Constraint-based Optimizations of Executable UML Models

Andrey Karaulov, Alexander Strabykin

***Abstract* — this paper describes an ongoing research aimed on the creation and implementation of a set of transformations of executable UML (Unified Modeling Language) models that would improve the execution performance while preserving the behavior. Additional information useful for transformations can be extracted from the constraints embedded in the model. The article contains an example of informal description of a transformation, a scheme of modeling environment extension implementing such transformations, and a review of related tools.**

***Index Terms* — Constraint Programming, Computer Aided Software Engineering, Model Transformation, Object Constraint Language, Program Optimization, System Modeling, Unified Modeling Language.**

I. Introduction

In recent years Model Driven Development methodology (MDD) has drawn much attention among software development industry. According to the MDD software models become the only first class artifacts of the development process; and the whole system creation is seen as a sequence of model refinements starting with the very abstract system model and ending with the model that can be executed performing the functions of the system. This makes models fully reflect the complexity of the system being created and hence readdresses to models many problems previously related to source code in traditional programming languages like Java or C++. Moreover since models contain the description of system behavior precise enough to be executed, the efficiency of behavior specification becomes crucial and cannot be ignored for example for systems that have requirements on their performance. According to our experience complex models usually have a great potential for optimization, however such optimizations are time consuming and error-prone if done manually. Therefore there is a strong need for model optimizing tools.

This paper describes optimization of models in the Unified Modeling Language (UML) [1], since it is de facto an industry standard language for software modeling. However the future results of our study may be applicable to other modeling languages, e.g. domain specific, if they use similar formalisms for specifying system behavior. Behavioral features of the system can be expressed in UML by means of activity, state machine, and sequence diagrams. The latter rather describe a behavior resulted from the interaction of all the participants, but do not exactly specify the behavior of each single party and are used more for scenario definition and logging. Often the expressiveness of UML often is not enough to fully describe the semantic details of a system. In such situations constraints written in Object Constraint Language (OCL) [2] can be used in order to better reflect the semantics. Also OCL can be used to define queries, derived attributes etc, but constraints are usually specified by means of class invariants and pre- and post-conditions of operations.

The goal of our research is creation and implementation of the new methods of optimizing executable UML models. The distinguishing feature of selected approach to model optimization is extraction of additional information needed for model transformations by analyzing constraints embedded in the model. The model therefore is considered as for sure holding all the constraints it contains and the issues of constraint violations are left out of the scope of the paper. This is a common situation when constraints express hardware limitations or other conditions the system should only perform under.

II. Related Works

A typical model execution scenario consists of several stages including generation of code in a programming language, code compilation and execution. On each stage some optimizing transformations can be applied. For example, compilers that transform source code into executables can apply a variety of optimizations like function inlining, loop unrolling, etc [3]. All these transformations certainly can be applied when a code generated from a model is compiled. Despite most of such transformations can also be applied directly to UML models, since the language standard provides a way to specify all standard actions like loops, conditions etc; in this paper we deal with the higher level transformations that operate on UML models. This approach gives an optimizer the advantage of viewing the system as a whole, since the models fully describe the system behavior. Moreover on a UML level even semantic details expressed in constraints, which are usually unavailable on lower levels, can be analyzed while performing optimization. Optimizations applied to source code in traditional programming languages can have two usually conflicting goals: optimizing either memory usage or performance. Since UML standard lacks information related to memory allocation and distribution our research concentrates

Andrey Karaulov, Alexander Strabykin, Institute for System Programming (ISP) of the Russian Academy of Sciences, Russia, 109004, Moscow, Solzhenitsyna st., 25. Emails: {aka, alexs}@ispras.ru.

on performance optimizations while memory issues are left to be dealt with on lower levels.

Software refactoring is a process of changing the internal structure of an object-oriented program that preserves the observed behavior and is aimed on simplification of modifications and improvement of the readability and design [4]. Since UML is an object-oriented language a lot of refactoring transformations can be applied to UML models [6]; sometimes as a side effect of their application the performance can be improved, but this is not the goal of refactoring. UML models have some specific constructs like state machines and activities that have no direct analogues in traditional object-oriented programming languages. The optimizations of our interest have to preserve the behavior of the system being transformed like refactorings; however the main purpose is a more effective execution of the model. Moreover unlike compiler optimizations optimizing transformations being studied should be visible to user and may rely on user decisions.

UML state machines are based on finite automata formalism, which has been proposed for more than thirty years ago. There are a number of techniques created to minimize automata by means of removing equivalent states [9]. Application of these methods to UML state machines is complicated by the fact that transition equivalence cannot be proved without proving the equivalence of the actions being performed when transition is fired. However proving actions equivalence is not a trivial task if complicated semantics of UML actions is taken into account, since general problem of equivalence of two programs is algorithmically unsolvable.

Another group of related works we are aware of studies the problem of model transformations in general, since it is the key activity of MDD. UML models are usually considered as labeled multigraphs as in [6]. In this case transformation of UML models can be based on graph rewriting formalism. A model optimizer of our interest can be implemented as an extension of a model transformation framework. The most important features of such framework would be extensibility, capability of defining complex and parameterized transformations, and support of constraint analysis in models being transformed.

GReAT language described in [5] is a Graph Rewriting and Transformation language. It contains sublanguages for specifying patterns, transformation rules and control flow for advanced transformation. Extensibility of transformations is achieved by adding user-defined code in a procedural language to specify attribute mapping, which is performed after all graph related operations are done. This approach does not seem suitable for implementing heavy model analysis that must be done before optimizing transformations are executed.

Visual Modeling and Transformation System [7] is modeling environment that allows creating and transforming UML models with OCL constraints. It has a visual language to define complex transformations similar to UML activity diagrams. Matching mechanism uses metamodel approach i.e. is looking for a part of a model that can be identified as an instance of a metamodel pattern. Transformation constraints can be defined in OCL and are finally transformed to C# code. The whole system is also implemented in C#, which limits the platforms it is available for. There are no means provided for simplification of the analysis and according to [15] the performance of VMTS turned to be a number of orders of magnitude worse than that of Fujaba.

FUJABA (From UML to Java And Back Again) [8] was initially developed as UML modeling tool with code generation to Java. Later a visual language for model transformation based on graph rewriting was added. Transformations are defined by means of Story Diagrams, which can be seen as a mixture of UML activity and collaboration diagrams. For each transformation Java code implementing it is generated. The framework can be extended by means of plug-ins and the Fujaba itself can be integrated with Eclipse modeling environment [12]. Support of model constraint analysis is also absent.

The only tool that provides some model analyzing capabilities, which might be helpful when implementing optimizing transformations, is MagicDraw with ParaMagic plug-in [11]. However it uses System Modeling Language (SysML) [10], not UML. Since SysML models can contain parametric diagrams specifying relations e.g. equations among system variables. Mentioned plug-in allows in some cases resolving the equations, but cannot currently work with OCL constraints in the model.

Any of the tools mentioned does not provide all the features needed for convenient implementation of transformations optimizing model performance. It means that this issue should be studied further in detail before the final decision is made, but a new solution designed with support of optimizing transformation in mind is likely to be created by authors in case no other tool is found to meet all the requirements.

III. Causes for Model Defects

There can be many reasons why UML models may be optimized. The most common case is a mistake of a user creating the model. Modeling languages have higher level of abstraction compared to those of traditional programming languages and therefore operate with the concepts that are closer to the problem domain, not to the programming language domain. Even users that do not have a professional knowledge in programming, but have it in the problem domain can develop software systems with MDD. However such users are more likely to make mistakes in design and implementation and hence should be supplied with the tools detecting and preventing them.

Another typical situation emerges in case of component reuse. The reuse of components from other systems or component libraries can save a lot of time and effort, but at the same time can lead to ineffective or redundant models. This drawback may be overcome if there is an optimizer that can transform the components being reused in the system into more effective ones taking into account the semantics of the system being created. For example if we consider a component that implements the process of organization of computers into a tree-structure according to standardized

protocol. Tree structures are used for example to implement multicast functionality in computer networks. Protocols standards usually describe several roles that participants can play during the interactions according to the protocol. For a tree-structure organization there are three basic roles a computer modeled as a class with a state machine can play in the interaction: newcomer – a computer that would like to join a multicast tree; root – a computer that accepts join requests of the newcomers and ex-son – a computer that was a part of the tree, but needs to find a new parent because of tree reorganization [16]. A reusable component implementing this protocol should cover all the roles and cases described in the standard. However when being used in a particular system such implementation can be redundant if for example a model contains constraints that limit protocol implementation to certain roles only. For instance a certain system can make ex-son roles to be impossible. An optimizer in this case should be able analyze the constraints, detect, and remove statemachine elements needed only for implementation of redundant roles.

Behavioral features of UML models can even be generated automatically, for example on the basis of the formal specification or a complete set of test cases. Generated models also need to be checked for their performance, since often there is a lot of space for improvement.

IV. Tool Support

The transformations we study can be implemented as an extension to existing integrated modeling environment like for example Eclipse Modeling Framework [12]. The module can be divided into two parts Analyzer and Transformer as shown in fig. 1. The following workflow looks natural when working with the extension. When one wants to optimize the model he activates the corresponding command in the modeling environment. This can be done automatically when code generation is performed. Analyzer then checks the model and reasons about contained constraints.

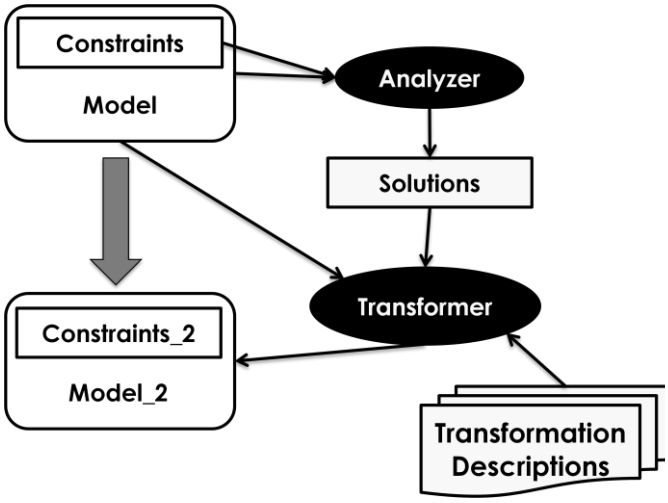

Fig.1. The Scheme of Optimizers Work

The purpose of analyzer's work is to provide additional information that might be helpful for optimization. The principles of its work are similar to those of partial evaluators [13]. In the beginning as a feasibility study we limit the types of supported constraints to algebraic expressions using operation parameters and class attributes e.g. *self.salary > 0*. Despite visible simplicity according to [14] such constraints are quite common in real systems. Analyzer iteratively propagates constraints over UML model actions. For example, if an input parameter *x : Integer* of an operation is constrained to be in range [0;c] and the first action of the operation declares a local variable *y*, which is initialized as *2x*, then the constraint can be propagated to that statement and limit the values of *y* to be within the interval [0;2c]. As a result of analyzer's work all actions in the model get a set of associated constraints. These results are available to Transformer module.

Transformer contains a set of transformation descriptions. For user convenience this set should be as flexible as possible, i.e. a user should be able to include and exclude transformations from that set. Moreover it is highly desirable that a user can create new transformations from scratch or by combining already existing transformations. A description of the transformation contains a pattern that is matched against user model and constraints defined on this pattern that must be satisfied. The patterns are defined on a metalevel that makes them independent on the model they are matched with. Therefore the matching process is not a search for a part of the model that is isomorphic as multigraph to the pattern being matched, but a search for a part of the model that is an instantiation of the metamodel pattern. In case all the pattern constraints are observed the transformation is added to the list of possible operations. After the matching for all active transformations is completed a user is presented with the list of possible operations for review and confirmation. In order to avoid undesired changes, e.g. those caused by a mistake in constraints, a user should be able to easily find out which constraints in the model made certain transformation possible. It is also important to keep the history of transformations for convenient use; this will allow reverting changes later if requested by user.

For a feasibility study the transformation that removes dead branches from the condition action can be considered. The pattern of this transformation matches all the choice pseudo states of state machines in the model. The constraint of this transformation should state that the estimated by Analyzer range for the expression on which the decision is based intersects with the only decision answer range. In this case all other answer transitions can be removed from the model as they are never fired. Transformation for decision nodes from activities specifications is defined similarly. The ways of formal specification of such transformations are currently under investigation by the authors.

V. Conclusion and Future Work

The spread and adoption of MDD by the industry of software development not only requires availability of the tools supporting MDD, but effective execution of the models being created with such tools, therefore transformations that can optimize performance of UML models are highly

demandable.

Current results of our research include the preliminary analysis of available tools supporting UML model transformation and the ways optimizing transformation can be formally described. The work will be continued in the following directions: new optimizing transformation will be created; model transformation tools study should be completed to decide the best implementation way; and the effectiveness of the transformations application will be studied on real industry projects.

REFERENCES

[1] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. http://www.omg.org/docs/formal/09-02-02.pdf

[2] Object Management Group. Object Constraint Language http://www.omg.org/docs/formal/06-05-01.pdf

[3] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[4] M. Fowler. Refactoring. Improving the Design of Existing Code. Addison-Wesley, 1999.

[5] D. Balasubramanian, A. Narayanan, C. vanBuskirk, G. Karsai. The Graph Rewriting and Transformation Language: GReAT. The proceedings of the Third International Workshop on Graph Based Tools, 2006.

[6] Mens, T., N. Van Eetvelde, S. Demeyer and D. Janssens, Formalizing refactorings with graph transformations, Int'l Journal on Software Tools for Technology Transfer 17 (2005), pp. 247–276.

[7] Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model transformation with a visual control flow language. International Journal of Computer Science 1(1) (2006) 45–53.

[8] FUJABA Homepage, http://wwwcs.upb.de/cs/fujaba/

[9] B.W. Watson, A taxonomy of finite automata minimization algorithms, Eindhoven University of Technology, The Netherlands. Computing Science Note 93/44 (1993).

[10] Object Management Group. OMG Systems Modeling Language. http://www.omg.org/docs/formal/08-11-02.pdf

[11] InterCAX SysML Parametric Solvers. http://www.intercax.com/sysml

[12] Eclipse Modeling Framework Project (EMF) http://www.eclipse.org/modeling/emf/

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial Evaluation and Program Generation. PrenticeHall, 1993.

[14] Wahler, M., Ackerman, L., Schneider, S.: Using IBM Constraint Patterns and Consistency. Analysis. IBM Developer Works, May 2008.

[15] A. Rensink, A. Dotor, C. Ermel, S. Jurack, O. Kniemeyer, J. de Lara, S. Maier, T. Staijen, and A. Zündorf. Ludo: A Case Study for Graph Transformation Tools. In A. Schürr, M. Nagl, and A. Zündorf, editors, Applications of Graph Transformation with Industrial Relevance, Proceedings of the Third International AGTIVE 2007 Symposium, volume 5088 of LNCS, Heidelberg, 2008.

[16] Dolejs, O. Tree Building Control Protocol - State Machine. Prague: CTU, Faculty of Electrical Engineering, Department of Control Engineering, 2001. K335/01/210. 9 s.