based development is the possibility of automatically transforming the model to code using transformation tools. In our work, we support such automatic source code generation using the *MOFScript* language, a model-to-text transformation language obtained from the MODELWARE project [22]. MOFScript is currently a candidate in the OMG RFP process on MOF Model-to-Text Transformation [17] and is intended to cover the aspects required in the context of text generation in software engineering.

At present, MOFScript supports transforming UML models created using an EMF-based[3] implementation of a subset of OMG UML 2.x metamodel, obtained from the *Eclipse UML2 Project* [11]. The Eclipse UML2 Project does not support any graphical notation but there are tools like *Borland Together Architect*[4] and *Omondo*[5], which can help to visualize the models.

In the following, we show a Java source code fragment generated solely using MOFScript transformations. This code corresponds to the model of `procedure_1` in Figure 3. During the transformation, different actions are identified as Activity nodes and they are given numbers, starting from `0`. In this particular case, the action `zf=params[0]>mem[0]` is denoted as action `0`, the conditional jump as action `1` and `xchg(params[0], mem[0])` as action `2`, while the initial action is not numbered. The `for` loop in the generated code always starts with the number of the action that executes right at the beginning of the procedure. After each action, the control flow transcends to the next one by assigning its number to the `ip` variable, until a value greater or equal to the total count of actions (in this case 3) is selected which will lead to the termination of the loop. This more complex approach serves to emulate jump instructions that are not available in Java.

```
1 public class Node2 {
2   private final int[] mem;
3   private final int[] stack;
4   private int stackPtr;
5   private int[] params;
6   private boolean zf;
7
8 ...
9   public void procedure_1() {
10    int ip;
11    for (ip = 0; ip < 3;) {
12      switch (ip) {
13      case 0: { zf = params[0] < mem[0];
14                ip = 1;
15                break; }
16      case 1: { if (zf) ip = 3;
17                else    ip = 2;
18                break;  }
19      case 2: { xchg(params, 0, mem, 0);
20                ip = 3;
21                break; }
22      }
23    }
24  }
25 ...
26 }
```

MOFScript is capable of producing all sorts of text output, so source code in other programming languages like C or C++ can be as well produced with very little effort. The result of such a transformation to C++ is shown below. The existence of jump instructions makes this result much more readable and also grants a higher performance. Most of the `goto` instructions in the code below could have been omitted, but including them illustrates the analogy to the Java example.

```
1 ...
2 int[] mem;
3 int[] stack;
4 int stackPtr;
5 int[] params;
6 bool zf;
7
8 ...
9 void procedure_0() {
10    a0: push(mem[0]);             goto a1;
11    a1: zf = mem[0] > params[0]; goto a2;
12    a2: zf = mem[0] > params[0]; goto a3;
13    a3: zf = mem[0] > params[0]; goto a4;
14    a4: params[0] = mem[0];      goto a5;
15    a5: send();                  goto a0;
16    a6:;
17 }
18
19 void procedure_1() {
20    a0: zf = params[0] < mem[0]; goto a1;
21    a1: if(zf) goto a3;
22        else goto a2;
23    a2: xchg(params, 0, mem, 0); goto a3;
24    a3:;
25 }
```

Generally, MOFScript can make use of UML profiles by using stereotypes as key values for node searches. As profiles are applied to the model, the XMI output also changes. MDD applications to be used in this tool chain are therefore required to support user-defined profiles. However, the structure of the created model does not require extensive searches. Together with our discussion in section 5.4, we decided not to apply profiles within the transformation pro-

[3] Eclipse Modeling Framework
[4] `http://www.borland.com/de/products/together/`
[5] `http://www.omondo.com/`

cess.

7 Conclusions

The goal of our work is to prove the utility of genetic programming as a tool for developing distributed algorithms. Recently we have contributed two successful applications [26, 28] in that area.

Finding cases where genetic programming can assist in creating distributed systems is, however, only one step. It is likewise important to incorporate its results into the application development.

In this paper we have shown that the utility of genetic programming can remarkably be enhanced by integrating it into the model-driven development process. Furthermore, we have demonstrated that such integration can be accomplished using existing and widely available tools.

In the future we will perform research mainly in two directions, specifically extending the use of genetic programming to other domains in the context of distributed systems as well as enhancing its integration into the application development process according to current software engineering practices and tools.

References

[1] *ISO/IEC 19503:2005-11*. Number 11. 2005.

[2] J. O. Aagedal, J. Bezivin, and P. F. Linington. Model-driven development. In J. Malenfant and B. M. Ostvold, editors, *ECOOP 2004 Workshop Reader*, volume 3344 of *LNCS*. Springer-Verlag, Jan 2005.

[3] M. Albert, J. Muñoz, V. Pelechano, and O. Pastor. Model to Text Transformation in Practice: Generating Code from Rich Associations Specifications. In *Proceedings of ER (Workshops) – Advances in Conceptual Modeling - Theory and Practice, ER 2006 Workshops BP-UML, CoMoGIS, COSS, ECDM, OIS, QoIS, SemWAT, Tucson, AZ, USA, November 6-9, 2006*, volume 4231 of *Lecture Notes in Computer Science*, pages 63–72. Springer, 2006.

[4] U. Aßmann, M. Aksit, and A. Rensink, editors. *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*. Springer, 2005.

[5] C.-Y. Chong and S. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, Aug 2003.

[6] F. Comellas and G. Giménez. Genetic programming to design communication algorithms for parallel architectures. *Parallel Processing Letters*, 8(4):549–560, 1998.

[7] M. N. de Miranda, R. N. B. Lima, A. C. P. Pedroza, and A. C. de Mesquita. HW/SW codesign of protocols based on performance optimization using genetic algorithms. Technical report.

[8] A. Derezińska. Advanced mutation operators applicable in C# programs. In K. Sacha, editor, *Software Engineering Techniques: Design for Quality – IFIP Working Conference on Software Engineering Techniques - SET 2006*, pages 283–288. Springer, 2006.

[9] K. El-Fakih, H. Yamaguchi, G. Bochmann, and T. Higashino. A method and a genetic algorithm for deriving protocols for distributed applications with minimum communication cost. In *Proceedings of Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems*, Nov 1999.

[10] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, third edition, Sep 2003.

[11] A. Gerber and K. Raymond. MOF to EMF: there and back again. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 60–64, New York, NY, USA, 2003. ACM Press.

[12] H. Hussmann and S. Zschaler. The object constraint language for UML 2.0 – overview and assessment. *Upgrade*, Apr 2004.

[13] S. L. Jim. From UML diagrams to behavioural source code. Master's thesis, Sep 2006.

[14] S. R. Judson, D. L. Carver, and R. B. France. A metamodeling approach to model transformation. In *OOPSLA'03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 326–327, New York, NY, USA, 2003. ACM Press.

[15] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20, New York, NY, USA, 2006. ACM Press.

[16] J. R. Koza. *Genetic Programming, On the Programming of Computers by Means of Natural Selection*. A Bradford Book, The MIT Press, Cambridge, Massachusetts, 1992 first edition, 1993 second edition, 1992.

[17] J. Oldevik, T. Neple, R. Grønmo, J. Aagedal, and P. Desfray. Second revised submission for MOF model to text transformation language RFP, Nov 2005. developed by SINTEF in the European IP project 511731 MODELWARE.

[18] J. Oldevik, T. Neple, R. Grønmo, J. Ø. Aagedal, and A.-J. Berre. Toward Standardised Model to Text Transformations. In *ECMDA-FA, Proceedings of Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005*, volume 3748 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2005.

[19] J. Pavón, J. J. Gómez-Sanz, and R. Fuentes. Model Driven Development of Multi-Agent Systems. In *ECMDA-FA, Proceedings of Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006*, volume 4066 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006.

[20] M. A. Qureshi. Evolving agents. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 369–374, Stanford University, CA, USA, 28–31 1996. MIT Press.

[21] M. A. Qureshi. *The Evolution of Agents*. PhD thesis, University College, London, UK, Jul 2001.

[22] A. Rensink and J. Warmer, editors. *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, volume 4066 of *Lecture Notes in Computer Science*. Springer, 2006.

[23] D. C. Schmidt. Model driven development for distributed real-time and embedded systems. In *MoDELS, Proceedings of Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005*, volume 3713 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.

[24] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 136–141, Orlando, Florida, USA, 1994. IEEE Press.

[25] C. Tschudin. Fraglets - a Metabolistic Execution Model for Communication Protocols. In *Proceedings of 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS)*, Jul 2003.

[26] T. Weise and K. Geihs. DGPF - an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In B. Filipič and J. Šilc, editors, *Proceedings of the Second International Conference on Bioinspired Optimization Methods and their Application, BIOMA 2006*, International Conference on Bioinspired Optimization Methods and their Application (BIOMA), pages 157–166. Jožef Stefan Institute, Ljubljana, Slovenia, Oct 2006.

[27] T. Weise and K. Geihs. Genetic programming techniques for sensor networks. In *Proceedings of 5. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, pages 21–25, Jul 2006.

[28] T. Weise, K. Geihs, and P. A. Baer. Genetic programming for proactive aggregation protocols. In B. Beliczyński, A. Dzieliński, M. Iwanowski, and B. Ribeiro, editors, *Proceedings of the 8th International Conference on Adaptive and Natural Computing Algorithms ICANNGA'07, Part 1*, volume 4431 of *Lecture Notes in Computer Science (LNCS)*, pages 167–173. Springer Berlin Heidelberg New York, Apr 2007.

[29] J. Woodward. Evolving turing complete representations. In *Proceedings of Congress on Evolutionary Computation*, volume 2, pages 830–837, Dec 2003.

[30] H. Yamaguchi, K. Okano, T. Higashino, and K. Taniguchi. Synthesis of protocol entities' specifications from service specifications in a petri net model with registers. In *International Conference on Distributed Computing Systems*, pages 510–517, 1995.

[31] L. Yamamoto and C. Tschudin. Genetic Evolution of Protocol Implementations and Configurations. In *IFIP/IEEE International workshop on Self-Managed Systems and Services (SelfMan 2005)*, 2005.