 Electronic Notes in Theoretical Computer Science 211 (2008) 241–250 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

An Example of Cloning Graph Transformation Rules for Programming

Mark Minas[1](#_bookmark0)

*Universit¨at der Bundeswehr Mu¨nchen 85577 Neubiberg, Germany*

Berthold Hoffmann[2](#_bookmark0)

*Technologiezentrum Informatik Universit¨at Bremen*

*28334 Bremen, Germany*

**Abstract**

Graphical notations are already popular for the design of software, as witnessed by the success of the Uniform Modeling Languages (UML). In this paper, we advocate the use of graphs and graph transformation for programming graph-based systems. Our case study, the flattening of hierarchical statecharts, reveals that *cloning*, a recently proposed transformation concept, makes graph transformation rules (in the double-

pushout approach) more expressive. Thus programming becomes easier, and gets along with simpler control conditions in particular.

*Keywords:* Statecharts, Graph transformation, Clones

# Introduction

Visual notations have always been popular for designing software, even more since the appearance of the *Uniﬁed Modeling Language* (UML), a family of mostly graph- like diagram languages. On the contrary, visual programming with graphs is still far less popular, although graphs are a convenient data structure and the compu- tational model of graph transformation is well developed. We suspect that this is because graph transformation rules alone are not enough. For programming, transformations have to be extended by control conditions (Grace [[1](#_bookmark16)]), by control programs (Progres [[6](#_bookmark21)]), by control diagrams (Fujaba [[3](#_bookmark17)]), or by control predicates (DiaPlan [[2](#_bookmark18)], GReAT [[5](#_bookmark19)]), and control mechanisms force the user to program in

1 Email: [Mark.Minas@unibw.de](mailto:Mark.Minas@unibw.de)

2 Email: [hof@tzi.de](mailto:hof@tzi.de)

1571-0661 © 2008 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2008.04.046

a way that is very similar to conventional languages. Graph transformation only simplifies the description of a program’s basic operations, but writing down an al- gorithm that applies these operations in a controlled way is hardly improved by current programming languages based on graph transformation.

This paper tries to improve graph-transformation-based programming by sim- plifying its control mechanisms. This requires that basic graph transformation steps become more expressive. We are adopting the concept of *cloning* graph transfor-

mation rules that has been proposed recently [[4](#_bookmark20)]. Cloning allows to express basic program steps with a single transformation rule (scheme) which would otherwise require complicated loops and alternatives. Yet, cloning is a natural approach that does not imply additional efforts to the programmer. This is demonstrated by a case study, the flattening of hierarchical statecharts, i.e., transforming statecharts containing compound states (and-states and or-states) into flat statecharts without them.

The following section briefly introduces the idea of graph transformation with cloning before the problem of flattening statecharts is described in Sect. [3](#_bookmark4). Sect. [4](#_bookmark7) shows the proposed approach of applying graph transformation with cloning to solve this problem. The last section concludes the paper.

# Graph transformation with cloning

This section gives only an informal overview of graph transformation with cloning. Details can be found in [[4](#_bookmark20)]. The idea is to specify rule schemes which represent an in general infinite number of rule instances by cloning certain subgraphs. Before describing the rule schemes, we describe the graphs that constitute their left-hand sides and right-hand sides.

A *pattern* is a graph *G* wherein some nodes are annotated by *cardinality vari- ables*. Each cardinality variable *y* determines a subgraph *Gy* which consists of the nodes annotated with *y* (the *y-fold nodes*), their incident edges, and their adjacent nodes (the *border nodes*). A *y-clone* of pattern *G* is obtained by binding *y* to an

integer value *k ≥* 0, removing all *y*-fold nodes and their incident edges from *G*, and gluing *k* disjoint copies of *Gy* to the border nodes. A pattern gets instantiated by binding each cardinality variable to a non-negative integer value and creating *y*- clones for each variable *y*. The order in which variables are cloned does not matter as cloning is commutative [[4](#_bookmark20)].

A *rule scheme* is a double-pushout rule where left-hand side (lhs), interface, and right-hand side (rhs) are patterns, and which uses pattern morphisms as a straight- forward extension of graph morphisms. Cardinality variables of the rhs have to occur in the lhs also. A rule instance is again a double-pushout rule. A rule is obtained by binding variables of the rhs to the same values as the ones of the lhs and instantiating lhs, interface, and rhs.

As usual, we will omit the interface when drawing a rule scheme, but indicate images of interface nodes by integer numbers. As an example, consider Fig. [6](#_bookmark10)a which is shown as lhs/rhs. Interface nodes are the ones with the same number, i.e.,

|  |  |
| --- | --- |
| And Comp | |
| ontains |  |
| Plain State | |

Fig. 1. Rule instance of the rule scheme shown in Fig. [6](#_bookmark10)a with *x* bound to 3.

*0’*

*1’*

contains

*‘0*

*‘1*

has has has

has has has

*2’*

*4’*

*6’*

*‘2*

*‘4*

*‘6*

contains contains contains

*3’*

contains contains

*‘3 ‘5*

c

*‘7*

Plain State

*5’*

Plain State

*7’*

Plain State

ori ori ori

And Comp

And Comp

And State

Work

And Comp

And Comp

And State

Work

And Comp

Plain State

Plain State

Plain State



B

a

C

D

x b

A

y

a

E

F

a



B

b

x

A

CE

a

a

DE

a

y

a

DF

b

CF

y

y

A

x

y

b

CE

a

a

DE

a

y

a

DF

b

CF

* 1. (b) (c)

Fig. 2. A statechart diagram with an and-state (a), an equivalent statechart diagram where the and-state has been transformed into an or-state (b), and an equivalent, flat statechart diagram (c).

nodes ‘1,‘2, ‘3, and ‘4 of the lhs resp. 1’, 2’, 3’, and 4’ of the rhs. The rule scheme is annotated with cardinality variable *x*. *x*-fold nodes are visualized as a stack of nodes with an inscribed *x*. Fig. [1](#_bookmark2) shows an instance of this rule after creating *x*-clones with *x* bound to 3.

# Statecharts

Statecharts are a visual language for describing behavior; under the name state dia- grams, it is one of the graph-like languages of the UML. Statechart diagrams are an extension of finite state machines where transitions are annotated by events, firing conditions, and actions that have to be evaluated when a transition “fires”. Hier- archical states can be used to simplify the description of complex behavior. There are two kinds of hierarchical states: An *or-state* contains a complete statechart dia- gram. When the or-state is active, then one of its contained states is active. Fig. [2](#_bookmark3)b shows a statechart diagram with the or-state *B*. An *and-state* consists of several and-compartments that are separated by dashed lines. Each compartment contains a complete statechart diagram, like an or-state. When an and-state is active, all statecharts contained in its compartments are active in parallel. Fig. [2](#_bookmark3)a shows a statechart diagram with such an and-state *B*. In order to fit the description of the flattening algorithm into this paper, we are using simplified statechart diagrams in the following. Simplified statecharts consist of a collection of plain states, and- states, or-states, and initial pseudo states. Initial pseudo states cannot be active; as soon as a statechart is activated, its initial pseudo state is entered and immediately left again via its only and non-annotated transition entering its connected state. Final states and history states are not used here. The annotation of transitions is



contains

init *Real State*

from

to

*Con tainer*

And State

And Comp

has

Or State

Plain State

annotation

Transition

Init State

*State*

Fig. 3. Metamodel of the considered statechart diagram language.

contains



And Comp

Init State

y

has

B And State

has

from

to

x

to

init A Plain State

from

Transition

contains from

to

init C Plain

State

D Plain State

b

Transition

Transition

a

Transition

Init State

contains from

to

init E Plain

State

F Plain State

a

Transition

a

Transition

Init State

And Comp

contains

contains

contains

to from to from

Fig. 4. Graph representation of the statechart diagram in Fig. [2](#_bookmark3)a.

also simplified: It may consist of events only; firing conditions and actions are not considered here. Also, transitions are not allowed to cross the borders of and-states and or-states. Finally, we simplify handling of conflicting transitions. UML state- charts give transitions firing from higher levels priority over those firing from lower levels. We ignore priorities and assume non-determinism for simplicity.

Fig. [3](#_bookmark5) defines the metamodel of the considered statechart diagram language as a class diagram. States are either initial pseudo states or real states, i.e., plain states, or-states, or and-states. Or-states are also containers which contain the

states of the contained state diagram. And-states consist of and-compartments (*And*

*Comp* in Fig. [3](#_bookmark5)) which are also containers with contained states. Initial pseudo states are associated to their connected state, and transitions are associated to their connected states by *from* and *to* associations. Transitions have an *annotation* attribute that contains the string-valued event for this transition. Based on this metamodel, each statechart diagram can be represented as a graph. Fig. [4](#_bookmark6) shows the graph representation of the statechart diagram in Fig. [2](#_bookmark3)a.

A transition fires if the state at the transition’s source is active and the event is the transition’s annotation. If more than one state is active within an and- state, all outgoing transitions with corresponding annotations fire. An active state whose outgoing transition fires gets inactive, and an inactive state whose incoming transition fires gets active. If an or-state gets active, the state that is connected with the or-state’s initial pseudo state gets active, too. For and-states, all of its compartments get active like or-states. Or-states and and-states of our simplified

language can be left from any contained state. State *B* of Fig. [2](#_bookmark3)a, e.g., is left as soon as a *y* event occurs where any of the contained states C, D, E, or F may be active.

Based on this semantics of transition firing, we can “flatten” a hierarchical state- chart diagram by replacing and-states and or-states by equivalent sub-statechart diagrams. Fig. [2](#_bookmark3) shows this process: All and-states are first replaced by equivalent or-states (b), and finally, all or-states are removed (c).

Replacing an and-state is the more complicated step. The idea of this step is to turn any combination of active contained states into new states. This is similar to building product automata of finite automata. Therefore, we build the

cross product of state sets for any and-compartment. [3](#_bookmark8) In Fig. [2](#_bookmark3), the contents of

and-state *B* are replaced by the cross product *{C, D}× {E, F}*. Transitions are considered next. This step is different from building product automata because a statechart drops an event if the event cannot be consumed by a firing transition. If more than one statechart is active in an and-state, an event can be consumed by some statecharts, but dropped by the others depending on the currently active states and their outgoing transitions. This defines the transitions that have to be

created: Let (*s*1*,..., sn*) be a state from the cross product. If it is active and an

event *e* occurs, there are some and-compartments’ statecharts which consume *e* by

a transition *si →e ti*, and the other compartments’ statecharts drop *e*, i.e., they stay

in their state *si* = *ti*. Therefore, the equivalent or-state replacing the and-state

must contain a transition (*s*1*,..., sn*) *→e* (*t*1*,..., tn*). Fig. [2](#_bookmark3)b shows the result.

Removing an or-state is actually simple. As one of the contained state is active iff the or-state is active, we can simply drop the or-state frame, but must take care of the transitions from and to the or-state. Transitions to the or-state are redirected to the contained state that is connected to the or-state’s initial pseudo state. And as the or-state can be always left by a transition from the or-state, each transition from the or-state has to be replaced by copies from each of the contained states. Fig. [2](#_bookmark3)c shows the result.

The previous paragraphs have outlined the algorithm of flattening hierarchical statecharts rather coarsely. A precise algorithm in a textual or a common graph- transformation-based language would make use of an abstract representation like the one shown in Fig. [4](#_bookmark6), and one can imagine that writing it down requires several nested loops and complicated transformations. In the next section, we define the algorithm based on rule schemes that need only a very simple control structure.

# Flattening Statecharts by graph transformation

Sect. [2](#_bookmark1) introduced rule schemes which shall here be used for specifying the single transformation steps. They have to be combined with a control program in order to present a complete algorithm for flattening hierarchical statecharts. We do not present the control program in some existing language, but use a rather informal notation as we focus on rule schemes and how they make programming with graph

3 This is an expensive operation. A more efficient solution, e.g., is discussed in [[7](#_bookmark22)].

1. **while** the graph contains *And State* or *Or State* nodes **do**
2. *// remove a bottom level and-state*
3. **if possible** mark bottom level and;
4. **for all matches** and create cross product;
5. **if possible** and create init;
6. **for all matches** and create trans;
7. **as long as possible** and clean up;
8. *// remove all or-states*
9. **as long as possible** or move outgoing trans;
10. **as long as possible** or remove
11. **done**

Fig. 5. Control program for flattening statecharts

*‘1*

*1’*

*‘2 2’*

contains contains contains

has has

*‘3*

contains

And Comp

*3’*

And Comp

x

x

contains contains

Init

*‘4*

Plain

*4’*

init

Plain State

x

x

ori

*‘5*

*5’* ori

State

x

State

And State

Work

And State

Work

Plain State

*1’*

Work

*‘1*

Work

*2’*

And State

has

contains

*‘2*

And State

has

*3’*

And Comp

*‘3*

And Comp

x

contains

x

*4’*

contains

Plain State

*‘4*

Plain State

x

ori

x

*5’*

Plain State

|  |  |  |
| --- | --- | --- |
| Init State |  | Plain State |
| init |

(a) (b)

Fig. 6. Rule schemes of operations *and create cross product* (a) and *and create init* (b).

transformations easier. Fig. [5](#_bookmark9) shows this control structure which calls several oper- ations that make use of a single rule scheme. The control program, its constructs, and the rule schemes are explained in the following. Graph transformations require statecharts to be represented as graphs according to the metamodel in Fig. [3](#_bookmark5).

The control program consists of an outer loop that is repeated as long as there are still hierarchical states. The loop body consists of two parts. The first one (line 3–7) transforms a single and-state which does not contain any hierarchical sub-states (a

so-called *bottom level and-state*) into an or-state. This or-state, together with all

other or-states, is flattened in the second part (line 9 and 10). This procedure has to be repeated because the graph may contain several and-states, and and-states can be nested, i.e., and-states are flattened from the inside out. The operations follow the overview of the algorithm given in Sect. [3](#_bookmark4).

Lines 3–7 operate on an and-state that does not contain any hierarchical states. Rule *mark bottom level and* arbitrarily selects one of them by adding a *Work* node [4](#_bookmark11) to the graph and connecting it with the corresponding *And State* node. Negative

application conditions make sure that this and-state does not contain any and- or or-state. This elementary rule is not shown here for space restrictions. The control **if possible** tries to apply rule *mark bottom level and*, but simply continues if the rule fails, i.e., if there is no bottom level and-state.

4 Please note that the new *Work* node actually violates the metamodel shown in Fig. [3](#_bookmark5). A possible solution would be relaxing the metamodel while transforming the graph or extending the metamodel accordingly.

Fig. [6](#_bookmark10)a shows the rule scheme of the operation *and create cross product* that creates the cross product of the state sets of all and-compartments. This rule scheme can be applied to the and-state that has been selected by the previous rule. The idea of the rule scheme is to match an and-state node together with all of its and-compartments and one contained state of each compartment by a rule instantiation. Fig. [1](#_bookmark2) shows an instance of this rule scheme when binding *x* to 3, i.e., when the and-state has 3 and-compartments.

By applying the rule instance, a new plain state node 5’ is added. Node 5’ becomes a new contained node of the and-state (which later will become an or- state by operation *and clean up*). Moreover, 5’ gets connected by *ori* edges to the

original state nodes of the different and-compartments. That way, 5’ represents the tuple of state nodes from the different and-compartments. However this requires that cardinality variable *x* is always bound to the maximum possible value when instantiating the rule. By applying this rule scheme for any possible match for the maximum value being bound to *x*, this rule scheme creates the complete cross product of the state sets of all and-compartments for the selected and-state. This behavior is specified by the loop control **for all matches**. Please note that this control does not repeat applying the specified rule scheme as long as possible. As the rule scheme’s right-hand side contains the left-hand side, this would specify a non- terminating transformation. The loop control rather has to find for the maximum *x*-value all possible matches and consecutively apply the corresponding rule instance to these matches.

The previous rule instance sets up the cross product of state sets which is the new or-state’s set of contained states. However, the initial pseudo state has not yet been created. This is done by operation *and create init* (Fig. [6](#_bookmark10)b). Its left-hand side is the same as the right-hand side of the previous rule scheme, but with an additional *Init State* node and with *init* edge. Therefore, the instantiation of the left-hand side, matches the tuple of all initial pseudo states, their connected real states and the new tuple state node that has been created by operation *and create cross product*. The original initial pseudo states get removed, and a new initial pseudo state gets created. Again, cardinality variable *x* must be bound to the maximum possible value. The control **if possible** is needed as this rule fails if there is no bottom level and-state.

The next operation has to create transitions between the new states. Fig. [7](#_bookmark13) shows the corresponding rule scheme. The left-hand side represents two tuple states ‘4 and ‘6 that represent the states ‘1 and ‘5, resp. ‘3 and ‘5. Cloned nodes ‘1 and ‘3 together with ‘2 are those states *si* resp. *ti* together with their connecting transition that – as described in Sect. [3](#_bookmark4) – consume an event by firing the corresponding transitions. As a consequence, this rule scheme has to require as an application condition that the transitions that are matched by the instance of the left-hand side must have the same value of their annotation attribute. [5](#_bookmark12) Node ‘5 represents the states of those and-compartments that do not consume the corresponding event. Again,

5 This application condition in Fig. [7](#_bookmark13) uses *card* as a function that obtains the set cardinality of the multiset of all ‘2.annotation values. The action uses *select* which selects a value from a multiset.

Fig. 7. Rule scheme of operation *and create trans*



*‘1*

*‘4*

*7’*

*1’*

*4’*

Plain State

Plain State

ori

Plain State

*‘9*

ori

Work

*9’*

ori

n

Plain State

ori

from

Work

*‘8*

n

*‘7*

contains

from

*8’*

has

And Comp

n

*‘2* Transition

*‘5*

from

*2’* Transition

*5’*

*‘10*

contains annotation

Plain State

n

*10’*

contains annotation

Plain State

*11’*

Transition

annotation

has

And Comp

n

m

has

n

has

And Comp

m

m

contains

ori

m

to

contains

ori

*‘3*

*‘6*

*3’*

*6’*

Plain State

ori

Plain State

Plain State

ori

Plain State

to

n

n

**Application condition:** card(‘2*.*annotation) = 1

**Priority:** *n ≺ m*

**Action:** 11’.annotation := select(‘2.annotation)

And State

And State

to

And Comp

Fig. 8. Rule scheme of operation *or move outgoing trans*



*‘4*

*‘3*

Or State

*1’*

*‘1*

*Real State*

*2’*

from

to

Or State

contains

contains

*3’*

*Real State*

Transition

annotation

*Real 5’* Transition

*State* from annotation

to

p

p

**Action:** 5’.annotation := ‘4.annotation

*Real State*

*‘2*

cardinality variable *n* has to be bound to the maximum possible value such that the application condition is satisfied. Afterwards, *m* has to be bound to the maximum possible value. This requirement is specified by introducing an ordering on the cardinality variables, here *n ≺ m*, telling whose value has to be maximized first. When applying an instance of this rule, a new transition is added between the tuple states 4’ and 6’. This new transition has to get the same annotation value as the one of all matched transitions of the left-hand side. The loop control **for all matches** takes care of creating all possible transitions.

The “internals” of the new or-state have been completely created by line 4–6. However, the remaining and-compartments, the original transitions and state nodes as well as the connection edges together with the *Work* node have to be removed. The *And State* node, moreover, has to be replaced by an *Or State* node. This is performed by the operation *and clean up* in line 7. *And clean up* is actually a set of three straight-forward rule schemes that has been omitted here for space restrictions.

The remaining two operations with loop controls (Fig. [5](#_bookmark9)) flatten all or-states of the graph. This applies in particular to the or-state that just has been created from a bottom level and-state. As described in Sect. [3](#_bookmark4), we have to add copies of transitions leaving an or-state to each of the contained states. This is done by the rule scheme shown in Fig. [8](#_bookmark14) which is applied with maximum *p* value to each match of the left-hand side. Please note that as many copies of transition ‘4 are created as there are states contained in the or-state. All of these new transitions have to get assigned the same annotation values as the match of ‘4 (c.f. the action in Fig. [8](#_bookmark14)).

Fig. 9. Rule scheme of operation *or remove*

*‘1*

*1’*

*Container*

*Container*

m

m

contains

contains

*‘5*

*‘4*

Init State

init

Or State

to

*5’*

Transition

Init State

*2’*

init

*3’*

*Real State*

q

contains

q

*Real State*

to

p

contains

Init State

contains contains

*‘2*

init

*‘3*

*4’*

*Real State*

*Real State*

Transition

p

s

s

Fig. [9](#_bookmark15) shows the final rule scheme which redirects all incoming transitions to the state that has been connected to the or-state’s initial pseudo state, removes this pseudo state and the or-state frame, and uses the containing state – if any – of the previous or-state as the container of the states that have been contained by the or-state. Node ‘5 with cardinality variable *q* represents a potential initial pseudo state that has been connected to the or-state. This transition has to be redirected, too.

Please note that the graph does not contain any *Or State* nodes after processing line 10. However, new *Or State* nodes are created in the next loops if there are still *And State* nodes in the graph.

# Conclusions

In this paper, we have considered the flattening of hierarchical statecharts as a case study for programming based on graph transformation. Space restrictions did only allow to treat simplified statecharts. The missing concepts, like history states, final states, firing conditions, and transition actions as well as enter and exit actions of hierarchical states, can be added in a straight-forward way. A complete specification will be provided in the future.

The case study revealed that instantiation of rule schemes by cloning makes graph transformation more expressive; it simplifies programming by graph transfor- mation as control programs become simpler. We hope that this concept will support graph transformations to be better accepted for programming tasks in the future.

The expansion of variables to graphs proposed in [[4](#_bookmark20)] makes graph transformation still more expressive. But this is beyond the scope of this paper.

Control programs have been presented quite informally in this paper. We are currently combining rule instantiation by cloning with the proposed graph- transformation-based programming language DiaPlan [[2](#_bookmark18)] (which already supports

variable expansion) and its control structures. This will allow to discuss control structures more thoroughly.

# References

1. Andries, M., G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schu¨rr and G. Taentzer, *Graph transformation for specification and programming*, Science of Computer Programming **34** (1999), pp. 1–54.
2. Drewes, F., B. Hoffmann, R. Klein and M. Minas, *Rule-based programming with DiaPlan*, ENTCS **127**

(2004), pp. 15–26, Proc. Int. Workshop on Graph-Based Tools (GraBaTs’04). Rome (Italy), Oct. 2, 2004.

1. Fischer, T., J. Niere, L. Turunski and A. Zu¨ndorf, *Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java*, in: H. Ehrig et al., editors, *Theory and Application of* *Graph Transformation (TAGT’98), Selected Papers*, LNCS 1764 (2000), pp. 296–309.
2. Hoffmann, B., D. Janssens and N. Van Eetvelde, *Cloning and expanding graph transformation rules for refactoring*, in: *Proc. Int. Workshop on Graph and Model Transformation*, Tallinn (Estonia), Sept. 28, 2005, to appear in ENTCS.
3. Karsai, G., A. Agrawal, F. Shi and J. Sprinkle, *On the use of graph transformation in the formal specification of model interpreters*, Journal of Universal Computer Science **9** (2003), pp. 1296–1321.
4. Schu¨rr, A., A. Winter and A. Zu¨ndorf, *The P* rogres *approach: Language and environment*, in:

G. Engels et al., eds., *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, World Scientific, Singapore, 1999, pp. 487–550.

1. Wasowski, A., *Flattening statecharts without explosions*, in: *LCTES ’04: Proc. 2004 ACM SIGPLAN/SIGBED Conf. on Languages, compilers, and tools for embedded systems* (2004), pp. 257– 266.