COMPUTER SCIENCE

ISSN 1407-7493

DATORZINĀTNE

2008-34

APPLIED COMPUTER SYSTEMS LIETIŠĶĀS DATORSISTĒMAS

TRANSFORMATIONS BETWEEN UML AND FIRST ORDER LOGIC

Janis Birgelis, Riga Technical University, Meza 1/3, Riga, LV 1048, Latvia, Mg. sc. ing, jbirgelis@e-apollo.lv

UML, program synthesis, first order logic, formalization

1. Introduction

Nowadays UML becomes one of the main software development standards. It provides a lot of diagrams [1]. Each diagram describes the modelling system in different aspects, but together they can give full map of the modelling system. However, too large number of diagrams can increase the time of modelling sufficiently. There have to be a balance between the number of diagrams and the time designers spend to create models. Moreover only a few diagrams are necessary in the specific problem area (for example, standard business application) to get full understanding about the modeling system.

Such approach conforms to the ideology of automatic program synthesis. Automatic program synthesis usually works in domain specific area because knowledge and rules are difficult to specify for universal domain area. In this field research has been carried out since 60-ies of the last century and there are a lot of promises. Scientists have been looking for how to synthesize a code from UML models during recent years.

This article will show how UML models can be translated into the language that is computable for computer.

2. History

The idea how UML together with automatic program synthesis meet principles of MDA was presented in publication "Generalization of MDA and Software Synthesis" [2], see figure 1.



Figure 1 Scheme of Automatic Program Synthesizer

Figure 1 shows that UML diagrams at first have to be checked by a checker for reusable objects. It is the step that allows a prover reach the aim faster. After that specification is made by translating UML diagrams into the first-order expressions. So the prover can use these expressions, domain specific knowledge basis and search algorithms to construct a proof. This proof uses can be viewed as a program [3]. So a translator takes the newly created proof and translates back to UML. A parser combines UML diagrams with reusable objects and makes a source code. As a result two things are created: an executable program and actual specification of the program in UML.

Figure 1 contains several text boxes one of which is gray. It shows the area which this publication covers using all research data.

3. Translation between UML and First - Order

As it is written in the internet encyclopedia WIKIPEDIA [4], first – order logic is very suitable for different kind of automated theorem proving tasks. Maybe there are some restrictions [4][5], but other logics are less developed. For example, *deductive tableau* that was developed by Manna and Waldinger [3] in 1992. It works on the first – order logic basis and provides fully automated solutions. So there is no need for human interaction in the process of program synthesis.

Taking into account all mentioned facts it has been decided to look for possibilities to make transformation from UML to first – order expression. At present there are thirteen different UML diagrams [1]. Each diagram describes the specific area of the modeling system, but often they overlap each other. And as it was mentioned earlier there is no need for all diagrams because some of them are used for modeling specific domain area.

With reference to Albert Zündorf who has developed the rigorous software development process (the second name: story driven modeling) only some UML diagrams to make full software project can be used [6]. He has developed the tool Fujaba that can make program's executable code from UML diagrams, too. Only these diagrams have to be very detailed and some of them are very specific. For example, the role of story board is like a collaboration diagram, but it is drawn like an activity diagram, where each activity is described by means of a state diagram. This diagram can be viewed as a graph with respect to the notation of UML,

too. Or we can say that UML is not only a modeling language but also a visual programming language. This ideology is different from my approach but any way it shows that there is no need for all thirteen diagrams for common projects included in the arsenal of UML.

But what is a minimal number of UML diagrams that can provide enough complex map of requirements of the project? Some scientists think that basic diagrams are: class diagrams, state diagrams and collaboration diagrams. These diagrams are commonly used in different types of modeling systems [6][7]. It follows thence that a basic situation can be modeled using these diagrams. Rest diagrams of UML show specific twists and turns that are very important for programmer but not so important for automatic program synthesizer. Because automatic program synthesizer can easy compute rest information from given diagrams while programmer can easy make some false interpretation that depends on his skills and knowledge.

To make transformation from UML to first - order logic expression basic ideas will be used that has been developed by scientists of Stockholm University [7]. At first a meta-model is given that shows the relationship between first - order logic and UML, see Figure 2.

There are two layers in the Figure 2 (Logic and UML). These layers are divided with a dotted line. In the figure, such UML artifacts as Classes, Attributes and Association can be translated as Predicate Symbols while other UML artifacts can be translated as Static or Event rules. The figure shows other important aspect, too. UML diagrams contain two sorts of information: static and dynamic information. The static information represents class structure, name of classes, attributes of classes, association between classes. While the dynamic information represents actions that are included in UML diagrams. For example, methods of classes, actions of collaboration diagrams and state diagrams.



Figure 2 Meta Model Showing the Relationship between First Order Logic and UML [7]

3.1. Example

To show how the static information can be translated from UML diagrams to first - order logic expressions a synchronization program will be modeled in UML and then translation to first - order logic will be made.

The synchronization program is the program that allows make the content of a destination directory equal to the content of a source directory. There are several uses that have to be taken into account. First, the destination directory can hold the file that is not represented in the source directory. Such file has to be deleted from the destination directory. Second, there can be a case when the version of the file in the destination directory is older than the version of the file in the source directory. Such file has to be overwritten by newer version from the source directory. Third, there can be a case when the destination directory already has the file that is represented in the source directory. Such file has not to be copied, because such operation is time consuming and unnecessary.

From a programmer's point of view such requirements are easy to model using two lists. List A represents files that are in the source directory, while List B represents files that are in the destination directory. The program has to search within all elements of List A in List B. If

there is no match, then List A and List B will remain as they are. But if the program finds the match between elements of List A and List B, then other check-up has to be performed. The program has to check the versions of the files. If the versions of files are equal then both elements have to be deleted from List A and List B else only the element from List B has to be deleted. A graphical view of the algorithm is shown in Figure 3.



Two lists are created as a result of the algorithm. List A will represent files that have to be copied or overwritten in the destination directory while List B will represent files that have to be deleted from the destination directory.

Actually Figure 3 shows the algorithm that has to make by the system itself. Three types of diagrams have to be written by a programmer: class diagram, state diagram and collaboration diagram.

In Figure 4, *File, Directory* and *System* are UML classes. There are two associations between these classes, too.



Figure 4 Class Diagram

3.2. Translation of the Static Information

Let R be the finite set of closed first - order formulae in a language L. L(R) is the restriction of L to R, i.e. L(R) is the set $\{p|p \in L, where p \text{ contains predicate symbol that is in a formula in R}\}$.

There a set C of class diagrams is given. *agg* is a predicate symbol of arity two in $L(R_C)$ while *lex* is a predicate symbol of arity one in $L(R_C)$.

• If r is a name of a class definition in C, then r is a predicate symbol of arity one in L(R_C).

Example: File(x), System(x) and Directory(x) - are a class definition showed in first - order logic.

• If t is a name of an association in C, then t is a predicate symbol of arity two in $L(R_C)$. Set R_C contains expression in form $\forall x \forall y(t(x,y) \rightarrow (r(x) \land s(y)))$, where r and s are names of class definitions in C.

Example: $\forall x \forall y (Asoc_l(x,y) \rightarrow (File(x) \land Directory(y)))$

• If t is a name of an attribute in C, then t is a predicate symbol of arity two in $L(R_C)$. Set R_C contains expression in form $\forall x \forall y(t(x,y) \rightarrow (r(x) \land lex(y)))$, where r is name of class definitions in C and *lex* is a reserved word.

Example: $\forall x \forall y (Name(x,y) \rightarrow (File(x) \land lex(y)))$

 $\forall x \,\forall y (Path(x,y) \rightarrow (File(x) \land lex(y)))$

 $\forall x \ \forall y (Source(x, y) \rightarrow (System(x) \land lex(y)))$

Of course, a class diagram can be more complex than it is shown in figure 4. In such case there are more translation rules, they are listed in Table 1.

Table 1 Translations for UML constructions

Case of translation	General form in Set R _C
Aggregation	$\forall x \forall y(t(x,y) \rightarrow (r(x) \land s(y) \land agg(x,y))), where t is name of$
	aggregation,
	r and s are name of classes in C, agg is a reserved word.
ISA	$\forall x (r(x) \rightarrow s(x))$, where r and s are names of classes in C.
Subclass (in case when two subclasses are disjoint)	$\forall x \neg (p(x) \land r(x))$, where p and r are names of classes in C
Subclass (in case when two subclasses exhaustive from main)	$\forall x (s(x) \rightarrow (p(x) \lor r(x)))$, where p, r and s are names of classes in C
Cardinality (minimum number of	$\forall y \exists x_1 \land \dots \land \exists x_{\min_r} ((s(y) \to (t(x_1, y) \land \dots \land t(x_{\min_r}, y) \land$
associations for the domain)	$\neg(x_1 = x_2) \land \dots \land \neg(x_1 = x_{\min_r}) \land$
	$\neg (x_2 = x_3) \land \dots \land \neg (x_2 = x_{\min_r}) \land \dots \land$
	$\neg(x_{\min_{r-1}} = x_{\min_r}))$
Cardinality (maximum number of associations for the domain)	$ \forall y \forall x_1 \dots \forall x_{\max_r} \forall x_{\max_r+1} ((t(x_1, y) \land \dots \land t(x_{\max_r}, y) \land t(x_{\max_r+1}, y)) \rightarrow $
	$((x_1 = x_2) \lor \lor (x_1 = x_{\max_r}) \lor (x_1 = x_{\max_r + 1}) \lor \lor$
	$(x_{\max_r} = x_{\max_r + 1})))$
Cardinality (minimum number of	$\forall y \exists x_1 \land \ldots \land \exists x_{\min_s} ((r(y) \to (t(x_1, y) \land \ldots \land t(x_{\min_s}, y) \land$
associations for the range)	$\neg(x_1 = x_2) \land \land \neg(x_1 = x_{\min_s}) \land$
	$\neg (x_2 = x_3) \land \land \neg (x_2 = x_{\min_s}) \land$
	^
	$\neg(x_{\min_{s-1}} = x_{\min_s}))$
Cardinality (maximum number of associations for the range)	
	$((x_1 = x_2) \lor \lor (x_1 = x_{\max_s}) \lor (x_1 = x_{\max_s + 1}) \lor \lor$
	$(x_{\max_s} = x_{\max_s + 1})))$

3.3. Translation of the Dynamic Information

ER is a set of event rules. An event rule in L has a structure $\langle P(z), C(z) \rangle$, where z is a vector of variables and P(z) describes the precondition of the event rule while C(z) the post condition. There is given a set of methods M in a UML specifications, where string *inv* is a reserved word (inv is a predicate symbol of arity one in $L(ER_M)$).

Translation of the Method:

If $g(y) = \langle pre(x), post(x) \rangle$ is a method in M, then the corresponding predicate symbols with the same arities are in $L(ER_M)$.

General definition of method in set ER_{M} : $\langle inv(g(y) \land pre(x), post(x) \land \neg inv(g(y)) \rangle_{k}$, where k is a class, inv(g(y)) means that the method g(y) has been invoked.

Example: Definition of methods of class File (figure 4) in first - order logic. $\{inv(copy_file(x,y) \& true, true \& \neg inv(copy_file(x,y), inv(delete_file(x) \& File_exist(x)=TRUE, true \& \neg inv(delete_file(x), methods)\}$ $inv(get_files(x) \& Left(x,2) = "C:", true \& \neg inv(get_files(x))$

Translation of the Collaboration Diagram:

There a collaboration diagram D as a pair $\langle B,M \rangle$ is given, where B is a set of classes and M is an ordered set with the structure $\langle a, ue(x), b, i \rangle$, where a and b are name of classes, ue(x) is an event and i is the integer. There is a reserved word *sent*, too.

If $\langle a, ue(x)_k, b, i \rangle \in M$, then the corresponding predicate symbols and constants with the same arities are in L(R_D).

The general form of translation rules is:

- $\{\forall x(inv(x) \rightarrow sent(x))\} \in \mathbb{R}_D$
- If $\langle y_1, ue_i(x), y_2, i \rangle, \langle z_1, ue_j(x), z_2, j \rangle \in M$, and $\times i \langle j$, then

 $\forall xy(sent(ue_i(x)) \rightarrow sent(ue_i(x))) \in \mathbb{R}_D$



Figure 5 Collaboration Diagram

Example: Set of methods in the collaboration diagram: <directory, get_directories(source), system, 1> <file, get_files(source), system, 2) <directory, get_directories(destination, system, 1>)

> Rules for Objects in the Collaboration diagram: $\forall x(inv(get_directories(source)) \rightarrow sent(get_directories(source))$ $\forall x(inv(get_files(source)) \rightarrow sent(get_files(source))$

Translation of a State Diagram:

There a set S of state diagrams is given <N,A,G>_k, where N is a set of nodes, A is a set of directed arcs and G is a set of quadruples <a, ue(x,y), guard(x,y), p(x,y)>, where a is an arc, ue(x,y) is a UML event, where x is an object of class k and y is a vector of objects and values, guard(x,y) is an open first order formula, p(x,y) is a set of formulas,

where each formula has one of the following forms:

 $insert(a(x,y_i)), delete(a(x,y_i)), invoke(umlev(z)).$

If <<t_i, t_j>, ue(x,y), guard(x,y), p(x,y)> \in G, then the predicate symbols and constants in t_i, t_j, ue(x,y), guard(x,y) and p(x,y) with the same arities are in L(ER_S) and L(R_S). There is a reserved word *state*, too.

The general form for a rules for object in a state:

 $R_{S} = \{\forall x(state(x,t_{i}) \rightarrow \neg(state(x,t_{j}))|t_{i},t_{j} \in \mathbb{N} \land i \neq j\} \cup \{\forall x \exists t(state(x,t)) \} \cup \{\forall x (state(x,t_{i}) \rightarrow k(x))|t_{i} \in \mathbb{N}\}, \text{ where } t_{i} \text{ is a state in a class } k.$

The general form for arcs:

If $\langle \langle t_i, t_j \rangle$, ue(x,y), guard(x,y), p(x,y) > is an arc in G, then:

 $\langle inv(ue(x,y) \land state(x,t_i) \land guard(x,y), state(x,t_j) \land \neg inv(g(y) \land \{\land a(x,y_i) \mid insert(a(x,y_i) \in p(c,y))\} \land \{\land \neg a(x,y_i) \mid delete(a(x,y_i) \in p(c,y))\} \land \{\land inv(umlev(z) \mid invoke(umlev(z) \in p(c,y)))\} \rangle_k$

In business applications it seems that a very small proportion of classes perhaps 5% at most are complex enough to warrant creation of a state diagram [1]. However, state diagrams are much more common in real time systems [8].

The structure of classes is too simple in the given example, that's way in which a state diagram is not represented.

4. Conclusions

Ideas that are presented in this work are very important for the future research in the area of automatic synthesis. Choosing UML as the main tool for modeling tasks [6] there have to be a possibility to make transformation from UML to the language that is computable for computer. The first – order logic has been chosen as the most suitable language for such task because it is supported by lots of tools [3][7][9]. Using ideas about transformations between UML and first order logic and ideas about story driven modeling it is possible to describe how automatic program synthesizer will work with modeling program's specification [2], figure 1.

Unfortunately at the moment there is no possibility to make transformation for all UML diagrams into first – order logic, but the article shows the minimal number of UML diagrams that can be translated and can be used for modeling almost all standard business applications. In fact, other drawback is that the article doesn't show ways how constraints of OCL can be translated into UML, too. There is some previous work in this area [10], but there is still a lot of work to be done.

The next step in the future research will be to build an algorithm in Prolog. Prolog is a programming language that is very close to first – order expressions and there are a lot of algorithms already made in the area of artificial intelligence. Moreover, the work in the area of transformations shall be continued as well.

This work has been partly supported by the European Social Fund within the National Programme "Support for the carrying out doctoral study program's and post-doctoral researches" project "Support for the development of doctoral studies at Riga Technical University".

References

1. Internet: "http://www.agilemodeling.com/artifacts" //27.09.2007;

2. J. Birgelis, J. Osis "Generalization of MDA and Software Synthesis" //Scientific Proceedings of Riga Technical University. Computer Science. Applied Computer Systems (22. vol.). Riga: RTU, 2005, p. 217 – 228;

3. Z. Manna, R. Waldinger, *"Fundamentals of Deductive Program Synthesis"* //IEEE Transactions on Software Engineering, 1992;

4. Internet: "http://en.wikipedia.org/wiki/First-order_logic" //28.09.2007;

5. A. Ayari, D. Basin, "A Higher – Order Interpretation of Deductive Tableu" //Albert Ludwings Universität, Freiburg, 2001;

6. Albert Zündorf, "Rigorous Object Oriented Software Development" //draft version 0.3, 05.03.2007;
7. Benedict Amon, Love Ekenberg, Paul Johannesson, Marcelo Munguanaze, Upendo Njabili, Rika Manka Tesha, "From first-order logic to automated word generation for Lyee" //Knowledge-Based Systems (16. vol.), 2003, p. 413 – 429;

8. Bruce Powel Douglass, "Doing Hard Time: Developing Real – Time Systems with UML, Objects, Frameworks and Patterns" //Addision-Wesley Longman Publishing Co., 1999

9. Ivan Bratko, "Prolog Programming for Artificial Intelligence" //Addision-Wesley Longman Publishing Co., 2001

10. Bernhard Beckert, Uwe Kellere, Peter H. Schmitt, "Translating the Object Constraint Language into First-order Predicate Logic" //Karlsruhe Universität;

Birģelis J. Transformācijas starp UML un pirmās kārtas loģiku

Ar katru dienu MDA pamatnostādnes kļūst aizvien populārākas programmatūras inženierijā. Lai arī viena no galvenajām idejām, ideja, ka programmatūru jāvar automātiski pārvērst no projektējuma modeļiem līdz izejas kodam un otrādi, vēl nav pilnībā realizēta, tomēr eksistē daudz iestrādes, kā to var realizēt daļēji. Piem., viena no populārākajām modelēšanas valodām, kas daļēji atbalsta šādu ideju ir UML. Šo valodu plaši lieto programmatūras projektu izstrādē. Tā atbalsta daudz diagrammu, kas ļauj aprakstīt modelējamās programmas prasības no dažādiem skata punktiem. Bet lai apgalvotu, ka UML atbalstītu MDA pamatnostādnes pilnībā, nepieciešams izstrādāt iespēju veikt transformāciju starp UML diagrammām un izpildes kodu pilnībā. Eksistējošās transformācijas galvenokārt nodrošina automātisku programmatūras struktūras izveidi, tomēr tai būtu jāvar izstrādātos modeļus pilnībā transformēt izejas kodā. Šāda transformācijas attiecināmā uz mākslīgā intelekta jomu, kur, kā mēs zinām, jautājumu vēl joprojām ir vairāk nekā atbilžu. Pirmkārt UML diagrammas būtu jāvar formalizēt, lai dators spētu tās saprast viennozīmīgi. Pēc tam būtu jāizstrādā tāds algoritms, kas spētu veikt šāda tipa transformāciju. Šajā raksta ietvertas idejas, kā dažas no UML diagrammām, tādas kā klases diagramma, stāvokļa diagramma un sadarbības diagrammas, var tika transformētas pirmās – kārtas izteiksmēs. Savukārt iegūtās pirmās – kārtas loģikas izteiksmēs varētu tikt pielietotas dažāda veida programmatūras sintezatoros, lai iegūtu pilnībā izpildāmu kodu.

Birgelis J. Transformations between UML and first order logic

The basic principles of MDA are becoming increasingly popular in software engineering. Although one of the main concepts stipulating that software must be automatically convertible from project models into an output code and vice versa is not completely realized yet, there exist many solutions allowing to implement the concept partially. For example, one of the most popular modeling languages partly embracing such concept is UML. This language is widely used in elaboration of software projects. It supports many diagrams that allow to describe the requirements of modeled software from various points of view. But in order to assert that UML fully complies with the basic principles of MDA, it is necessary to elaborate the possibility to fully transform UML diagrams into an execution code. The currently existing transformations mainly ensure automatic creation of software structure, but they need to be able to completely transform the elaborated models into an output code. Such transformations apply to design of artificial intelligence, where, as we know, there are still more questions than answers. First, it must be possible to formalize UML diagrams in such a manner as to allow the computer to understand them unequivocally. Then, an algorithm that would allow to perform such transformation must be developed. The present article elaborates the idea that many UML diagrams, such as class diagrams, state diagrams and interaction diagrams can be transformed into first-order logic expressions. The obtained first-

order logic expressions, in their turn, could be implemented in various software synthesizers in order to obtain a fully executable code.

Биргелис Я. Преобразования между UML и логики первого разряда

С каждым днем основные принципы модельно ориентированной разработки программного обеспечения приобретают все большую популярность в программной инженерии. В настоящий момент одна из главных концепций, гласящая, что необходимо обеспечить возможность автоматического преобразования программного обеспечения из проектной модели в исходный код и наоборот, ещё не до конца реализован. Несмотря на это, существует большое количество разработок, позволяющих частично реализовать такой подход. Например, одним из наиболее популярных языков моделирования, который частично соответствует этой концепции, является унифицированный язык моделирования UML. Этот язык широко применяется в разработке проектов программного обеспечения. Он поддерживает множество диаграмм, позволяющих описать требования моделируемой программы с различных точек зрения. Однако для того, чтобы утверждать, что UML в полном объеме поддерживает основные положения модельно ориентированной архитектуры, необходимо разработать возможность преобразовать диаграммы UML в код исполнения. Преобразования, существующие на данный момент, главным образом обеспечивают автоматическое создание структуры программного обеспечения, однако они должны быть способны полностью преобразовывать разработанные модели в исходный код. Подобные преобразования касаются области создания искусственного интеллекта, где, как известно, вопросов всё ещё больше, чем ответов. Прежде всего, необходимо формализовать диаграммы UML таким образом, чтобы компьютер был способен их понять однозначно. Вслед за этим следует разработать алгоритм, который позволил бы произвести преобразование подобного рода. Полученные выражения логики первого разряда, в свою очередь, можно использовать в разнообразных синтезаторах программного обеспечения для получения полностью выполняемого кода.