

Assurance of System Consistency During Independent Creation of UML Diagrams

Lukasz Fryz, Leszek Kotulski
Department of Automatics
AGH University of Science and Technology
Poland
email:kotulski@agh.edu.pl

December 15, 2006

1 Abstract

Graph transformations are a very intuitive formalism used in visualization, modelling of distributed systems or in support its allocation. The formal description the modelling process (both UML diagram and the trace of designer decisions) complicate its structure; we should be able support both parallel activity of local graph transformation systems and their cooperation for the assurance of the systems consistency. In the paper, the concept of conjugated graphs is introduced for this purpose. Next, the implementation of this concept with the help of aedNLC graph grammar is presented.

2 Introduction

Unified Modelling Language [3] is a tool that supports modelling of large computer systems. Gathering of user requirements is made by considering their different points of view represented by a set of Use Case Diagrams, that often are concurrently generated by different consultants. Unfortunately, UML does not support neither the process of verifying the correctness of such distributed information consolidation nor the process of transforming it to another form of UML diagrams (interaction, class, deployment, ...). This is left to methodologies of system construction (like RUP [1], or ICONIX [7]) but is not formalized inside UML diagrams.

Let us note that UML diagrams are a notation that can be formally represented as a graph in a form specified by XMI (see Diagram Interchange Standard [2]). It is obvious that these graphs should be generated and modified by different graph transformation systems, because the rules of these transformations (graph grammar productions) differ themselves.

On the other hand the basic property of a good methodology is to ensure that the final functionality (represented by Class Diagrams) should arise from user requirements (represented by Use Case Diagrams). It means that some types of diagrams are strongly influenced by another types, due to the methodology

used for modelling. The remembering these relation is the same notation as diagram allows for example manage the risk of failure some part of the system [6].

From the formal point of view we have to manage a graph transformation system that not only consists of independently working graph transformation systems, but also maintains some relation among the elements of distributed graphs generated in this way. In section 3, such a system of conjugated graphs is introduced. Next, in section 4, it is shown how to implement conjugated graphs using an aedNLC graph transformation system [4, 5].

3 From UML to Conjugated Graphs

UML is a recent approach supporting the effort of designing object-oriented modelling systems, where all main issues of system analysis and design are taken into account. The newest UML version offers 13 types of diagrammatic notations that express different aspects of the modelled system's behavior. Each type of diagram describes the complete user knowledge on the given aspect at a given time. Let us consider a few of them:

- use case diagrams represent user requirements,
- class diagrams represent the structure of software,
- deployment diagrams represent the allocation of software components to hardware computing nodes,
- timing diagrams (a new artifact of 2.0 version) represent the timing properties of the system execution.

Within a given type of diagrams the notation formally describes the associations among its elements. The term formally can here be understood as meaning that this diagrammatic notation is equivalent to the XMI graph notation (due to Diagram Interchange standard [2]).

The influence of information contained within one type of diagrams on the development of diagrams of different types is left to the methodology of the process modelling (and thus outside the scope of the UML standard). Rational Unified Process [1] methodology seems to be the most complete of them and it tries to address influences among all types of UML diagrams. Another one, called ICONIX [7], presents practical aspect of system modelling from defining user requirements to code generation. This methodology describes the influence of requirements described by Use Case diagrams on Sequence Diagrams (via Robustness Diagrams), and the influence of Sequence Diagrams on Class Diagrams. Let us note that this approach is iterative and incremental, i.e. Class Diagrams are refined incrementally during the successive iterations through combined use of Use Case, Robustness, and Sequence Diagrams analysis. The incorporation of other types of diagrams to the modelling process in ICONIX is left to the designers team discretion.

The common idea of the presented modelling methodologies is using of information maintained in one type of diagrams to create new parts of another type of diagram (usually starting from requirements to the code description and its deployment). We must however note that the designer decision is correlated

only with the final effect of modelling (new parts of model has been created), but there is not any trace, in UML diagrams, the reasons of its creation; so we can assume that these reasons have been forgotten in a short period of time.

Lack of associations of the type "class C fulfills requirement R" was at the bottom of that when we would like to evaluate risk [6], we should extend UML notation and introduce the concept of a graph repository. The graph repository maintains the full description of a set of UML diagrams (in a form equivalent to XMI diagram notation) and the vertical relations among the elements belonging to different types of diagrams, like mentioned above, gathered during the modelling process.

The existence of vertical relations seems to be very useful during the software maintenance phase, so it seems reasonable to consider the graph repository as one of the concepts supporting creation of dependable system.

Let us consider some properties of a formal graph transformation system that could support such a solution. Among the demands from such a formalism the most challenging seem to be the following:

- it offers the proper descriptive power (sufficient for maintaining UML graphs),
- it assures the high efficiency of graph transformation (it should maintain and modify thousands of nodes and edges),
- it supports the distribution of information.

For better understanding of these demands let us consider an example. We assume that the first phase of modelling has already been finished and the user requirements are gathered and described as a Use Case Diagram. In the next phase a few developer teams (for example two) concurrently create the system description at the Class Diagram level.

In the above case we have defined the graphs:

- the first one, G_1 , equivalent to the Use Case Diagram
- the next two, G_2 and G_3 , equivalent to the appropriate parts of the Class Diagrams.

It is easy to notice that these graphs are conjugated:

- the elements of G_1 (requirements) are connected with the elements of G_2 or G_3 (classes) when they support fulfillment of this requirement (via a vertical relation)
- elements of G_2 and G_3 are connected when some classes created by one team are used by the other.

These connections are easy to maintain within a centralized repository, but some problems appear when we would like to distribute it in a way strictly corresponding to the workplaces of the development team. We have yet another reason of separating the mentioned graph— G_1 describes a different type of diagrams than G_2 and so G_3 its modification should be controlled by a different graph transformation system (in this case at the level of defined productions) than the others.

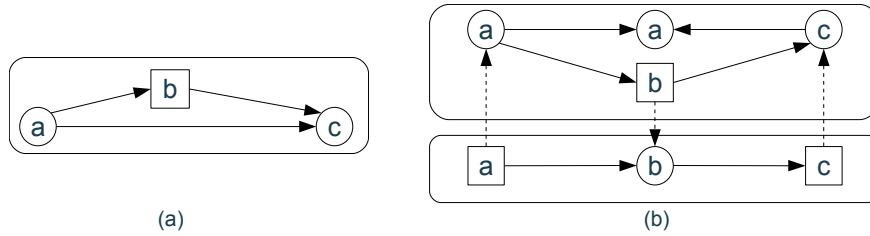
The outlined problem justifies the introduction of the conjugated graphs mechanism presented in the following section.

4 Conjugated Graphs

In order to define a system of conjugated graphs we have to extend the notion of a graph by introducing so called *L-R graphs* (from *Local-Remote*). First, we require that the nodes of a graph be divided into two disjoint subsets: the local nodes (LNs) and the remote node representatives (RNRs). The intuition is that RNRs visualize nodes that are local nodes of some other graph, and are incorporated into a graph only to allow introduction of edges that “cross boundaries” of graphs. This intuition is also a justification of the second condition we impose on L-R graphs that edges between RNRs are prohibited. This can be explained by pointing that such an edge should be represented by a local-remote or local-local edge in another graph.

Figure 1a illustrates the graphical convention that will be used for depicting L-R graphs: local nodes are drawn with circles and remote nodes with squares.

Figure 1: An example L-R graph (a) and a system of two conjugated graphs (b)



Before analyzing the definition of a system of conjugated graphs we need to introduce some notation. For a given graph G , let $E(G)$ denote the set of its edges, $V(G)$ the set of its nodes. Furthermore, let $R(G) \subset V(G)$ be the subset consisting of remote node representatives, and $L(G) = V(G) \setminus R(G)$ the subset of local nodes.

A *system of conjugated graphs* is an indexed family of graphs $\{G_i\}_{i=1,\dots,n}$ together with a family of functions T_i . A *system of conjugated graphs* is an indexed family of graphs $\{G_i\}_{i=1,\dots,n}$ together with a family of functions

$$T_i: R(G_i) \rightarrow \bigcup_{k=1}^n \{k\} \times L(G_k)$$

satisfying the condition

$$\forall x \in R(G_i): T_i(x) = (j, y) \Rightarrow j \neq i$$

For each RNR x of the i -th graph, the function T_i identifies the graph from which the node referred to by x is taken, and the specific node of this graph that x refers to.

As a convenience, we will denote the first projection of T_i as TG_i (*target graph*) and the second one as TN_i (*target node*).

We say that a system of conjugated graphs is *consistent* if the following conditions hold:

- for each $x \in R(G_i)$ the labels of x and $TN_i(x)$ agree,
- for each edge $(x, \mu, y) \in E(G_i)$, where $x \in L(G_i)$, $y \in R(G_i)$, $j = TG_i(x)$, $L(G_j) \ni v = TN_i(y)$ and μ is an edge label, there exists an edge $(z, \mu, v) \in E(G_j)$, where $z \in R(G_j)$ and $TN_j(z) = x$,
- for each edge $(y, \mu, x) \in E(G_i)$, where $x \in L(G_i)$, $y \in R(G_i)$, $j = TG_i(x)$, $L(G_j) \ni v = TN_i(y)$ and μ is an edge label, there exists an edge $(v, \mu, z) \in E(G_j)$, where $z \in R(G_j)$ and $TN_j(z) = x$.

The last two conditions imply that for each edge connecting a local node and a remote node there is a “mirror” edge in another graph. These are called a pair of *corresponding edges*.

Figure 1b demonstrates a system of conjugated graphs. The mappings T_i are represented with dashed arrows.

5 Modifying conjugated graphs with an aedNLC graph transformation system

We will now describe how modification of a sample conjugated graphs system can be managed using an aedNLC graph transformation system (GTS).

An aedNLC graph transformation system consist of the:

- an environment managing messages (called requests) among distributed entities
- a derivation control diagram (associated with each entity), that services incoming requests, generates new ones, and modifies a local graph structure (by applying graph grammar productions)
- an attributed edge-labelled directed NLC graph grammar that formalizes the rules of graph transformation.

A derivation control diagram is a sextuple $S = (N, I, F, T, \Pi, Wait)$, where:

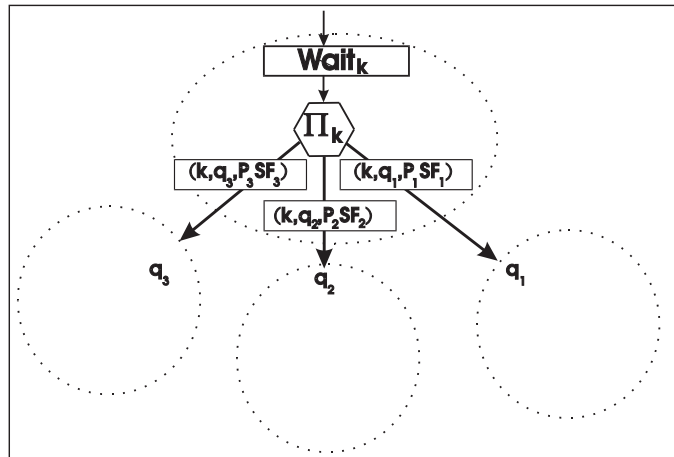
- N is the set of control points,
- $I \subset N$ and $F \subset N$ are, respectively, the set of starting control points and the set of final control points,
- T is the set of transitions of the form (k, q, P, SF) , where:
 - k, q are control points,
 - P is either a production, or the symbol \emptyset when no production is associated with this transition,
 - SF is a semantic action,
- $\Pi = \{\Pi_k\}$ is the set of selectors,
- $Wait = \{Wait_k\}$ is the set of synchronizing functions.

The synchronizing function $Wait_k$ suspends evaluation of the selector Π_k until the condition it specifies is fulfilled. Selector Π_k points: a production, a semantics action and a destination control point associated with the chosen transition. Initially, an activity is associated with each starting point; when a transition is fired the activity is moved to the destination control point. If the source control point is a starting one the activity is also replicated; if the destination node is a final one, the activity is cancelled. We assume that for any k the successful evaluation of the synchronizing function $Wait_k$ and the selector Π_k is performed in the critical section over the graph G and the set of requests ω . These assumptions imply not only the correctness of the sequential evaluation of the conditions defined inside control point, but also exclude busy form of waiting during evaluation of the awaiting synchronizing conditions. Like in the Ada programming language, in the worst case all the synchronizing conditions are evaluated after any modification of G or ω . If no transition is fired to modify G or ω , the evaluation is suspended until some new request r appears ($(\omega' = \{r\} \cup \omega)$). The semantic function SF (associated with the transition):

- enriches external request set (requesting some actions either of the designing system or the user),
- removes the request, that is serviced, from ω ,
- evaluates the parameters of the right-hand side graph of the production P . The production P is then applied to the current graph G and a new graph H replaces G .

More intuitively, DCD can be interpreted as a graph connecting control points (see fig. 2) inside of which both the synchronizing function and the selector, choosing one of the transitions from one control point to another one (drawn as an edge), are sequentially evaluated. During such a transition the production P_i is applied and the semantic action SF_i is executed.

Figure 2: Illustration of DCD



To subsume, the Derivation Control Environment (DCE) is responsible for controlling both the local graphs modification and the whole distributed system cooperation.

Our example will concern introduction of a relation among some requirement and classes that are created for its fulfillment. We assume that:

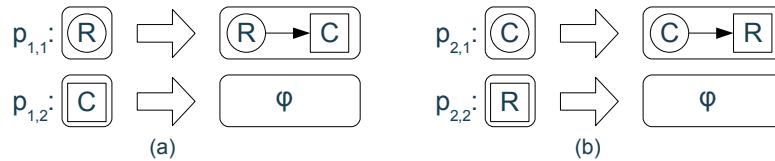
- The node representing a requirement, labelled by “R”, will appear in the graph G_1 .
- The nodes representing classes, labeled by “C”, will appear in the graphs G_2 or G_3 .

Of course, this implies that RNRs with the label “[R]” will appear in G_2 and G_3 , and RNRs labelled “[C]” will appear in G_1

The mapping T_i will be represented by storing the value of $T_i(x)$ as an attribute of each RNR x of the graph G_i .

Describing the graph grammars modifying graphs G_1 , G_2 and G_3 we will concentrate only on productions that are not local, i.e. that contain at least one RNR. So, the graph grammar used for modification of G_1 contains productions $p_{1,1}$ (creating connection) and $p_{1,2}$ (deleting connection) shown in Figure 3a. The figure shows only the left- and right-hand sides of the productions, symbol φ means an empty node (that is removed during garbage collection). The edNLC graph grammar productions also contain the embedding transformation. The embedding transformation for $p_{1,1}$ is assumed to preserve all the edges incident to the rewritten “R” node. The embedding transformation for $p_{1,2}$ is irrelevant.

Figure 3: Productions of the grammar controlling modification of G_1 (a) and G_2 (b)



Analogically, for modification of G_2 we will need the productions shown in Figure 3b. Productions for modification of G_3 are exactly the same. As before, the embedding transformation for $p_{2,1}$ preserves all the edges incident to the “C” node.

Establishing the representation of a relation “requirement X is supported by class Y” calls for strict cooperation between the GTS_1 and GTS_2 (or GTS_3). As already mentioned, the system designer analyzes the requirements and create some classes (that fulfill these requirements), so when the class Y is created, then GTS_2 sends the request *find_requirement_representative* to GTS_1 . GTS_1 replies with the message *found_requirement_id*(Rid); this message is serviced by the DCD of GTS_2 , and the production $p_{2,1}$ is applied to the node representing class Y (labeled “C”). The pair $(1, Rid)$ is stored in the attribute representing T_2 of the added node [R]. Next, the request *add_class_representative*($Rid, 2, Cid$) is sent to GTS_1 . This request causes the production $p_{1,1}$ to be applied to

the node indexed by *Rid*, and inside the added node (labeled by [C]) we store the attribute (2, *Cid*). The successful application of both productions creates a system of conjugated graphs in the form defined in section 2; to ensure that both of them have been executed we can add to the proposed schema some additional message assuring the transactionality of this action (e.g. in a form presented in [4]). Let us note that when any GTS decides to remove a connection with a RNR and remove it (see productions $p_{1,2}$ or $p_{2,2}$) then we assume that applying the appropriate production entails the semantic action of sending the request *removeRNR*(*Z*) to the GTS pointed by the TG parameter of the removed RNR and with the value of the TN parameter passed as *Z*. The evaluation of this request causes the removal of the counterpart RNR in the second GTS.

6 Conclusion

The concept of conjugated graphs allows us to describe how complicated graph structures, such as a graph repository maintaining both the UML notation and the trace of the most important decisions undertaken in the modelling process, are to be modified. Moreover, it has been shown that this concept can be implemented in the distributed environment using an aedNLC graph grammar.

The proposed solution places the entire responsibility for designing the system of cooperating productions (executed during Derivation Diagram's evaluation) on the designer, who has to personally check that the proposed set of productions does not violate the consistency rules introduced in section 2. Automatic verification that a given set of grammars ensures generation of consistent conjugated graphs seem to be the most urgent direction for further research.

References

- [1] *IBM Rational Unified Process*. <http://www-306.ibm.com/software/rational/>.
- [2] Unified modelling language (2005): Diagram interchange, version 2.0.
- [3] Unified modelling language (uml), version 2.0.
- [4] L. Kotulski. *On the support distributed software generation with help of graph grammars - in polish*. Qualifying for professorship dissertations. Jagiellonian University Press, 2000. ISBN 83-233-1391-1.
- [5] L. Kotulski. Parallel allocation of the distributed software using node label controlled graph grammars. *Institute of Computer Science Jagiellonian University*, (preprint 7), 2003.
- [6] L. Kotulski and D. Dymek. Evaluation of risk attributes driven by periodically changing system functionality. In *Transactions on Engineering, Computing and Technology*, volume 16, pages 315–320. Enformatica, 2006.
- [7] D. Rosenberg and K. Scott. *Use Case Driven Object Modelling with UML*. Addison-Wesley, 1999.