From Algebraic Specifications to Graph Transformation Rules to UML and OCL Models

Martin Gogolla and Karsten Hölscher

Abstract. Graph transformation and algebraic specification are well-established techniques in theoretical and practical computer science and claim to support software development with fundamental methods in a formal manner. Equational algebraic specifications can be translated into a graph transformation system in a systematic way. A graph transformation system in turn can be analyzed and processed by a number of tools. This paper studies how to step from equational algebraic specifications to graph transformation and from there to an operational representation in various graph transformation tools. We work with USE (UML-based Specification Environment), AGG (Attributed Graph Grammar system), and GrGen (Graph rewrite Generator). In particular, we discuss how to establish a connection between algebraic specifications and UML class diagrams and OCL constraints.

1 Introduction

Equational algebraic specifications [EM85,EGL89,Wir90] as well as graph grammars and graph transformation [Roz97,EEKR99] are two fields which have been studied since about thirty years and which share the use of fundamental categorical and algebraic techniques. Both fields claim to support software development with fundamental methods in a formal manner. In recent years, graph transformation attracted substantial research effort because of its closeness to model-driven and model transformation-oriented approaches. For a graph transformation system, practically applicable tools like AGG [dLT04], FUJABA [BGN+04], GrGen [GK07], GReAT [BNvBK06], MOFLON [AKRS06] or GROOVE [Ren03] have been developed and UML tools like USE have been extended to cope with graph transformation [BG06].

The transformation in our paper basically follows the method for translating algebraic specifications into graph transformation which has been proposed in [Löw90] but which has not been realized in a tool (in contrast to the work presented in this paper). Our work is based on the UML and OCL tool USE developed in our group since about ten years [GBR05,GBR07] and on the tools AGG [dLT04] and GrGen [GK07]. This selection of tools was determined by the fact that the authors have experience and knowhow in the use of these tools. We are sure that the other mentioned tools and further ones can be used for our purpose as well.

One main result of the paper is our observation that it is feasible to build a conceptual bridge between so distant fields like *"hard"* algebraic specifica-

F. Drewes, A. Habel, B. Hoffmann, D. Plump (Eds.): Manipulation of Graphs, Algebras and Pictures. Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday, pp. 149–169, 2009.

tion (AlgSpec) and *"soft"* popular approaches like the Unified Modeling Language (UML). We regard Graph Transformation (GraTra) as the *missing link*. GraTra tools provide the possibility of analyzing the underlying model. For example, GraTra tools apart from validating the model are able to check the model consistency or can provide a critical pair analysis of the underlying equations in the algebraic specification. Thus a central ingredient in the interplay between AlgSpec and UML is the ability of GraTra to mediate between the other fields and to broadcast results in both directions. There are some scientists which have been working in all three fields. Among them is Hans-Jörg Kreowski. According to DBLP, his earliest contribution in the GraTra field is from 1977 [Kre77], the earliest one on AlgSpec is from 1978 [EKW78], and the earliest one on UML is from 2002 [KGKK02].

The structure of the rest of the paper is as follows. Section 2 introduces our simple running example within the context of equational algebraic specification and graph transformation. Sections 3, 4 and 5 discuss the realization of this example in the tools USE, AGG, and GrGen, respectively. The paper is finished with concluding remarks in Sect. 6.

2 Running Example

We will study the relationship between equational algebraic specification, graph transformation, and tool realizations of graph transformation by a very simple equational algebraic specification for the natural numbers as shown in Fig. 1. The specification includes the two constructors `zero` and `succ` and an operation `plus` realizing the addition on natural numbers. Any term incorporating the operation `plus` can be reduced by means of the equations to a term using only the constructors `zero` and `succ`, and the different terms built over `zero` and `succ` represent all values for the sort `nat`.

```
spec Nat
srts nat
opns zero: -> nat
     succ: nat -> nat
     plus: nat nat -> nat
vars N, M: nat
eqns plus(zero,N) = N
     plus(succ(N),M) = succ(plus(N,M))
```

Fig. 1. Algebraic Example Specification for Addition on Natural Numbers

In Fig. 2 we have represented the two example equations as two graph transformation rules with left and right side: Each operation symbol and each variable becomes a node and the edges connect operation symbols with their arguments. The first argument of the operation `plus` is established with edges labelled `PlusNat1` and the second argument with label `PlusNat2`. Analogously, the argument of the constructor `succ` shows the label `SuccNat`. Although the

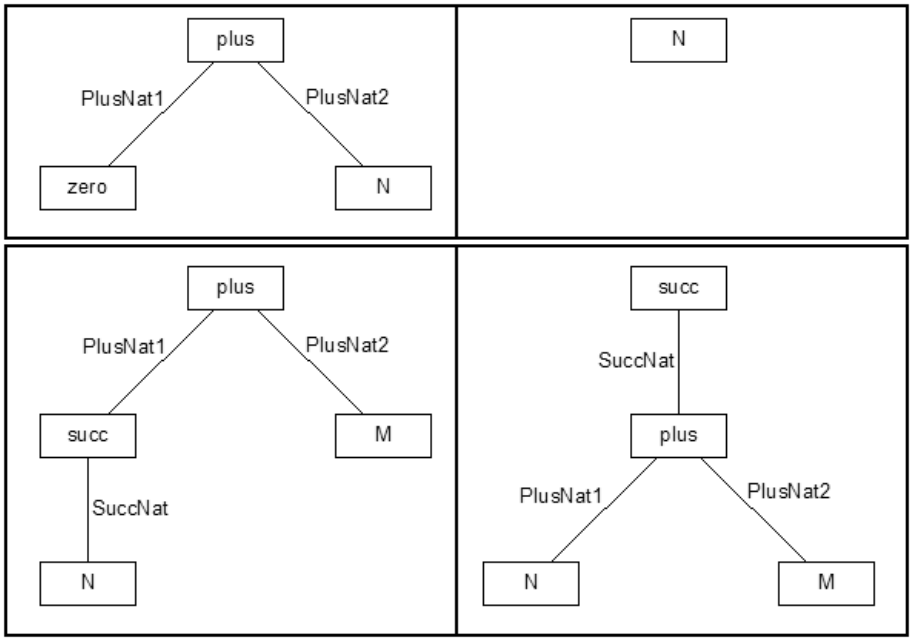

Fig. 2. Naive Representation of Example Equations as a Graph Transformation System

representation seems straight forward, it is too naive for general graph transformation and must be extended as explained below.

The main problem with the above representation in Fig. 2 lies in the fact that the context in which the rules are to be applied is not handled properly. For example, if the first rule is applied in the term `succ(plus(zero,zero))`, the context information that in this case `plus` is a subterm of `succ` is not preserved by the rule. In order to preserve this context information additional nodes are introduced. These nodes embody the context information and explicitly state the type information for each term. This extended representation of the rules is pictured in Fig. 3. The context information is held within the `NatType` nodes. In both rules it is essential, that the topmost `NatType` nodes are preserved by the rules, i.e., the topmost `NatType` nodes appear in the left *and* right side of the rules. Roughly speaking, incoming edges for both rules are handled by the `NatType` nodes `(1)`. Outgoing edges for the first rule are handled by node `(6)` whereas outgoing edges for the second rule are handled by nodes `(6)` and `(8)`.

3 USE

The example graph transformation system corresponding to the algebraic specification is given to USE in textual form. First, the underlying UML class diagram including classes, inheritance relationships, and associations is stated. The overall class diagram is shown in Fig. 4. Currently, our translator from rules to OCL

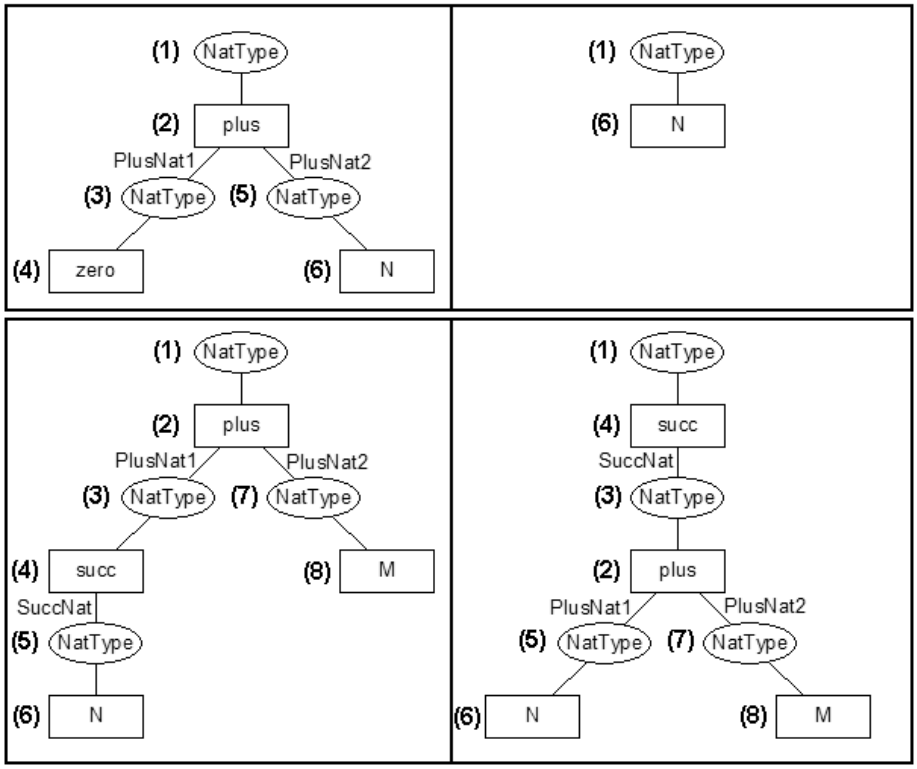

Fig. 3. Representation of Example Equations as a Graph Transformation System

only supports `0..*` multiplicities. Therefore only those multiplicities are stated in the class diagram, although more restricting multiplicities could be chosen.

In addition to the class diagram, the two rules are stated in textual form in Fig. 5 and are called `plusZeroN_2_N` and `plusSuccNM_2_succPlusNM`. These names will be used for generated UML operations. The rules basically make declarations for nodes and edges on the left and right hand side of the rule. In the UML class diagram context, nodes correspond to objects and edges to links. Additionally, OCL conditions could be declared in the left or right hand side, although this feature is not used in this example. OCL conditions in the left side correspond to rule preconditions and OCL conditions in the right side to rule postconditions. Left hand side conditions are called application conditions within the graph transformation area. A representation of the rules in form of UML object diagrams is pictured in Fig. 6.
In Fig. 7 the class diagram generated from the rules resp. the operations generated from the rules are pictured. Each rule induces three operations: The first operation is responsible for applying the rule and for replacing in the so-called working graph a matching left-hand side by the rule's right hand side; the second

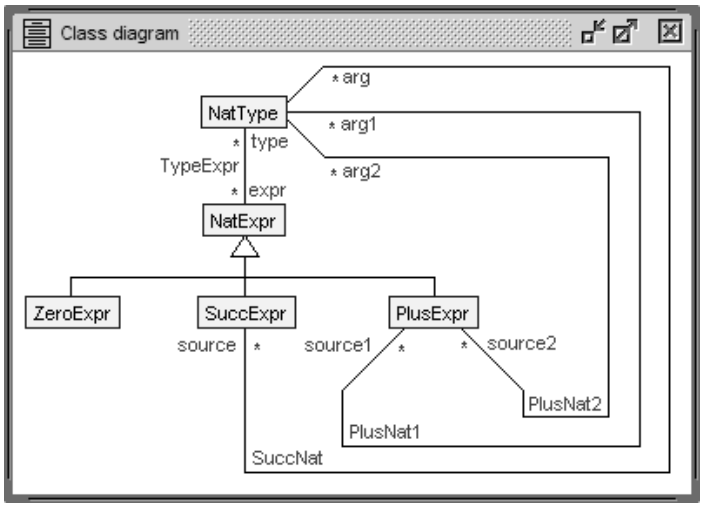

Fig. 4. UML Class Diagram for Terms over Natural Numbers Employed in USE

operation checks the precondition of the rule, and the third operation searches in the working graph for rule redexes, i.e., locations in the working graph where the rule can be applied. The parameters of the operations are determined by the objects (nodes) appearing on the left hand side of the rule.

An example for the reduction of a working graph is given in Fig. 8, basically as a sequence of working graphs in form of UML object diagrams. The reduction corresponds to the calculation `[(0+1)+(0)]+1 = [(0)+(0)]+1+1 = 0+1+1`. The lower part shows the calculation close to a mathematical notation, the middle part uses the naive term representation, and the upper part employs the correct detailed representation with nodes representing the types. The left column represents the term `[(0+1)+(0)]+1`, the middle column the term `[(0)+(0)]+1+1` and the right column the term `0+1+1`. The transition from the left column to the middle column is basically induced by an operation call to `plusSuccNM_2_succPlusNM` corresponding to an application of the second rule from Fig. 3 and the transition from the middle column to the right column by an operation call to `plusZeroN_2_N` corresponding to an application of the first rule from Fig. 3.

The example calculation is also pictured in Fig. 9 in form of a UML sequence diagram. The commands and calls in the sequence diagram can be classified into three parts: The first commands build up the working graph by creating objects representing the start term `[(0+1)+(0)]+1`, the second part is the application of the second rule, and the third part shows the application of the first rule. In the first part, also the links are introduced, however this is not shown in the sequence diagram in order to keep the diagram small. Because object-oriented ideas stand behind USE, every operation call must be directed to an object. Therefore, exactly one object `rc` of class `RuleCollection` is created. The following calls are directed to this object `rc`. The operation call in the third part which corresponds

```
-- plus(zero,N) = N
rule  plusZeroN_2_N
left  N:NatExpr
      NT:NatType
      zero:ZeroExpr
      zeroT:NatType
      plus:PlusExpr
      plusT:NatType
      --
      (NT,N):TypeExpr
      (zeroT,zero):TypeExpr
      (plusT,plus):TypeExpr
      --
      (plus,zeroT):PlusNat1
      (plus,NT):PlusNat2
right N:NatExpr
      --
      plusT:NatType
      --
      (plusT,N):TypeExpr
```

```
-- plus(succ(N),M) = succ(plus(N,M))
rule plusSuccNM_2_succPlusNM
left  N:NatExpr
      NT:NatType
      M:NatExpr
      MT:NatType
      succ:SuccExpr
      succT:NatType
      plus:PlusExpr
      plusT:NatType
      --
      (NT,N):TypeExpr
      (MT,M):TypeExpr
      (succT,succ):TypeExpr
      (plusT,plus):TypeExpr
      --
      (succ,NT):SuccNat
      (plus,succT):PlusNat1
      (plus,MT):PlusNat2
right N:NatExpr
      NT:NatType
      M:NatExpr
      MT:NatType
      succ:SuccExpr
      succT:NatType
      plus:PlusExpr
      plusT:NatType
      --
      (NT,N):TypeExpr
      (MT,M):TypeExpr
      (plusT,succ):TypeExpr
      (succT,plus):TypeExpr
      --
      (succ,succT):SuccNat
      (plus,NT):PlusNat1
      (plus,MT):PlusNat2
```

Fig. 5. Representation of Example Equations in Textual Form

to the application of first rule eliminates certain objects which corresponds to the fact that this rule deletes nodes. The redexes for rule application are determined by calls to the `redexes` operations.

Finally let us comment on the role of OCL within our approach. In USE, OCL plays a central role. In our view, OCL is the formal specification language of the UML and is comparable to other formal specification languages like Z or CASL although the emphasis is much more on practical usability than on theoretical underpinning, for example, on proof theory. The representation of graph transformation in USE is achieved by representing a rule as an operation which is characterized by OCL pre- and postconditions and an operational realization as a command script. Furthermore, OCL can be employed for analyzing the work-

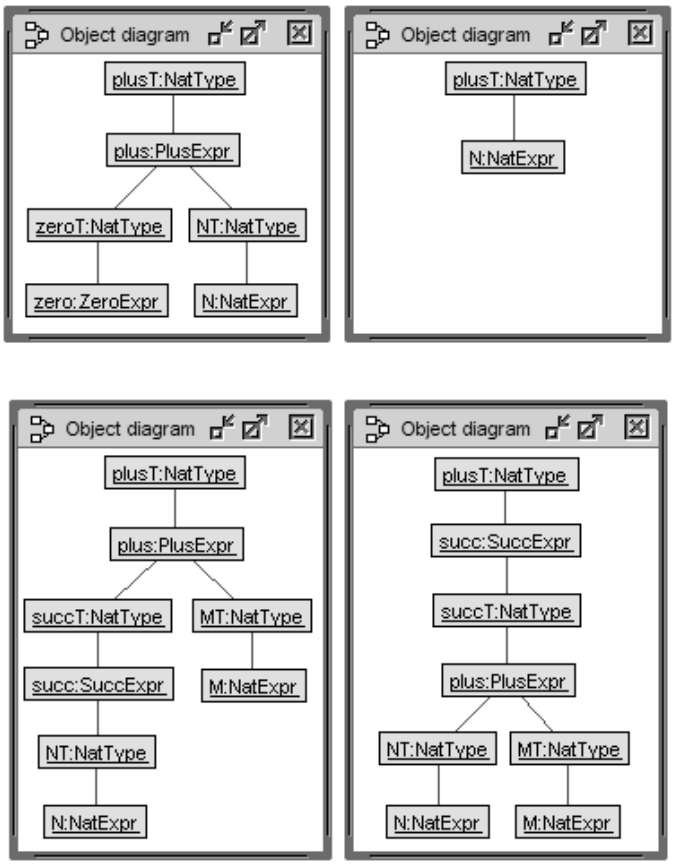

Fig. 6. Representation of Example Equations as Object Diagram Pairs

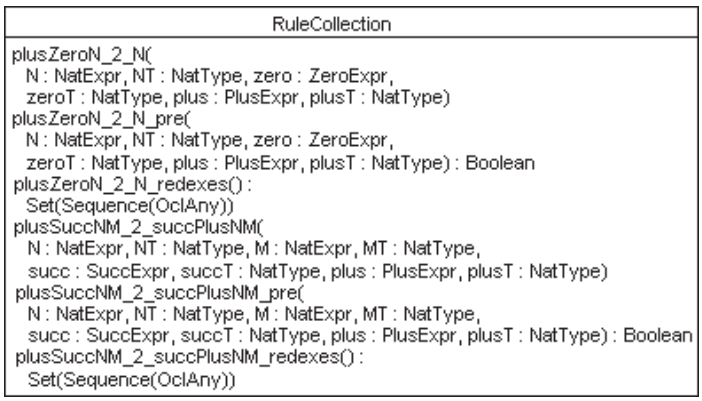

Fig. 7. Class Diagram Induced by the Rules

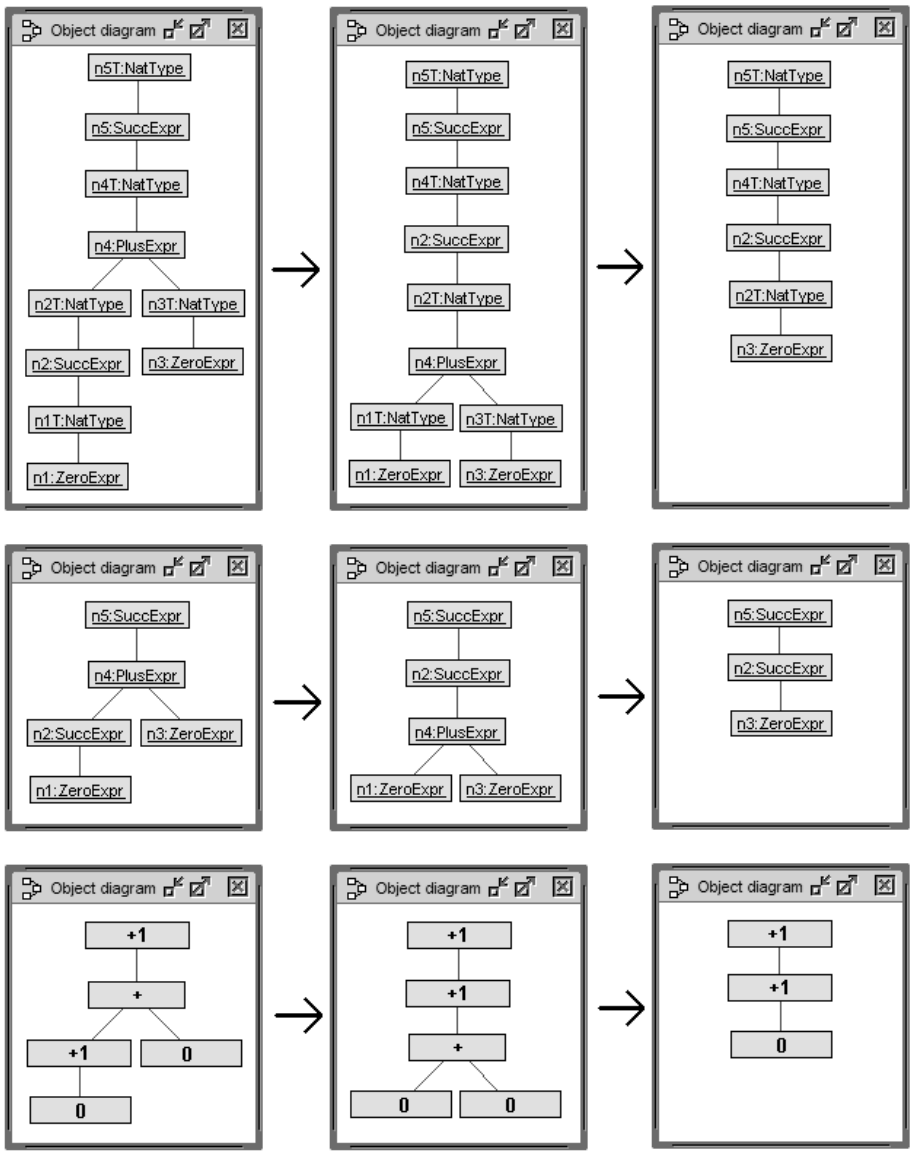

Fig. 8. Example Calculation as an Object Diagram Filmstrip

ing graph at any stage during its development by querying the underlying UML object diagram.

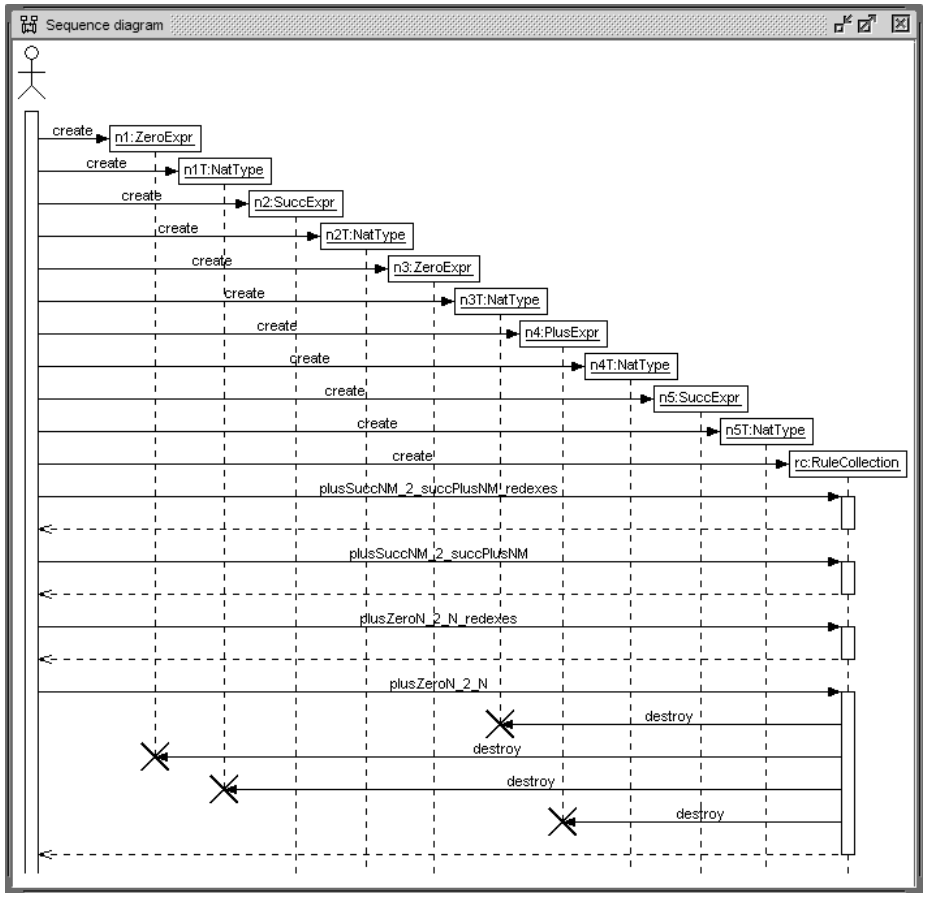

Fig. 9. Example Calculation as a UML Sequence Diagram

4 AGG

In order to animate graph transformation in AGG it is necessary to specify a graph grammar first. Such a grammar makes declarations for the node and edge types of the needed graph items. In the case of the sample algebraic specification from Fig. 1, a straightforward translation requires a node of type `Nat` as a base node for the sort. Additionally, a node of type `NatType` is needed for the same reason as explained in Section 2. Now for every operation of the algebraic specification a corresponding node type being a subtype of `Nat` is needed. An inheritance relation cannot be specified in the AGG grammar, but in a type graph for the corresponding grammar. For this reason the type graph in Fig. 10 is created. In AGG this can be done in a graphical editor. The type graph specifies the nodes `PlusExp`, `SuccExp`, and `ZeroExp` to be subtypes of the node `Nat`. Additionally edge types are declared for the arguments of the operations defined in the algebraic specification. Since the order of these arguments is usually im-

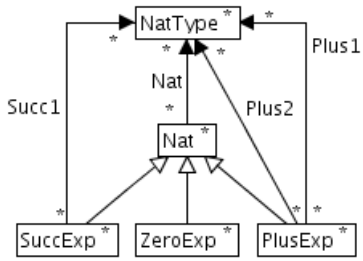

Fig. 10. Type Graph for Terms over Natural Numbers Employed in AGG

portant, a digit is appended to the type names. So the additional edge types are `Succ1`, `Plus1`, and `Plus2`.

Having specified a suitable graph grammar, it is now possible to create a host graph for the transformation, which can also be done in a graphical editor. Fig. 11 shows the host graph for the sample term `succ(plus(succ(zero),zero))`. The recipe for creating a host graph of a given equation is to read the equation from left to right and add the corresponding nodes and edges in that order. For every node that represents a sort it is additionally necessary to add a corresponding node for the context.

The sample term in Fig. 11 starts with `succ`, so a node of type `SuccExp` is added. This node is connected to a newly created context node `NatType` using an edge of type `Nat`. Since a `Nat` node is always attached to a `NatType` context node in this way, it is hence refered to as a `Nat` node pair.

The next operation occurring in the term is `plus`, so a `PlusExp` node pair is inserted into the graph. Since this node pair is an argument to the previous `succ` operation, an edge of type `Succ1` is added which connects the `SuccExp` node with the `NatType` node belonging to the new `PlusExp` node. The remainder of the equation can be treated in an analogous way.

Now that we have specified a host graph, the actual graph transformation rules can be derived. The specification contains two equations. These equations can be translated into a graph transformation rule in an analogous way as the translation above. The left-hand side of the equation is translated to a graph which becomes the left-hand side of the rule. Similarly the right-hand side of the equation becomes the right-hand side of the rule. The rule creation is finished by the specification of those elements that have to be preserved when the rule is actually applied.

Consider the first equation `plus(zero,N)=N` of the specification. The left-hand side of this equation, i.e., `plus(zero,N)` has to be translated into a graph first. As previously explained, this is accomplished by reading the expression from left to right and inserting corresponding elements into the graph. The expression starts with `plus`, so a new `PlusExp` node pair is inserted into the graph. The first argument of the operation `plus` is `zero`, so a new `ZeroExp` node pair is added to

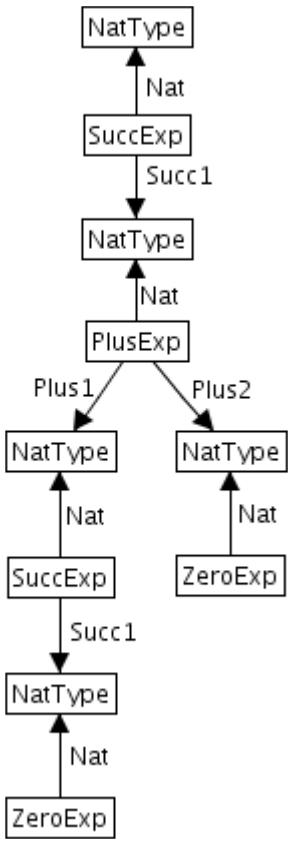

Fig. 11. AGG Graph Representation of `succ(plus(succ(zero),zero))`

the graph. Additionally the `PlusExp` node is connected to the `NatType` node of the `ZeroExp` node pair with an edge of type `Plus1`. The second argument of `plus` in the equation is `N`, so a new `Nat` node pair is created. The `NatType` node of this pair is connected to the `PlusExp` node with an edge of type `Plus2`. Since `N` is not further specified, the concrete subtype is not known. For this reason a node of the supertype `Nat` is used.

The right-hand side of the equation is `N`. So the graph for the right-hand side of the new rule contains only a `Nat` node pair.

In AGG a rule can also be created in a graphical editor. Fig. 12 shows a screenshot of the rule corresponding to the equation `plus(zero,N)=N`.

The graph to the left of the vertical bar is the left-hand side of the rule while the graph to its right is the right-hand side. The items that have to be preserved are represented by identical numbers in graph elements of the left-hand and the right-hand side. In this case, only the `NatType` node indicated by `1:` and the `Nat` node indicated by `2:` are specified as elements that have to be preserved when the rule is applied. Since the equation specifies that `plus` and `zero` do not occur

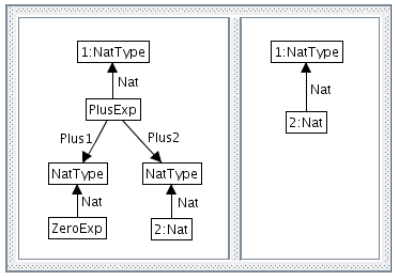

Fig. 12. AGG Rule Representing the Equation `plus(zero,N)=N`

in the right-hand side, the graph transformation rule specifies to delete the corresponding nodes. It may be an intuitive approach to simply keep the `N:Nat` node pair when the rule is applied, but this may yield a wrong result. If the `NatType` node of the `PlusExp` node pair is connected to another node, e.g., to a `SuccExp` node via a `Succ1` edge (representing an expression like `succ(plus(...))`, then this connection would be deleted together with the respective `NatType` node. For this reason, the `NatType` node of the `Nat` node pair representing the first term in the equation expression always has to be preserved. The additional context following `N` is preserved anyway, since the `Nat` node representing `N` is preserved and with it all its connections.

So the rule creation works as stated above bearing in mind that the the topmost (in the sense of no incoming edges) `NatType` of the left-hand side has to be preserved.

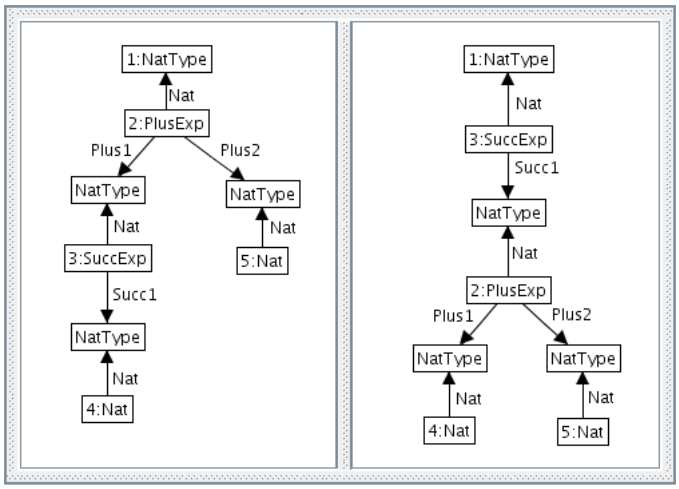

Fig. 13. AGG Rule Representing the Equation `plus(succ(N),M)=succ(plus(N,M))`

Fig. 13 pictures the AGG rule that corresponds to the second equation `plus(succ(N),M) = succ(plus(N,M))` of the specification. It has been created analogously to the first rule.

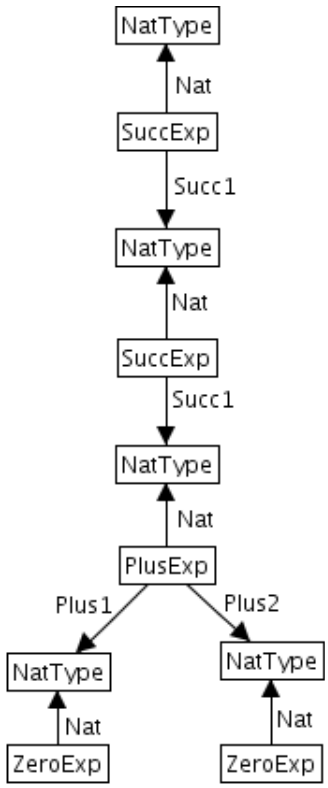

Fig. 14. Transformed Graph After Applying the First Rule in AGG

Considering the host graph from Fig. 11 the first rule cannot be applied. This holds, since the first argument of `plus` would have to be `zero` for the rule to be applicable. The second rule can be applied in exactly one fashion and in the same way as one would apply the second equation. It yields the transformed graph in Fig. 14. The figure shows a screenshot of the actual transformation in AGG which can directly be observed in the GUI version of the tool.

The second rule is not applicable to the transformed graph. This holds, since the rule expects a `succ` as first argument of `plus`. But in the expression represented by the graph, the first argument of the only `plus` is `zero`. For this reason the first graph transformation rule is applicable, yielding the transformed graph depicted in Fig. 15. It represents the term `succ(succ(zero))`, which is the expected result.

5 GrGen

In GrGen the specification of the underlying graph model as well as the graph transformation rules is given in textual form. Since GrGen supports subtypes for nodes and edges as well as subtype matching in rule application, the algebraic

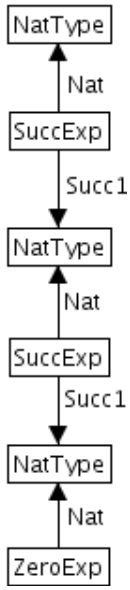

Fig. 15. Transformed Graph After Applying the Second Rule in AGG

specification can be translated in a straight-forward way. The graph model can be derived from a specification in the following way. For every sort there is a node type with the same name and a context node type. Then for every operation that yields a certain sort, there is a node type which extends the node type representing the sort. For every argument of an operation there is an edge type with a type name consisting of the operation name and a successive number to indicate the order of the arguments. Therefore, in the case of the running example specification from Fig. 1, the GrGen graph model description can be stated textually as follows.

```
node class NatType;
node class Nat;

node class Zero extends Nat;
node class Succ extends Nat;
node class Plus extends Nat;

edge class nat;
edge class succ1;
edge class plus1;
edge class plus2;
```

This model can also be pictured as a UML class diagram as shown in Fig. 16. In addition to the simple, but sufficient model we have used here, GrGen would also allow to declare the edges to possess more specific types.

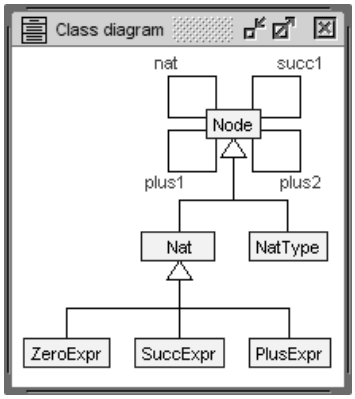

Fig. 16. UML Class Diagram for Terms over Natural Numbers Employed in GrGen

In GrGen a graph transformation rule consists of a `pattern` part and a `replace` part. The `pattern` part represents the left-hand side of the rule, while the `replace` part corresponds to the right-hand side. The graph items that have to be preserved are indicated by using the same identifiers for nodes and edges in both parts.

In order to specify the left-hand side of the rule for `plus(zero,N)` a node pair consisting of the corresponding `Nat` node connected to its context `NatType` node is inserted for every `Nat` expression. Then connecting edges are specified for the operation arguments, which can be directly derived from the specification. The right-hand side is specified analogously. Similarly to the AGG specification the topmost `NatType` node of the left-hand side has to be mapped to the topmost `NatType` node of the right-hand side in order to preserve a possible context. So the first rule looks like this:

```
rule plusZeroN {
  plus:Plus -:nat-> plusType:NatType;
  zero:Zero -:nat-> zeroType:NatType;
  n:Nat -:nat-> nType:NatType;
  plus -:plus1-> zeroType;
  plus -:plus2-> nType;

  replace {
    n -:nat-> plusType;
  }
}
```

Analogously the second rule looks like this:

```
rule plusSuccNM {
  plus:Plus -:nat-> plusType:NatType;
  suc:Succ -:nat-> sucType:NatType;
  n:Nat -:nat-> nType:NatType;
  m:Nat -:nat-> mType:NatType;
  plus -:plus1-> sucType;
  plus -:plus2-> mType;
  suc -:succ1-> nType;

  replace {
    suc -:nat-> plusType;
    suc -:succ1-> newPlusType:NatType;
    plus -:nat-> newPlusType;
    plus -:plus1-> newNType:NatType;
    n -:nat-> newNType;
    plus -:plus2-> newMType:NatType;
    m -:nat-> newMType;
  }
}
```

The actual graph transformation is executed in GrGen's *grshell*. Within grshell, a graph transformation specification can be loaded and a graph can be created manually or by a script. Testing the above specification using an initial graph representing the sample term `succ(plus(succ(zero),zero))` yields the expected result. Initially only the rule `plusSuccNM` and after that only the rule `plusSuccNM` is applicable. For debuging purposes, GrGen is shipped with yComp, a graph visualization tool that draws the current host graph handled in GrGen. Fig. 17 shows a screenshot of the graph after the two rule applications. As expected it is the graph representation of the term `succ(succ(zero))`.

6 Conclusion

This paper has explained how algebraic specification in their basic form as conditional equations can be represented as a graph transformation system and how the result can be validated, animated, and executed in various graph transformation tools. These graph transformation tools offer the possibility of analyzing the underlying model (although we have not demonstrated this feature). We have considered the equational specification as a rewriting system which works in one direction only. The work shows that classical software specification techniques still have a close connection to modern object-oriented techniques like UML. The translation may also be seen as an example for a conceptual model transformation from one computer science field (Algebraic Specification) into another one (UML and OCL). Another aspect of the current work was to show and compare the graph models in the different tools by formally fixed UML class diagrams. These different graph models underpin the flexibility of current graph transformation tools.

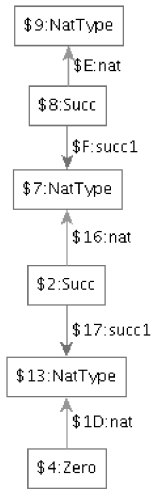

Fig. 17. Final Graph in GrGen Drawn with yComp

Future work might concentrate on the question how to utilize the strengths and analysis features of the different tools in order to give feedback to system developers. Apart from the considered tools, other tools like FUJABA, MOFLON, GReAT, GROOVE, VIATRA, or VMTS might be taken into consideration. The equations might also be treated as rules in both directions. We think that for courses on formal software development the translation which we have proposed gives insight into connections between the different computer science fields, namely algebraic specification, graph transformation, and model-driven development.

References

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink, J. Warmer, editors, *Proc. 2nd Eur. Conf. Model Driven Architecture (ECMDA'2006)*, 361–375. LNCS 4066, Springer, Berlin, 2006.

[BG06] F. Büttner and M. Gogolla. Realizing Graph Transformations by Pre- and Postconditions and Command Sequences. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Proc. 3rd Int. Conf. Graph Transformations (ICGT'2006)*, 398-412. LNCS 4178, Springer, Berlin, 2006.

[BGN+04] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level: The FUJABA Approach. *STTT*, 6(3):203-218, 2004.

[BNvBK06] D. Balasubramanian, A. Narayanan, C.P. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECEASST*, 1, 2006.

[dLT04] J. de Lara and G. Taentzer. Automated Model Transformation and Its Validation Using AToM 3 and AGG. In A.F. Blackwell, K. Marriott, and A. Shimojima, editors, *Diagrams*, LNCS 2980, 182-198. Springer, 2004.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* World Scientific, Singapore, 1999.

[EGL89] H.-D. Ehrich, M. Gogolla, U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, Stuttgart, 1989.

[EKW78] H. Ehrig, H.-J. Kreowski, H. Weber. *Algebraic Specification Schemes for Data Base Systems.* In S. Bing Yao, editor, Proc. 4th Int. Conf. Very Large Data Bases, IEEE, 427-440, 1978.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification.* Springer, Berlin, Germany, 1985.

[GBR05] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386-398, 2005.

[GBR07] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27-34, 2007.

[GK07] R. Geiß and M. Kroll. GrGen.Net: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *AGTIVE*, *LNCS 5088*, 568-569. Springer, 2007.

[Kre77] H.-J. Kreowski. *Transformations of Derivation Sequences in Graph Grammars.* In M.Karpinski, editor, Proc. 1st Int. Conf. Fundamentals of Computation Theory, Springer, LNCS 56, 275-286, 1977.

[KGKK02] S. Kuske and M. Gogolla and R. Kollmann and H.-J. Kreowski. *An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation.* In M.J. Butler, L. Petre, K. Sere, editors, Proc. 3rd Int. Conf. Integrated Formal Methods, Springer, LNCS 2335, 11-28, 2002.

[Löw90] M. Löwe. Implementing Algebraic Specifications by Graph Transformation Systems. *Elektronische Informationsverarbeitung und Kybernetik*, 26 (11/12):615-641, 1990.

[Ren03] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J.L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, *LNCS 3062*, 479-485. Springer, 2003.

[Roz97] G. Rozenberg, editor. *Handbook on Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific, Singapore, 1997.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* North-Holland, 1990.

Appendix: USE Protocol for the Example Calculation

```
-------------------------------------------------------------------------
use> open alg2ocl.use
-------------------------------------------------------------------------
use> !create n1:ZeroExpr
use> !create n1T:NatType
use> !insert (n1T,n1) into TypeExpr

use> !create n2:SuccExpr
use> !create n2T:NatType
use> !insert (n2T,n2) into TypeExpr

use> !insert (n2,n1T) into SuccNat

use> !create n3:ZeroExpr
use> !create n3T:NatType
use> !insert (n3T,n3) into TypeExpr

use> !create n4:PlusExpr
use> !create n4T:NatType
use> !insert (n4T,n4) into TypeExpr

use> !insert (n4,n2T) into PlusNat1
use> !insert (n4,n3T) into PlusNat2

use> !create n5:SuccExpr
use> !create n5T:NatType
use> !insert (n5T,n5) into TypeExpr

use> !insert (n5,n4T) into SuccNat
-------------------------------------------------------------------------
use> !create rc:RuleCollection

use> ?rc.plusZeroN_2_N_redexes()
     Set{}: Set(Sequence(OclAny))
use> ?rc.plusSuccNM_2_succPlusNM_redexes()
     Set{Sequence{@n1,@n1T,@n3,@n3T,@n2,@n2T,@n4,@n4T}}:
       Set(Sequence(OclAny))
-------------------------------------------------------------------------
use> read plusSuccNM_2_succPlusNM_find_redex.cmd
     -- the following commands are executed by reading the command file
CMD> !let _redex = rc.plusSuccNM_2_succPlusNM_redexes()->any(true)
CMD> !let _N = _redex->at(1)
CMD> !let _NT = _redex->at(2)
CMD> !let _M = _redex->at(3)
CMD> !let _MT = _redex->at(4)
CMD> !let _succ = _redex->at(5)
CMD> !let _succT = _redex->at(6)
CMD> !let _plus = _redex->at(7)
```

```
CMD> !let _plusT = _redex->at(8)
CMD> !openter rc plusSuccNM_2_succPlusNM
       (_N,_NT,_M,_MT,_succ,_succT,_plus,_plusT)
     precondition 'plusSuccNM_2_succPlusNM_pre' is true
CMD> !insert(_plusT,_succ) into TypeExpr
CMD> !insert(_succT,_plus) into TypeExpr
CMD> !insert(_succ,_succT) into SuccNat
CMD> !insert(_plus,_NT) into PlusNat1
CMD> !delete(_succT,_succ) from TypeExpr
CMD> !delete(_plusT,_plus) from TypeExpr
CMD> !delete(_succ,_NT) from SuccNat
CMD> !delete(_plus,_succT) from PlusNat1
CMD> !opexit
     postcondition 'plusSuccNM_2_succPlusNM_post' is true
------------------------------------------------------------------------
use> ?rc.plusZeroN_2_N_redexes()
     Set{Sequence{@n3,@n3T,@n1,@n1T,@n4,@n2T}}: Set(Sequence(OclAny))
use> ?rc.plusSuccNM_2_succPlusNM_redexes()
     Set{}: Set(Sequence(OclAny))
------------------------------------------------------------------------
use> read plusZeroN_2_N_find_redex.cmd
     -- the following commands are executed by reading the command file
CMD> !let _redex = rc.plusZeroN_2_N_redexes()->any(true)
CMD> !let _N = _redex->at(1)
CMD> !let _NT = _redex->at(2)
CMD> !let _zero = _redex->at(3)
CMD> !let _zeroT = _redex->at(4)
CMD> !let _plus = _redex->at(5)
CMD> !let _plusT = _redex->at(6)
CMD> !openter rc plusZeroN_2_N(_N,_NT,_zero,_zeroT,_plus,_plusT)
     precondition 'plusZeroN_2_N_pre' is true
CMD> !insert(_plusT,_N) into TypeExpr
CMD> !delete(_NT,_N) from TypeExpr
CMD> !delete(_zeroT,_zero) from TypeExpr
CMD> !delete(_plusT,_plus) from TypeExpr
CMD> !delete(_plus,_zeroT) from PlusNat1
CMD> !delete(_plus,_NT) from PlusNat2
CMD> !destroy _NT
CMD> !destroy _zero
CMD> !destroy _zeroT
CMD> !destroy _plus
CMD> !opexit
     postcondition 'plusZeroN_2_N_post' is true
------------------------------------------------------------------------
```

Prof. Dr. Martin Gogolla

Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
gogolla@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Being a professor for Computer Science at the University of Bremen, Martin Gogolla has been a colleague of Hans-Jörg Kreowski since 1994. They first met in 1982 the 1st Workshop on Abstract Data Types, and have collaborated in both international (e.g., COMPASS) and national projects (e.g., UML-AID).

Dr. Karsten Hölscher

Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
hoelsch@informatik.uni-bremen.de

Karsten Hölscher studied Computer Science at the University of Bremen. He was introduced to Theoretical Computer Science by Hans-Jörg Kreowski and Frank Drewes, who, during that time, was an assistant professor in Hans-Jörg's team. Karsten graduated in 2003 under Hans-Jörg's supervision and became a doctoral student in his group, working as a research associate from 2003 to 2008. Karsten received his doctoral degree in September 2008 under Hans-Jörg's supervision. After a short intermezzo in software industry, Karsten returned to the University of Bremen, switching sides to a more practical field. He now works as a research associate in the Software Engineering Group headed by Rainer Koschke.