A Rule-Based, Integrated Modelling Approach for Object-Oriented Systems

Benjamin Braatz [1]

Insitut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany

Abstract

In this paper an integrated modelling approach for object-oriented systems is proposed. The integrated language consists of three layers. On the first layer UML class diagrams are used to define the structure of the modelled systems and OCL expressions specify queries, which do not modify the object configuration. On the second layer transformation rules model local state modifications of the system. On the third layer Nassi-Shneiderman diagrams describe complex control flows built over the rules and queries on the lower layers. The proposed integrated language is evaluated by a running example on modelling doubly linked lists and the mergesort algorithm.

Key words: Rule-Based Transformation, Nassi-Shneiderman Diagrams, UML, Integration

1 Introduction and Related Work

The Unified Modeling Language (UML) [10] has become the pre-dominant modelling language in object-oriented software development. The behavioural techniques provided by the UML, however, do not contain a method for the declarative, rule-based specification of modifications on object structures. Moreover, the interconnection between different behavioural techniques is treated rather superficially in the UML specification, because the UML tries to permit as many usage and interconnection scenarios as possible.

In this paper an integrated modelling approach is proposed, which tries to eliminate these deficiencies by giving a layered collection of specification techniques with a clear separation of concerns and well-defined interconnections. The proposed integrated language is organised in three constitutive layers.

On the first layer UML class diagrams are used to specify the structure of the system. Expressions of the Object Constraint Language (OCL) [9]

[1] bbraatz@cs.tu-berlin.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

specify the behaviour of query operations, which do not change the object configuration of the system. This layer is introduced in Sect. 2.

On the second layer local state changes are modelled using a variant of single-pushout graph transformation rules [1] tailored to the UML on this layer. This layer is described in Sect. 3. Object-Based Graph Grammars [11] and Object-Oriented Graph Grammars [7,6] are other approaches using graph transformation rules to specify the behaviour of object-oriented systems. They are, however, designed as self-contained specification techniques without relation to the UML.

On the third layer the assembly of the queries and rules from the previous layers into complex control flows is achieved by structured flowcharts [8], which are also known as Nassi-Shneiderman diagrams. The third layer is explored in Sect. 4. Structured flowcharts are favoured over e. g. UML activity diagrams in this paper, because we believe that they provide a viable alternative for the visual modelling of control flows. On the one hand, they are close to the structure of the constructs in most contemporary, imperative programming languages, which could lead to a broader acceptance among programmers and software designers. On the other hand, the separation of concerns realised by the layered, integrated language facilitates the compactness and comprehensibility of the flowcharts making them easier to grasp than the source code of a programming language. Different approaches to control the application of graph transformation systems are proposed in the literature. Transformation Units [4] are one of the most prominent examples. They provide an abstract framework for the definition of control structures over transformation rules, where the structured flowcharts of this paper could probably be integrated into that framework as a sophisticated specification language for control structures.

These layers are integrated in the sense that the interconnections between the different layers are precisely defined: The queries on Layer 1 are only allowed to call other queries on Layer 1. The transformation rules on Layer 2 may use OCL expressions on Layer 1 in their attribute specifications. The structured flowcharts on Layer 3 can use OCL experessions on Layer 1 and call arbitrary other operations, which may be specified by transformation rules on Layer 2 or other flowcharts on Layer 3. The integrated language is intended to be constructive in the sense that the behaviour of a system can (and should be) completely described using the sublanguages on the appropriate layers.

The Fujaba Tool Suite [3] uses Story Diagrams [2], which are a combination of activity and collaboration diagrams, to specify transformations on object-oriented systems. The Fujaba approach is very similar to the one proposed in this paper. In contrast to Fujaba, which employs Java source code for the specification of low-level expressions, we use OCL expressions, which are on the one hand already integrated into the UML family of languages, on the other hand they aid in keeping the approach platform independent. Another difference is the strict separation of concerns with transformation rules and flowcharts specified in self-contained diagrams, respectively, where

Fujaba uses the integrated Story Diagrams. The separation of concerns eases the reuse of transformation rules in different flowcharts and of flowcharts in other flowcharts. The choice of visualisation techniques for rules and control flow is the last main difference. Fujaba uses collaboration diagrams to visualise rules, while in this paper a seperate visualisation of left- and right-hand-side is chosen, because the collaboration visualisation cannot adequately capture the change of attribute values and the specification of negative application conditions. In Fujaba UML activity diagrams are used for the control flow, where our reasons for choosing structured flowcharts have already been given above.

The semantics of the UML languages is – sometimes deliberately – left ambiguous and only given in natural language. In order to allow features like precise reasoning and code generation, a more restricted and formal approach has to be considered. Therefore, the integrated modelling language proposed in this paper is designed to allow the definition of a precise semantics.

It is possible to use graph transformation systems also as the semantic domain of object-oriented modelling techniques. This is done in [13,5], where UML class, object, state, collaboration, and use case diagrams are translated into graph transformation systems. This approach is complementary to the one in this paper, where graph transformations are used as an additional modelling technique on the syntactical level.

2 UML Foundations

In this paper we use UML class diagrams [10] to specify the class structure of the modelled system. Additionally, the behaviour of query operations, which do not change the object configuration of the system, is specified by OCL constraints [9], which are guaranteed to have no side effects on the system state. Note, that we do not employ OCL invariants and pre-post-conditions in this paper. Those constraints will be considered in future work on verification, shortly discussed in Sect. 5, where they will play the role of descriptive specifications against which the constructive models defined in this paper should be checked.

The two subsequent layers will be designed to closely match and reuse the concepts of class diagrams and OCL constraints. For example, the self and return parameters of OCL have counterparts in both rules and flowcharts. Additionally, rules and flowcharts also require OCL expressions in various locations.

As a running example we specify doubly linked lists, whose elements contain an integer as key and a string as data content. The class diagram of the example is depicted in Fig. 1. The abstract class ListItem is used as an abstraction of the common characteristics of lists and the elements in the lists. The objects of the List class serve as sentinels for the list, such that the next item is the first element of the list and the previous is the last one. This approach

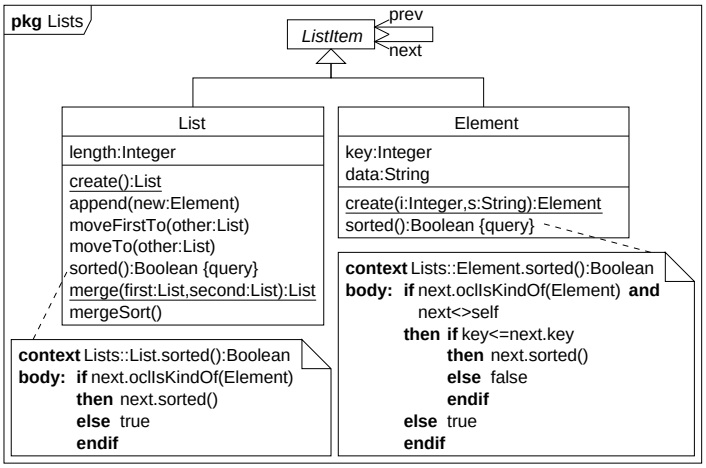


Fig. 1. Class diagram of the example

ensures that we do not have to deal with undefined pointers. Moreover, it allows to check if a list is non-empty and if an element has another successor by calls to the builtin OCL property oclIsKindOf(Element) on the next link. Note, that this class diagram would also allow object structures with multiple instances of List in a list, but the operations – namely the structure modifications specified by the rules in the next section – do not allow the creation of such senseless structures.

The underlined operations List.create, Element.create, and List.merge are static operations, which are called in the context of the class instead of a particular object of the class. The operations List.sorted and Element.sorted are annotated with a query property string expressing that they are not allowed to change the configuration of objects and attributes in the system.

The sorted queries are specified by the OCL constraints in Fig. 1. The List.sorted query returns true for an empty list and calls the Element.sorted query on its first element otherwise. The Element.sorted query returns true if the element is the last in the list, i. e. the next link points to the containing instance of List and not to another instance of Element, or the element is not contained in a list at all, i. e. the next link points to the element itself. If there is another element in the list, the keys are compared and false is returned if they are not in the correct order. Otherwise the query is called recursively on the next element.

3 Transformation Rules

On the second layer of the integrated modelling approach we will use transformation rules to describe local state changes in object configurations. These rules are a variant of single-pushout graph transformation rules [1].

A rule is given by a left- and a right-hand-side consisting of instance specifications. The left-hand-side (LHS) is connected to the right-hand-side (RHS) by a partial, injective mapping. Since the application of a rule should be de-

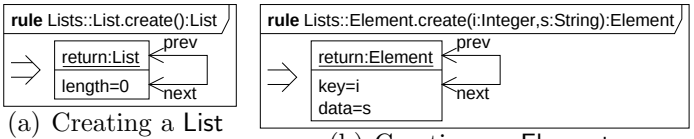
termined by the parameters given to the operation, we require the LHS to be uniquely navigable from the instance specifications representing the parameters, whose type is a class. A special parameter self is available in non-static operations to denote the object on which the operation is called. If the operation has a return parameter, the special parameter return has to be used on the RHS to designate the output of the rule. For attributes the instance specifications on the LHS may declare OCL constraints, which have to hold for the rule to be applicable. The instance specifications on the RHS may then specify the new values of attributes by OCL constraints, which may similarly to post-conditions use the @pre operator to access the attribute values before the rule application. The OCL attribute constraints may also use any data type parameters given to the operation.

Given a match of the LHS in an object configuration, the application of the rule can be constructed by removing the parts of the LHS not mapped to the RHS and adding the parts of the RHS, which do not have a preimage in the LHS. Since we assume an execution environment with garbage collection, we will use only rules, which are non-deleting on objects. Matches may in general be non-injective, but if there are contradicting attribute constraints for objects identified by the match, then the rule is not applicable. Operation invocations leading to a non-applicable rule should result in some kind of error handling, e. g. by throwing an exception, but the integration of exception handling is outside the scope of this paper and is left as future work.

In addition to the LHS and RHS, negative application conditions (NAC) may be defined for a rule. Such conditions are defined as non-injective extensions of the LHS, where non-injectivity is used to forbid the identification of certain elements by the match and extensions are used to forbid auxiliary object structures. If the NAC can be matched compatibly with the match of the LHS, then the rule is not applicable.

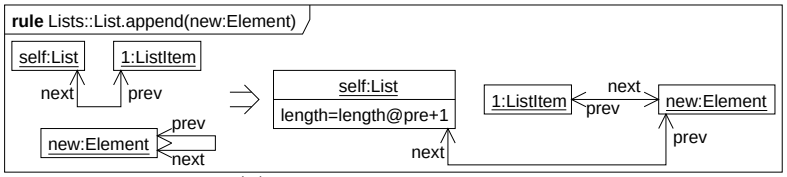
It may be argued that rules, which are allowed to manipulate all structures navigable from the called object and the call's parameters, contradict the object-oriented paradigm of encapsulation of object behaviour, but for the modelling of complex structure changes it seems appropriate to specify them as a rule operating under the control of one of the participating objects rather than dividing the operation, which logically belongs together, into operations on the different objects. In a more elaborated framework, which takes into account visibility and accessibility constraints to facilitate object encapsulation, these visibilities and accessibilities would of course have to be respected by the rules.

For our running example, the transformation rules in Fig. 2 can be used to create lists and elements. The List.create rule in Fig. 2(a) creates an empty list, while the Element.create rule in Fig. 2(b) creates an element containing the given integer as the key and the given string as the data content. The created element is not contained in a list, which is expressed by the next and previous links pointing to the element itself. Since these operations are both

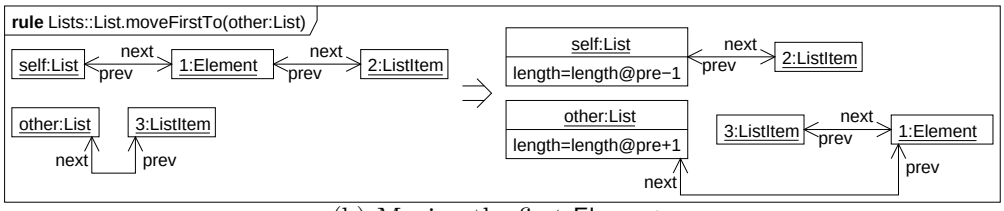


(a) Creating a List (b) Creating an Element

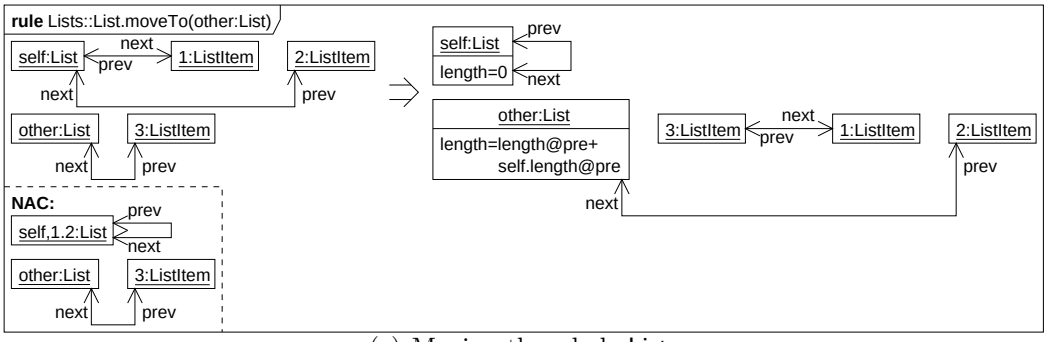
Fig. 2. Transformation rules for creating instances of ListItem



(a) Appending an Element



(b) Moving the first Element

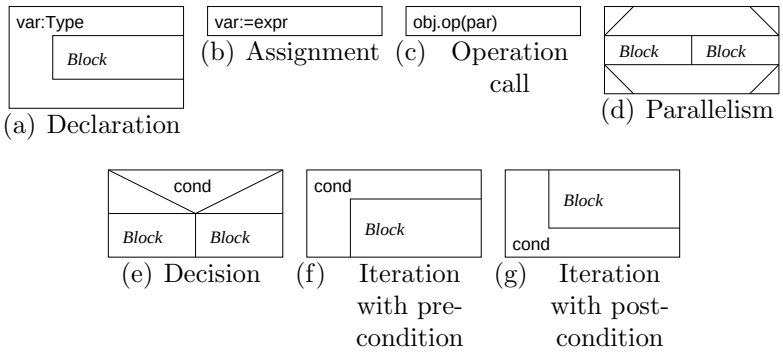


(c) Moving the whole List

Fig. 3. Transformation rules for modifying a List

static, there is no self instance in the LHS. The return parameter is used to transmit the created instances as operation results to the caller.

The rules in Fig. 3 modify a given list. The List.append rule in Fig. 3(a) appends a given element to the end of the list. The rule is not applicapble if new is already contained in another list, because the previous and next links are required to point to new itself. The List.moveFirstTo rule in Fig. 3(b) moves the first element of a list to the end of another list. This rule is not applicable if the self list is empty, because then there is no match for 1:Element, and if the caller tries to move to the same list, because the conflicting attribute specifications for self and other prohibit their identification. The List.moveTo rule in Fig. 3(c) moves a whole list to another empty list. Again, the rule is not applicable, when self and other refer to the same object, because of the conflicting attribute specifications. Additionally, the given NAC forbids



(a) Declaration (b) Assignment (c) Operation call (d) Parallelism

(e) Decision (f) Iteration with pre-condition (g) Iteration with post-condition

Fig. 4. Symbols used in flowcharts

the application on an empty list, because the application would lead to an ill-formed object configuration with the self list contained in the other list.

4 Structured Flowcharts

In this section we use structured flowcharts [8] as defined by Nassi and Shneiderman to describe control flows. The flowcharts are built over the queries and rules defined in the previous sections. Because the details of state changes are delegated to the rule-based operation specifications, the control flows remain concise and comprehensible.

Flowcharts are constructed using the block symbols in Fig. 4, where these blocks can be sequentially composed and recursively inserted for the *Block* nonterminals. A variable declaration shown in Fig. 4(a) consists of a previously undeclared variable var and a type Type. The scope of the variable is the contained block. Values can be assigned to variables by an assignment as shown in Fig. 4(b), where the variable var can be a previously declared variable or the special variable result. Note that neither the parameters of the operation including self, nor attributes may appear on the left side of an assignment, because modifications of the object structure should be specified by rules not by direct assignments. The expression expr with corresponding return type is constructed similar to OCL expressions, but it is, in contrast to OCL, also allowed to contain calls to non-query operations. Operations without return parameter can be called with the block in Fig. 4(c), where obj is a navigation path from a parameter or variable to an object, op is an operation of the class of that object, and par are parameters for the operation. Control flows, which can be executed parallely independent, can be expressed by the block in Fig. 4(d). A decision as in Fig. 4(e) is given by an OCL expression cond with Boolean return type, which is constructed over the parameters and previously declared and defined variables. If the query evaluates to true, the left block is executed, if it evaluates to false, the right block is chosen. The iteration with precondition in Fig. 4(f) corresponds to a while-loop in common programming languages. While the Boolean query cond evaluates to true, the block is executed. Conversely, the iteration with postcondition in Fig. 4(g)

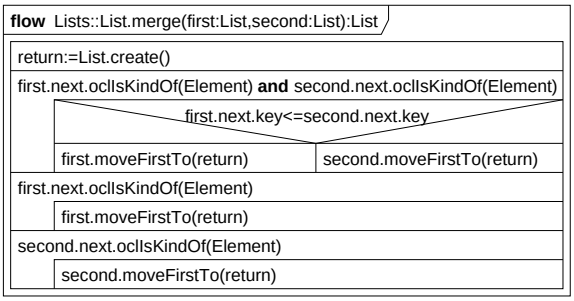


Fig. 5. Flowchart of merging two sorted lists

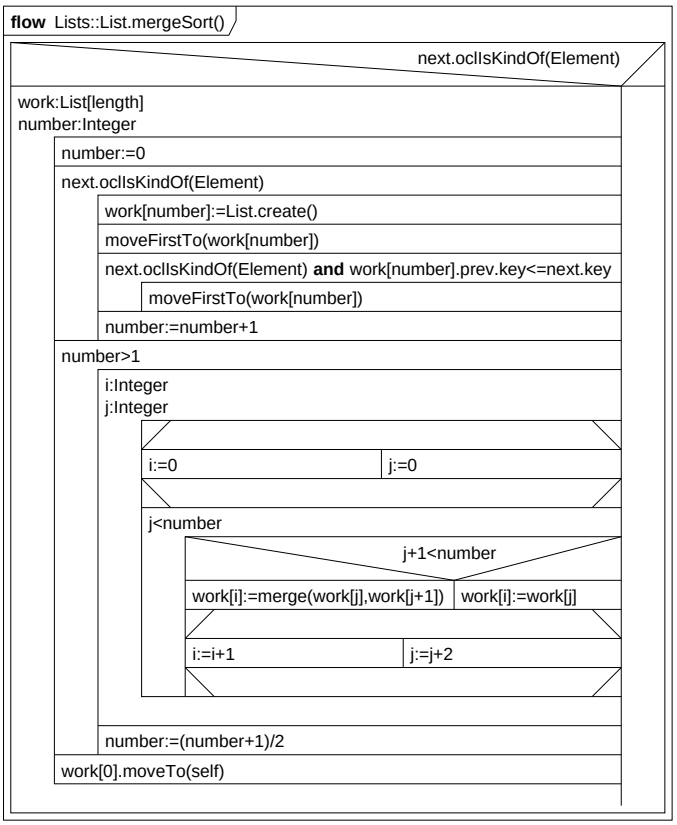


Fig. 6. Flowchart of the Natural Mergesort algorithm

corresponds to a repeat-loop, where cond is evaluated for the first time after the first iteration.

For our example of doubly linked lists, we will specify the Natural Mergesort algorithm [12], which utilises sublists that are already sorted to optimise the performance of the sorting procedure. In Fig. 5 we first specify an auxiliary operation for merging two already sorted lists. As long as both lists contain elements, the first elements are compared and the smaller one is moved to the resulting list. When one of the lists becomes empty, the rest of the other one is moved to the result and the operation terminates.

The Natural Mergesort algorithm itself is specified in Fig. 6. If the list is empty, nothing is done, otherwise the main part of the algorithm is started,

which consists of two parts. First, the list is broken up into already sorted sublists, which are stored in the work array of lists, where this array is declared to contain at most length lists. The actual number of sorted lists is stored in the number variable. Then, these lists are merged pairwise, which halves the number of sorted lists in each pass. This is done until only one list remains and this list is moved to the self list.

5 Summary and Future Work

In this paper an integrated modelling approach for object-oriented systems has been developed, which is organised in three constitutive layers. The first layer employs OCL to yield a functional description of query operations without side effects. On the second layer transformation rules are used to define the behaviour of operations, which change the object configuration locally. Finally, structured flowcharts are used on the third layer to specify complex control flows.

Interesting lines of future work include the extension of the presented approach with respect to further structural and behavioural aspects of the UML, like multiplicities, visibilities, signal and exception handling, and redefinition. The presented languages should be aligned with the UML metamodel by giving metamodels for the three layers. The definition of the abstract syntax by a graph grammar could complement the metamodel, where this also permits the use of graph transformation rules for refinement, refactoring, code generation, and other model transformations.

One of the main motivations for the work in this paper is to define a fully formalized object-oriented modelling technique. Hence, the languages will also be given an integrated formal semantics, which will then be used to facilitate formal verification. For the purpose of verification the constructive modelling techniques presented here will be complemented by descriptive specification techniques like OCL invariants and pre- and post-conditions or UML sequence diagrams. The system described by the constructive techniques can then be verified against the properties required by the descriptive techniques using a variety of verification methods.

Acknowledgement

I would like to thank Gabi Taentzer, Hartmut Ehrig, Rayo Kniep, and the anonymous referees for their valuable comments on previous versions of this paper.

References

[1] Ehrig, H., R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, *Algebraic approaches to graph transformation – Part II: Single*

pushout approach and comparison with double pushout approach, in: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, World Scientific, 1997 pp. 247–312.

[2] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story Diagrams: A new graph rewrite language based on the Unified Modeling Language and Java*, in: *Theory and Application of Graph Transformations, TAGT '98*, LNCS **1764**, Springer, 2000 pp. 296–309.

[3] *Fujaba Homepage*, http://www.fujaba.de/.

[4] Kuske, S., "Transformation Units – A Structuring Principle for Graph Transformation Systems," Ph.D. thesis, Universität Bremen, Bremen, Germany (2000).

[5] Kuske, S., M. Gogolla, R. Kollmann and H.-J. Kreowski, *An integrated semantics for UML class, object, and state diagrams based on graph transformation*, in: *Integrated Formal Methods, IFM 2002*, LNCS **2335**, Springer, 2002 pp. 11–28.

[6] Lüdtke Ferreira, A. P., "Object-Oriented Graph Grammars," Ph.D. thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil (2005).

[7] Lüdtke Ferreira, A. P. and L. Ribeiro, *Towards object-oriented graphs and grammars*, in: *Formal Methods for Open Object-Based Distributed Systems, FMOODS 2003*, LNCS **2884**, Springer, 2003 pp. 16–31.

[8] Nassi, I. and B. Shneiderman, *Flowchart techniques for structured programming*, ACM SIGPLAN Notices **8** (1973), pp. 12–26, http://www.geocities.com/SiliconValley/Way/4748/nsd.html.

[9] Object Management Group, "OCL Specification, Version 2.0," (2005), http://www.omg.org/cgi-bin/doc?ptc/05-06-06.

[10] Object Management Group, "UML Superstructure Specification, v2.0," (2005), http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[11] Ribeiro, L., F. L. Dotti and R. Bardohl, *A formal framework for the development of concurrent object-based systems*, in: *Formal Methods in Software and Systems Modeling*, LNCS **3393**, Springer, 2005 pp. 385–401.

[12] Rolfe, T. J., *List processing: Sort again, naturally*, ACM SIGCSE Bulletin **37** (2005), pp. 46–48, http://penguin.ewu.edu/~trolfe/NaturalMerge/NatMerge.html.

[13] Ziemann, P., K. Hölscher and M. Gogolla, *From UML models to graph transformation systems*, in: *Visual Languages and Formal Methods, VLFM 2004*, ENTCS **127**, Elsevier, 2005 pp. 17–33.