Nondeterministic Control Structures for Graph Rewriting Systems

Albert Zündorf[1] and Andy Schürr[2]

Lehrstuhl für Informatik III
Aachen University of Technology
Ahornstraße 55, D-5100 Aachen, W-Germany

e-mail: albert@... and andy@...
...@rwthi3.informatik.rwth-aachen.de

Abstract:

The work reported here is part of the IPSEN[3] project whose very goal is the development of an Integrated Project Support ENvironment. Within this project directed, attributed, node- and edge- labeled graphs (diane graphs) are used to model the internal structure of software documents and PROgrammed Graph REwriting SyStems are used to specify the operational behavior of document processing tools like syntax-directed editors, static analyzers, or incremental compilers and interpreters. Recently a very high-level language, named PROGRESS, has been developed to support these activities. This language offers its users a convenient, partly textual, partly graphical concrete syntax and a rich system of consistency checking rules (mainly type compatibility rules) for the underlying calculus of programmed diane-graph rewriting systems.

This paper presents a partly imperative, partly rule-oriented sublanguage of PROGRESS for composing complex graph queries and graph transformations (transactions) out of simple subgraph tests and graph rewriting rules (productions). It also contains a formal definition of this sublanguage by mapping its main control structures onto so-called nondeterministic control flow graphs. We believe that these control structures or at least the underlying flow graph formalism could be incorporated into many graph-/tree-/term- rewriting systems in order to control the nondeterministic selection and application of rewriting rules.

1 *Introduction*

Modern software systems for application areas like office automation and software engineering are usually highly interactive and deal with *complex, structured objects*. The systematic development of these systems requires precise and readable descriptions of their desired behavior. Therefore, many specification languages and methods have been introduced to produce formal descriptions of various aspects of a software system, such as the design of object structures, the effect of operations on objects, or the synchronization of concurrently executed tasks. Many of these languages use special classes of graphs as their underlying data models. Conceptual graphs [Sowa 84], (semantic) data base models [HK 87], petri nets [GJRT 82], or attributed trees [Reps 84] are well-known examples of this kind.

1. Supported by *Deutsche Forschungsgemeinschaft.*
2. Supported by *Stiftung Volkswagenwerk* and the *Bundesministerium für Forschung und Technologie.*
3. cf. [Nagl 80], [EJS 88], [Lew 88]

Within the research project IPSEN a graph grammar based specification method is in use to model the internal structure of software documents and to produce executable specifications of corresponding document processing tools, as e.g. syntax-directed editors, static analyzers, or incremental compilers [ES 85]. The development of such a specification, which is termed '*programmed graph rewriting system*', consists of two closely related subtasks. The first one is to design a graph model for the corresponding complex object structure. The second one is to program object (graph) analyzing and modifying operations by composing sequences of subgraph tests and graph rewriting rules.

Parallel to the continuous evolution of this graph grammar formalism and a graph grammar engineering method, the design of a *graph grammar specification language* is in progress. The outcome of this design process is - as far as we know - the first attempt to combine the advantages of object-oriented database definition languages and attribute grammars (with respect to the description of static data structures) with the advantages of programmed graph rewriting systems (with respect to the definition of nondeterministic graph transformations) within one strongly-typed language (cf. [Zünd 89], [Schürr 91], [Schürr 91a]).

A first version of this language named PROGRESS (for <u>PRO</u>grammed <u>G</u>raph <u>RE</u>writing <u>S</u>y<u>S</u>tems) has been fixed a few months ago, and a prototype of a programming environment for this version of the language is now under development (cf. [NaSc 91]).

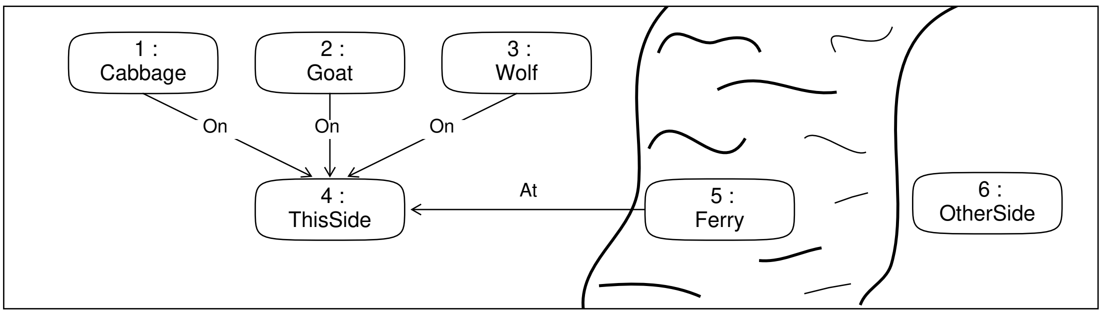
2 The "ferryman's problem" specified with PROGRESS

For an informal introduction to the way of programming the language PROGRESS supports we will use a small specification called "the ferryman's problem". This specification is a precise description as well as a programmed solution of the following problem:

> There are a cabbage, a goat, and a wolf that must be ferried over a river. But the ferry is so small that it is not able to carry more than one of these objects at the same time. Furthermore, at no time one should leave the goat and the cabbage or the wolf and the goat without any supervision on the same river side.

Figure 1 shows the start situation of "the ferryman's problem" in the form of a <u>di</u>rected, <u>a</u>ttributed, <u>n</u>ode- and <u>e</u>dge labeled graph (<u>diane</u>-graph).

<u>Figure</u> <u>**1**</u>: Start situation for "the ferryman's problem"



PROGRESS is a *strongly typed language*. Thus, all kinds of graph elements like nodes, edges, and attributes belong to exactly one type describing their properties. Common properties of different node types may be described in an object-oriented/based hierarchy of node classes.

Figure 2 shows a cutout of the PROGRESS data definition part for the graph of Figure 1. This

Figure 2: PROGRESS description for the graph of Figure 1

```
section GlobalGraphscheme;
(*Description of the example's graph model.*)
    node class THING end;
    edge type On: THING -> RIVER_SIDE;
    node class RIVER_SIDE end;
    node class CARGO is a THING end;
    edge type In: CARGO -> BOAT;
    node class BOAT is a THING end;
    edge type At: BOAT -> RIVER_SIDE;
end; (* GlobalGraphscheme *)
```

```
section SpecialObjectTypes;
(* Specialization of the global graph scheme. *)
(* The concrete object/node types are given. *)
    node type ThisSide: RIVER_SIDE end;
    node type OtherSide: RIVER_SIDE end;
    node type Ferry: BOAT end;
    node type Cabbage: CARGO end;
    node type Goat: CARGO end;
    node type Wolf: CARGO end;
end; (* SpecialObjectTypes *)
```

part of the specification determines the following properties for the graph elements of Figure 1:

- A node of the type Wolf
  - belongs to the class CARGO and possesses all properties defined in that class and its superclass THING, i.e., it may be a source of an edge of the type On or In.
- A node of type Ferry belongs to the class BOAT with the following properties:
  - it may be source of an edge of type On or At and
  - it may be target of an edge of type In.
- A node of type ThisSide, belonging to the class RIVER_SIDE, may be target of an edge of type On or At.
- An edge labeled On
  - demands a source node of a type that belongs (directly or indirectly) to the class THING. (In our example these are the types Wolf, Goat, Cabbage, and Ferry) and
  - demands a target node of a type that belongs (directly or indirectly) to the class RIVER_SIDE (ThisSide and OtherSide).
- An edge labeled In demands a source node class Cargo and a target node class BOAT.
- An edge labeled At leads from a BOAT to a RIVER_SIDE.

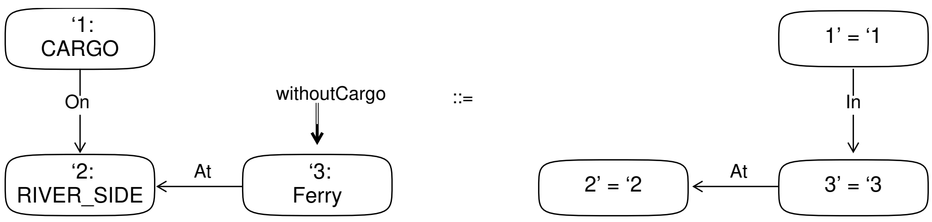
We hope the reader now is able to imagine how the static structure of diane-graphs are described in PROGRESS although our example doesn't include any examples for the definition of attributes and directed attribute dependencies. For a full, detailed description of these features of PROGRESS cf. [Schürr 91], [Zünd 89].

For the definition of basic operations on diane-graphs PROGRESS provides its users with means for the construction of *graph rewriting rules* (Productions and Tests) and *derived relations* on nodes (Pathes and Restrictions).

Figure 3 shows the production LoadFerry. This production applied to the graph of Figure 1

<u>Figure</u> <u>3</u>: The production LoadFerry

<u>production</u> LoadFerry
(Take some cargo on board of the empty ferry.*)*
=



<u>end</u>;

<u>restriction</u> withoutCargo: BOAT = <u>not</u> <u>with</u> <-In- <u>end</u>;

works as follows:

1. First a (partial) subgraph of the current host graph is determined which is isomorphic to the graph on the left hand side of the production. I.e., a subgraph consisting of two nodes with types of the classes CARGO and RIVER-SIDE respectively, one node of the type Ferry, where the CARGO node is connected to the RIVER-SIDE node by an edge with label On and the Ferry is connected to the (same) RIVER-SIDE node by an edge with label At. Additionally, the Ferry node must fulfill the *restriction* withoutCargo, whose definition is also part of Figure 3. It requires the Ferry node not to be the target of an edge of type In. One subgraph of the graph of Figure 1 matching these conditions consists of the nodes 2, 4 and 5.
2. When a matching subgraph of the host graph has been found/selected, this subgraph is removed from the host graph and the graph on the right hand side of the production is inserted. A node inscription like "2' = '2" signals an identical replacement of a node by itself. This means that 2' is the same node as '2 and all its attribute values and all incoming or outgoing edges remain unchanged unless they are explicitly mentioned within the production.

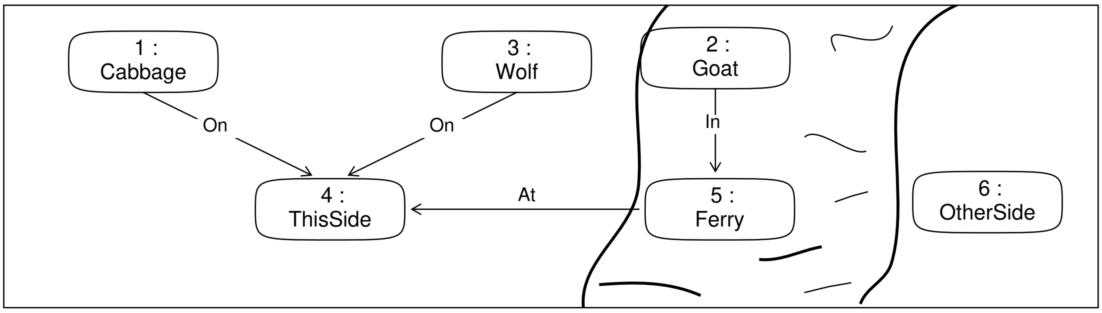
In our example the production LoadFerry replaces all nodes identically and, therefore, neither deletes an old node nor creates a new one. Beside this (identical replacement of three nodes) the production deletes an On edge (between the selected CARGO node and the RIVER_SIDE node) and a new edge with label In is inserted. One possible result graph of an application of LoadFerry to the graph of Figure 1 is shown in Figure 4.

In our running example for this paper we only use productions of this simple form. In general a PROGRESS production may also include[1]:

- ❑ parameters for node types and attribute values,
- ❑ application conditions concerning attribute values,
- ❑ instructions for changing the identical embedding of the new inserted subgraph into the surrounding host graph,

1. For a full description of the features of a production cf. [Schürr 91], [Zünd 89].

Figure 4: One possible result of the application of LoadFerry to the graph of Figure 1



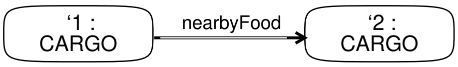
❏ assignments to attributes of new or identically replaced nodes,

❏ assignments to attribute valued return parameters.

At the end of this chapter we will discuss two other sorts of basic operations within PROGRESS, so called (subgraph) tests and pathes. Figure 5 shows the PROGRESS test SupervisionNecessary and two pathes used for its implementation.

Figure 5: The subgraph test SupervisionNecessary

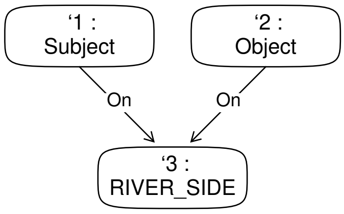
```
test SupervisionNecessary
=   (* If an animal is nearby its *)
    (* preferred food *)
```




```
end;
```

```
path nearbyFood: CARGO -> CARGO
=       nearby<Wolf, Goat>
    or  nearby<Goat, Cabbage>
end;
```

```
path nearby<Subject, Object: CARGO>:
                CARGO -> CARGO
(* Two things are 'nearby' each other if they *)
(* are on the same RIVER_SIDE.*)
=   '1 => '2 in
```




```
end;
```

A *test* may be seen as a special form of a production with identical left- and right-hand side and, therefore, never causes any graph modifications (It is executed just by searching for a subgraph in the current host graph that matches the described conditions). The test SupervisionNecessary searches for two pieces of CARGO that are related through the *path* nearbyFood. This path relates two nodes to each other if they are *either* related by nearby<Wolf, Goat> *or* by nearby<Goat, Cabbage>. And two nodes are related to each other by the path nearby<x, y> if they are connected to the *same* RIVER_SIDE by edges with label On and have types equal to the actual parameters x and y.

We hope, that our explanations of pathes, productions and tests were detailed enough to understand the rest of the paper dedicated to . . .

3 The Control Structures of PROGRESS

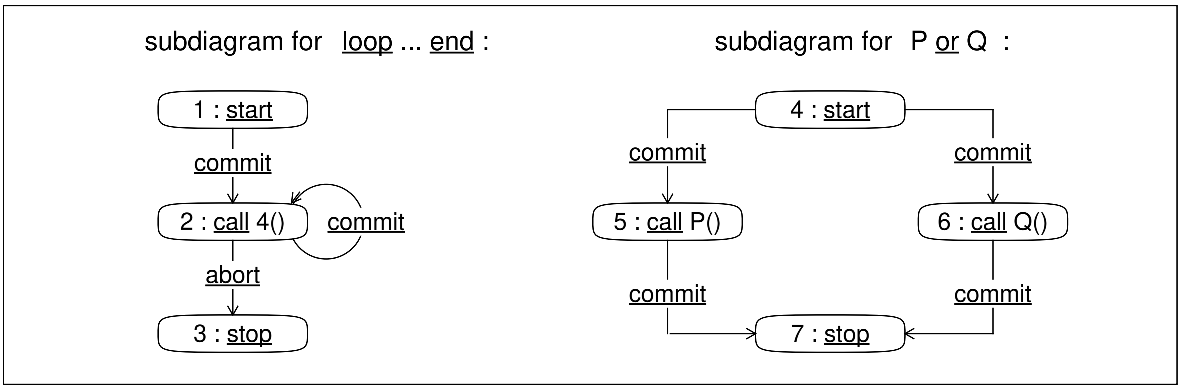
After a short and informal introduction to the definition of 'basic' operations within PROGRESS we now are able to discuss the language's control structures. These control structures guide the selection of applicable graph rewriting rules/productions and may be used to compose complex graph transformation rules, named transactions, out of simple tests and productions. Designing the control structures of PROGRESS we had to take care to preserve the main properties of tests and productions on the level of transactions. These properties are:

- the **atomic** character: The application of a simple production either replaces one subgraph by another one and terminates successfully or fails without causing any graph changes. As a consequence of this fact, the execution of a sequence of graph rewriting rules should also either terminate successfully (if and only if the application of all graph rewriting rules is possible) or fail without any graph changing effects.
- the **boolean** character: Simple productions and complex transactions signal their state of termination to the calling environment. This implicit (boolean) return value determines the selection of the next graph rewriting rule.
- the **nondeterministic** character: Even in simple cases there may be more than one subgraph of the current graph which is isomorphic to the left hand side of the applied production. Executing the production requires a nondeterministic choice of one of these subgraphs. In the example of chapter 2 the production LoadFerry has a nondeterministic character. It contains no directives concerning the selection of an *appropriate* piece of CARGO. Thus the PROGRESS runtime system has to select an *appropriate* one nondeterministically. During the execution of a transaction such nondeterministic choices (if done wrong) may influence the success of future rewrite steps. Our control structures have to deal with this kind of nondeterminism.

3.1 Nondeterministic Control Flow Diagrams

In order to provide the reader with a formal but easily readable definition of PROGRESS' control structures, *nondeterministic control flow diagrams* are the right means. These flow diagrams are very similar to the well-known class of deterministic control flow diagrams and are proper descendants of the control flow graphs in [Nagl 79]. Therefore, an informal description of their semantics should suffice for the purpose of this paper[1].

Figure 6: Example control diagrams



1. A formal definition may be found in [Schürr 91].

A node labeled with a production call (P(), Q(), . . . , 4(), . . .) stands for the execution of the corresponding production, test or another PROGRESS action, or for the execution of a whole subdiagram, whose start node is called. If the call is executed successfully, the execution of the calling diagram proceeds by following one of the outgoing commit edges (nondeterministically chosen). If the call fails, a nondeterministically chosen abort edge has to be followed (if the required type of edge doesn't exist, then another path through the control diagram must be tried). Every subdiagram contains exactly one node labeled start and exactly one stop node. A successful execution path through a subdiagram leads from its start node (through all the called subdiagrams) to its stop node. If there exists no such path the execution of this subdiagram fails, the host graph remains unchanged, and the calling diagram will have to follow an abort edge.

Note: An interpreter which has to find a successful path through such a diagram may have to keep track of its nondeterministic choices. If in one step there exists no outgoing edge of the desired label, the interpreter will have to "backtrack" to the state of its last choice (undoing all changes made to the host graph and the execution environment) and to try another alternative. (Additionally, there may be the problem of nonterminating execution cycles, cf. chapter 3.4.)

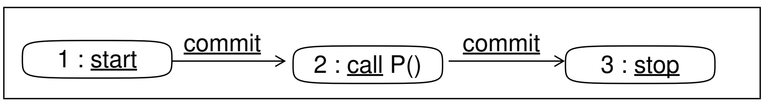
Additionally note, that our control diagrams properly reflect the properties of productions because a subdiagram call

- has an **atomic** character: only those pathes through a called subdiagram ending at its stop node might have any graph changing effects. All modifications done while following pathes into dead-ends are undone within the "backtracking" steps.
- has a **boolean** character: the success of the execution of an action determines if commit or abort edges must be followed.
- has a **nondeterministic** character: any Call node may have an arbitrary number of outgoing commit or abort edges. Thus, there may be several successful execution pathes through a subdiagram performing different graph transformations. A successful path through the whole control diagram will be selected nondeterministically. A deterministic working interpreter for PROGRESS will have to use mechanisms like backtracking to simulate this behavior (and for escaping out of dead-ends).

3.2 skip-, fail-, not- *and* def-*statement*

This chapter introduces some "pseudo" actions and simple operations, that we will need later on. It also introduces a control diagram for the call of a single PROGRESS action[1].

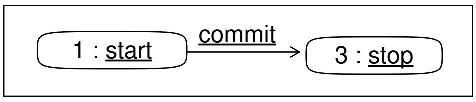
Figure 7: Control diagram for the call of a single production or test P():



An execution of the above control diagram means: apply the production/test P() to the current host graph. If there are several matching subgraphs choose one which allows the calling diagram to succeed (if possible). If there is no matching subgraph, the host graph will remain unchanged and the execution of the subdiagram fails.

1. In the following PROGRESS action means a production, a test, or a compound action built by the proposed control structures.

Figure 8: Control diagram for skip:

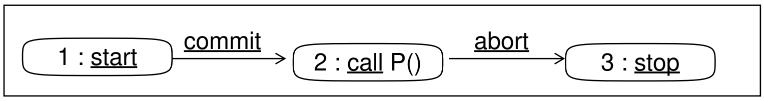


Always successful "pseudo" actions. The host graph remains unchanged. Follow a commit edge in the calling diagram.

Figure 9: Control diagram for fail:

Always failing "pseudo" action. The host graph remains unchanged. Follow an abort edge in the calling diagram.

Figure 10: Control diagram for not P():

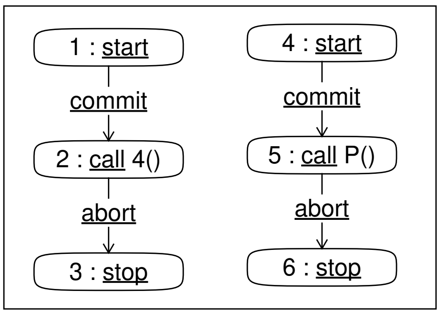


If it is possible to execute P() then the subdiagram for not P() has no legal execution path. Thus the calling diagram has to follow an abort edge. If P() is not executable, then the subdiagram for not P() succeeds and the calling diagram will follow a commit edge.

Note, that not P() never will have a graph changing effect. If P succeeds, then not P fails and the host graph remains unchanged. If P fails, it did not change the graph, thus not P doesn't too. Nevertheless, not LoadFerry is NOT semantically equivalent to skip. If it is possible to apply LoadFerry to the current graph, then not LoadFerry **fails**, while skip is always successful. As the user may be utterly confused by using not for an action that normally modifies the graph, this is forbidden in PROGRESS. The operator not may only be used with *effectless actions* like tests, skip, fail, or composite actions consisting of effectless subactions.

In order to allow the user to apply not to an action with graph changing effects we introduce the def-statement in Figure 11.

Figure 11: Control diagram for def P():



If P is executable, then the right subdiagram (of the figure on the left) fails (the graph remains unchanged) and thus the left subdiagram succeeds. If P fails, then the right subdiagram reaches its stop node by the abort edge and the left subdiagram fails leaving the current graph unchanged. To sum up, the diagram that calls def (P) will proceed, as if it had called P, but the graph modifications of P are NOT performed.

3.3 &-, or-, and and-statement

The first control structure we want to propose is the simple sequence of two or more PROGRESS actions, the &-statement[1].

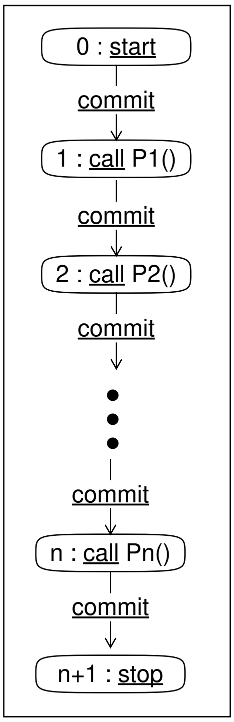
1. Say concatenation statement.

Example: LoadFerry & not SupervisionNecessary & FerryOver & UnloadFerry

General form: P1 & P2 & . . . & Pn

Control diagram: Figure 12

Figure 12: Control diagram for a &-statement:



A legal path through the control diagram of a &-statement demands that all of its subactions are executable. The called actions are executed nondeterministically. Theoretically, only those alternatives are considered which allow the subsequent actions to succeed (in practice, backtracking will be necessary). In our example above the action LoadFerry has several matching subgraphs in the graph of Figure 1. It might put a Cabbage, a Goat, or a Wolf into the ferry. Depending on that choice, the test not SupervisionNecessary succeeds or fails. In the current situation, the Goat should be selected in order to be able to complete the path through our control diagram.

For the sequence

LoadFerry & LoadFerry & FerryOver

there exists no successful execution path, because only an unloaded ferry may be loaded. Thus the second call to LoadFerry will fail no matter what piece of CARGO is loaded by the first one. According to the dynamic semantics of control flow diagrams this means that no part of the sequence is executed (resp. everything is undone) and the calling diagram has to continue with an abort edge.

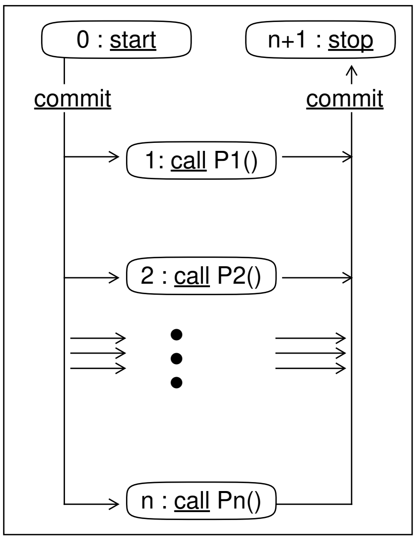
The or-statement combines a number of alternative actions. One of these actions must be chosen nondeterministically and then executed.

Example: LoadFerry or skip

General form: P1 or P2 or . . . or Pn

Control diagram: Figure 13

Figure 13: Control diagram for an or-statement:



If none of the actions Pi is executable, then the subdiagram for the or-statement has no legal execution path. If several alternatives are possible, then that will be chosen which allows the calling diagram to succeed.

Thus, if there is a number of alternatives like “LoadFerry” or “do nothing” a PROGRESS interpreter has to select one that guarantees a successful termination of the whole execution process.

While the &-statement determines the order for the execution of its subactions, the following and-statement only demands that all its subactions have to be executed (in any order).

Example: not SupervisionNecessary and FerryOver

General form: P1 and P2 and . . . and Pn

Semantic equivalence: (P1 & P2 & . . . & Pn) or (P2 & P1 & P3 & . . . & Pn) or (. . .) . . .

Again PROGRESS' dynamic semantics demand that sequence of basic actions to be chosen which allows the calling diagram to succeed.

3.4 choose- and loop-statement

In the same way as the &-statement is the 'deterministic' variant of the and-statement, the following choose-statement may be considered as a deterministic variant of the or-statement.

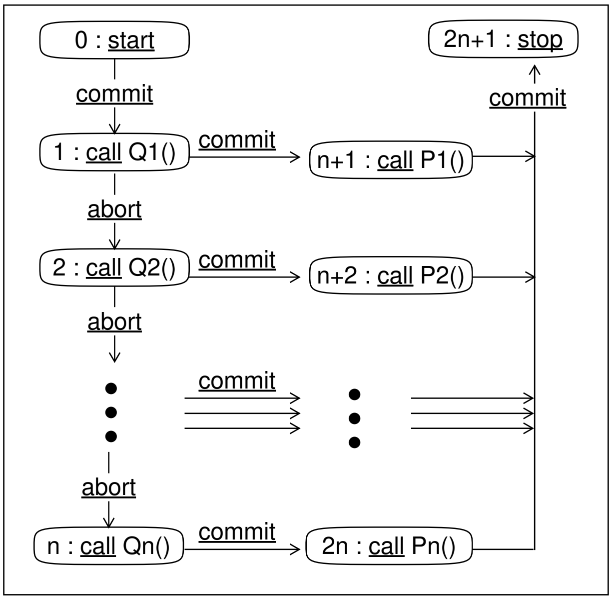
Example: choose ReducePopulation else skip end

General form:
```
choose when Q1 then P1
   else when Q2 then P2
   . . .
   else when Qn then Pn
end
```

Semantic equivalence: choose . . . else P else . . . end
is a abbreviation for:
choose . . . else when P then P else . . . end

Control diagram: Figure 14

Figure 14: Control diagram for a choose-statement:



In general, a choose-statement consists of a list of *guarded* alternatives. The guards are examined subsequently until a succeeding one is found. That alternative will be chosen and then be executed. If there is no executable guard or if the chosen call isn't executable then the diagram for the choose-statement has no legal execution path.

We request that guards may only be composed of effectless actions like tests (for similar reasons as mentioned by the discussion of the not operator). Thus the action performed by the choose-statement is that which is performed by the action of the chosen alternative.

Executing our example above means first trying to perform ReducePopulation (with an implicit guard def(ReducePopulation)). Only if this action fails, then the always succeeding alternative skip will be selected.

The last control structure we want to propose is the iteration, the loop-statement.

Example:

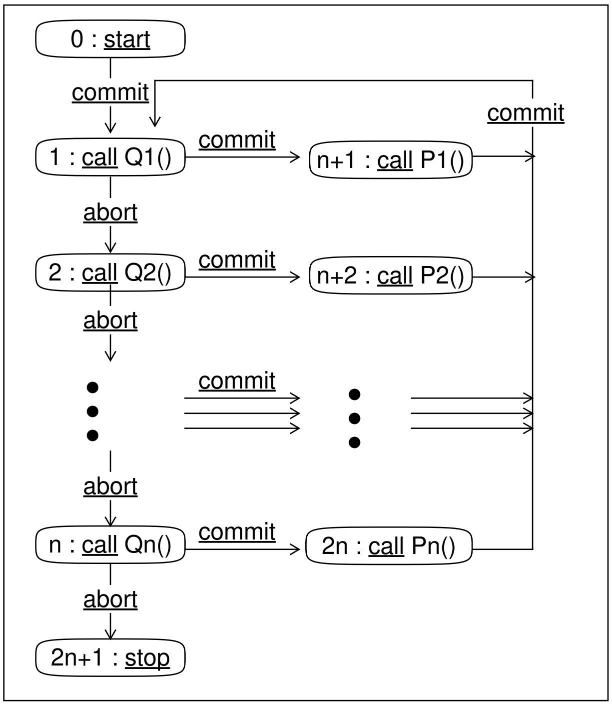
```
loop when not ProblemSolved then
          LoadFerry
       or (not SupervisionNecessary & FerryOver )
       or UnLoadFerry
end (* loop *)
```

General form:

```
loop    when Q1 then P1
   else when Q2 then P2
   . . .
   else when Qn then Pn
end
```

Control diagram: Figure 15

Figure 15: Control diagram for a loop-statement:



Like a choose-statement a loop-statement consists of a sequence of guarded alternatives. The body of a loop is executed like a choose-statement. The first guard which succeeds determines the action which is executed by the next iteration step. If no guard succeeds, the loop terminates successfully.
Note: If a selected action Pi fails, there is no legal continuation for the execution path through the control diagram. In this case backtracking starts (undoing graph changes) and the PROGRESS interpreter first tries to revise nondeterministic choices during the execution of earlier iteration steps and then at last aborts the execution of the whole loop (undoing all graph changes).

Note: A deterministically working interpreter using backtracking might have a serious termination problem with the example loop-statement above. The interpreter might loop forever just executing FerryOver in every iteration step. Our semantics for the loop-statement defines only how a successful execution looks like. It doesn't determine (like in Prolog) how this execution path has to be found by an deterministically working interpreter. In this case only an interpreter with a fairly (randomly) working algorithm for the selection of possible alternatives will be able to terminate the above loop (but probably not in the minimal number of iteration steps[1]). In general **the user himself has to take care of the termination of the loop**. An always terminating solution for the "ferryman's problem" may be found in Figure 16.

1. There is no way to arrange the order of the three 'ferry' actions so that an deterministically choosing interpreter (like a Prolog interpreter which takes the 'first' possible action) terminates the loop. cf. chapter 4.

Figure 16: A solution for the Ferryman's problem:

```
use    StepsLeft : Integer := 1000;
              (*Estimated upper bound for steps needed *)
in  loop when not ProblemSolved and StepsLeft > 0 then
              ( LoadFerry or FerryOver or UnloadFerry)
          &  (      choose ReducePopulation else skip end
               and StepsLeft := StepsLeft  '-' 1 )
    end (* loop *)
```

This loop will find a solution for the Ferryman's problem that needs less (equal) than one thousand steps (if there exists one). For a PROGRESS specification that finds a solution with a minimal number of steps cf. Appendix B.

3.5 Overview

The following table contains a short overview of the control structures of PROGRESS:

	deterministic	nondeterministic
sequence	&	and
branch	choose (optional guards)	or
iteration (tail recursion)	loop (optional guards)	loop (. . . or . . .)

In general, the control structures a language needs are the sequence, the (conditional) branch, and the iteration. PROGRESS has all these control structures in a deterministic and a nondeterministic version[1], thus reaching a smooth integration of the imperative and rule based programming paradigm. For the deterministic version of our branch-statement (and of our iteration) we introduced guarded alternatives. We did not introduce guards for our nondeterministic or-statement like those of [Dij 75]. We did not so, because this would not enhance the expressiveness of our language. A statement like

. . . or when FerryIsFull then UnloadFerry or . . .

would be semantically equivalent to

. . . or FerryIsFull & UnloadFerry or . . .

If in this example the guarded operation UnloadFerry fails the nondeterministic semantics of our or-statement will **not abort** but just try another alternative in order to find a successful execution for the whole statement. In our choose-statement a guarded operation that fails causes the whole statement to fail, because the order of examination of alternatives is prescribed.

In the following chapter 4 we will compare our control structures with those of several other programming languages based on (graph) (rewriting) rules or production systems.

1. Using a single or-statement as body of a loop-statement, we yield a nondeterministic choice of alternatives within our iteration statement.

4 *Related Work*

There are many approaches to define control structures for production systems, (graph) grammars, and (graph) rewriting systems. An exhaustive comparison of PROGRESS with such languages may be found in [Schürr 91], [Zünd 89]. Due to lack of space we have to focus our interest here onto *two typical representatives* of the whole class of (programmed) rule based systems. These are the well-known language Prolog and the Programmed Graph Grammars of [Gött 88].

[Gött 88] describes a graph rewriting system whose basic operations are very similar to productions of PROGRESS (without complex application conditions) and proposes three control structures for his language: wapp, sapp, and capp. sapp stands for Sequential APPlication. The sapp-statement's subactions will be executed one after the other, skipping all inexecutable subactions. Consequently, the execution of the whole sapp construct succeeds as long as at least one of its subactions succeeds. A capp-statement (Case APPlicable) consists of an sequence of alternative subactions. The first subaction that succeeds terminates the capp. The wapp-statement (While APPlicable) executes its body until it fails. To sum up, the control structures of [Gött 88] do respect the *boolean character* of graph rewriting rules but they neither support the definition of *atomic sequences* of graph rewriting rules (already aborting without any graph changing effects in the case of failure of just a single subaction) nor do they pay regard to the *nondeterministic character* of graph rewrite steps (by backtracking out of dead-ends and thus preventing “wrong” nondeterministic subgraph selections).

Now we will discuss the “control structures” of Prolog. We have chosen this language, because its comparison with PROGRESS shows the benefits of our control structures from another point of view. Figure 17 shows a solution for the Ferryman's problem implemented in Prolog,

Figure 17: Ferryman's problem in Prolog:

```
solveFMP(StepsLeft, StartGraph, TargetGraph)   :- problemSolved(StartGraph, TargetGraph) .
solveFMP(StepsLeft, StartGraph, TargetGraph)
                                          :-  StepsLeft > 0, StepsLeft2 = StepsLeft - 1,
                                              oneStep(StartGraph, StepGraph),
                                              tryRemovePopulation(StepGraph, RemoveGraph),
                                              solveFMP( StepsLeft2, RemoveGraph, TargetGraph) .

oneStep(OldGraph, NewGraph) :- loadFerry(OldGraph, NewGraph) .
oneStep(OldGraph, NewGraph) :- ferryOver(OldGraph, NewGraph) .
oneStep(OldGraph, NewGraph) :- unloadFerry(OldGraph, NewGraph) .

tryRemovePopulation(InG, OutG)  :- reducePopulation(InG, OutG), ! .
tryRemovePopulation(InG, InG) .
```

which is analogous to our implementation given in Figure 16[1]. The PROGRESS loop has been translated using tail recursion within the definition of the predicate solveFMP[2]. The predicate oneStep is (semantically) equivalent to the following deterministic branch statement:

choose LoadFerry else FerryOver else UnloadFerry end

1. Skipping the definition of predicates corresponding to basic PROGRESS tests and productions.
2. The parameters StartGraph and TargetGraph are implicit in PROGRESS.

Note, that the PROGRESS specification in Figure 16 makes use of the nondeterministic or- and and-statements. Thus we were not forced to lay down any arbitrarily chosen precedence order for the selection of executable subactions as the Prolog implementation necessarily does[1]. The predicate tryRemovePopulation corresponds to the choose-statement of Figure 16. Note the cut (!) at the end of the first rule for this predicate. A guarded alternative in PROGRESS:

choose . . . when Q1 then P1 . . . end

corresponds to the following Prolog rule:

. . . :- Q1, !, P1 .

If the guard succeeds our semantic rules demand that the corresponding action is executed. If this action fails, then the whole enclosing control structure fails and backtracking to other alternatives is forbidden. To enforce this behavior in Prolog we have to use the cut. (An unguarded alternative is semantically equivalent to one with its action as a guard.)

As one can see, there is a close correspondence between the control structures proposed in this paper and the execution mechanisms for Prolog. We think that our control structures are in some way superior, because we do not only provide the deterministic 'and' and 'or' (like Prolog), but also nondeterministic 'and' and 'or'-statements. And last but not least the unification of (treelike) terms has been replaced by the construction of subgraph isomorphisms and PROGRESS' backtracking mechanism includes undoing of changes to the underlying graph database whereas Prolog does not support undoing of assert and retract of facts during backtracking[2].

5 *Conclusions, Future Work*

We think that the control structures we have proposed meet the requirements of chapter 3.1. The atomic, boolean, and nondeterministic character of the basic operations of PROGRESS are transferred in a consistent manner to its control structures. And the practical experiences made with PROGRESS and its control structures, e.g. in [Westf 91], and several other specifications are very satisfying.

Within the IPSEN[3] project PROGRESS currently is used to specify the internal behavior and implementation of several software tools. There exists a first prototype of a programming environment for PROGRESS itself. It now includes a syntax-directed editor and an incrementally working analyzing tool performing a large number of checks during editing.

There are two main directions for further development of PROGRESS:

- Development of the language PROGRESS: The main problem that remains is to incorporate a module concept into PROGRESS in order to be able to build large, complex specifications by composing simple, abstract, reusable subspecifications. This problem has to be seen together with the introduction of a concept of hierarchical graphs.

---

1. Note, that in Prolog a mechanism like the StepsLeft counter is necessary to guarantee the termination. (Another common mechanism to guarantee termination in Prolog is to keep track of the whole execution history. Then, after each step the new state is compared with all existing states. If it already has been 'visited' the current execution path fails, cf. [SteSha 86]). In PROGRESS a fairly working interpreter could find a solution without such a mechanism because we have used the nondeterministic or-statement, cf. chapter 3.4.
2. We will have to pay the prize when we are building an interpreter for PROGRESS.
3. Interactive Programming Support ENvironment cf. [Nagl 80], [EJS 88], [Lew 88]

- ❏ Development of software tools for PROGRESS: After the implementation of the PROGRESS editor our main activities now are directed towards the final goal of directly executing PROGRESS specifications and of using PROGRESS specifications as a base for the generation of semantically equivalent source code written in a conventional (imperative) programming language (Modula-2, C(++), . . .). Some basic implementation work is already done. Hopefully, at the end of this year an interpreter and a compiler for the graph scheme definition sublanguage will be available, including the (incremental) evaluation of derived attributes and path definitions. The development of an interpreter for PROGRESS supporting the whole language is under work (but will last longer).

As a result of these activities we hope that the family of PROGRESS users as well as the number of its applications will increase, thus getting more feedback and practical experience for future work.

References

[Dij 75] Dijkstra, E.W.: Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, CACM Vol. 18, No. 8, acm Press (1975), S. 453-457

[EKR 91] Ehrig H., Kreowski H.-J., Rozenberg G. (eds.): *Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science*; (published in) LNCS, Springer Verlag (1991)

[Eng 86] Engels G.: *Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung*; Ph.D. Thesis, VDI-Verlag (1986)

[EJS 88] Engels G., Janning Th., Schäfer W.: *A Highly Integrated Tool Set for Program Development Support*; Proc. ACM SIGSMALL Conf., acm Press(1988), pp.1-10

[ES 85] Engels G., Schäfer W.: *Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment*; in Ehrig et al.(Eds.): Proc. TAPSOFT'85, LNCS 186; Berlin: Springer-Verlag, pp. 179-193

[HK 87] Hull R., King R.: *Semantic Database Modeling: Survey, Applications, and Research Issues*; in: ACM Computing Surveys, vol. 19, No. 3, acm Press (1987), pp. 201-260

[GJRT 82] Genrich H.J., Janssens D., Rozenberg G., Thiagarajan P.S.: P*etri nets and their Relation to Graph Grammars*; in [ENR 82], pp. 115-142

[Gött 88] Göttler H.: *Graphgrammatiken in der Softwaretechnik*; IFB 178, Springer Verlag (1988)

[Lew 88] Lewerentz C.: *Extended Programming in the Large in a Software Development Environment*; in: Proc. 3rd. Int. ACM SIGPLAN/SIGSOFT Symp. on Practical Software Engineering Environments, SIGSOFT Notes, Vol. 13, No. 5, acm Press (1988), pp. 173-182

[Nagl 79] Nagl M.: *Graph-Grammatiken*; Vieweg Verlag (1979)

[NaSc 91] Nagl M., Schürr A.: *A Specification Environment for Graph Grammars*; in [EKR 91]

[Nagl 80] Nagl M.: *An Incremental Compiler as Part of a System for Software Production*; IFB 25, Springer Verlag (1980), pp. 29-44

[Reps 84] Reps T.: *Generating Language-Based Environments*; Ph.D. Thesis, MIT Press (1984)

[Schürr 91] Schürr A.: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*; Ph.D. Thesis, RWTH Aachen (1991)

[Schürr 91a] Schürr A.: *PROGRESS: A VHL-Language Based on Graph Grammars*; in: [EKR 91]

[Sowa 84] Sowa J.F.: *Conceptual Structures: Information Processing in Minds and Machines*; Addison-Wesley (1984)

[SteSha 86] Sterling, L., Shapiro, E.: The Art of Prolog; The MIT Press, Cambridge, Massachusetts, London, England (1986).

[Westf 91] Westfechtel B.: *Revisionskontrolle in einer integrierten Softwareentwicklungsumgebung*; Ph.D. Thesis, RWTH Aachen (1991)

[Zünd 89] Zündorf A.: *Kontrollstrukturen für die Spezifikationssprache PROGRESS*; Master Thesis, RWTH Aachen (1989)

Appendix A: Cutout of the PROGRESS syntax

Cutout of PROGRESS' EBNF:

This is a cutout of PROGRESS' EBNF. It describes the concrete syntax of the control structures proposed within this paper. This EBNF is part of some input files that are used to generate a syntax-directed editor (for the language PROGRESS) within the PROGRESS programming support environment, cf. [Schürr 91]

Overall structure of an PROGRESS specification:

(1) Specification ::= "spec" DeclSpecId ["; "]
[OptImportList]
[OptDeclList]
"end" "." ;

(2) OptImportList ::= . . .

(3) OptDeclList ::= { Decl } Decl ;

(4) Decl ::= Section | NodeClassDecl | NodeTypeDecl | EdgeTypeDecl | PathDecl | RestrictionDecl | TestDecl | ProductionDecl | QueryDecl | TransactionDecl | FunctionDecl;

(5) Section ::= "section" SectionId [";"]
[OptDeclList]
"end" ";" ;

Declaring the graph scheme:

(6) NodeClassDecl ::= "node class" DeclNodeClassId [OptSuperClassIdList] [";"]
[OptAttDeclList] [OptAttRuleList]
"end" ";" ;

(7) OptSuperClassIdList ::= "is a" { ApplNodeClassId "," } ApplNodeClassId ;

(8) NodeTypeDecl ::= "node type" DeclNodeTypeId ":" ApplNodeClassId [";"]
[OptAttRuleList]
"end" ";"

(9) EdgeTypeDecl ::= "edge type" DeclEdgeTypeId ":" RelType ";" ; *(* cf. rule (136). *)*

Declaration of attributes and automatic attribute derivation rules:

(10) OptAttDeclList ::= . . .

(11) OptAttRuleList ::= . . .

Declaration of pathes and restrictions:

(12) PathDecl ::= "path" DeclPathId FormParPart ":" RelType "="
PathBody
"end" ";" ; *(* cf. rule (136). *)*

(13) PathBody ::= . . .

Declaration of tests and productions:

(14) TestDecl ::= "test" DeclTestId FormParPart "="
TestBody
"end" ";" ;

(15) TestBody ::= [OptLeftSideList] [OptAttCondList] [OptReturnList] ;

(16) ProductionDecl ::= "<u>production</u>" DeclProductionId FormParPart "="
ProductionBody
"<u>end</u>" ";" ;

(17) ProductionBody ::= GraphPart [OptAttCondList] [OptEmbList]
[OptAttTransferList] [OptReturnList] ;

(18) GraphPart ::= OptLeftSideList "::=" OptRightSideList ;

Description of a subgraph on the left-hand side of a rule:

(19) . . .

Declaration of queries and transactions:

(20) TransactionDecl ::= "<u>transaction</u>" DeclTransactionId FormParPart "="
StatExpr
"<u>end</u>" ";" ;

(21) QueryDecl ::= "<u>query</u>" DeclQueryId FormParPart "="
StatExpr
"<u>end</u>" ";" ;

Concrete syntax of control structures:

(22) StatExpr ::= ConcStatList | OrStatList | AndStatList | StatTerm ;

(23) ConcStatList ::= { StatTerm "&" } StatTerm ;

(24) OrStatList ::= { StatTerm "<u>or</u>" } StatTerm ;

(25) AndStatList ::= { StatTerm "<u>and</u>" } StatTerm ;

(26) StatTerm ::= CallStat | UseStat | AssignStat | ValidStat | DefStat | NotStat |
SkipStat | FailStat | BracketStat | BranchStatList | LoopStatList ;

(27) CallStat ::= ApplActionId ActParPart ;

(28) UseStat ::= "<u>use</u>" LocalsList "<u>in</u>" StatExpr "<u>end</u>" ;

(29) AssignStat ::= ApplVarOutParId ":=" Expression ;

(30) ValidStat ::= "<u>valid</u>" Term ;

(31) DefStat ::= "<u>def</u>" "(" StatExpr ")" ;

(32) NotStat ::= "<u>not</u>" StatTerm ;

(33) SkipStat ::= "<u>skip</u>" ;

(34) FailStat ::= "<u>fail</u>" ;

(35) BracketStat ::= "(" StatExpr ")" ;

(36) BranchStatList ::= "<u>choose</u>" { CondStat "<u>else</u>" } CondStat "<u>end</u>" ;

(37) LoopStatList ::= "<u>loop</u>" { CondStat "<u>else</u>" } CondStat "<u>end</u>" ;

(38) CondStat ::= GuardStat | StatExpr ;

(39) GuardStat ::= "<u>when</u>" StatExpr "<u>then</u>" StatExpr ;

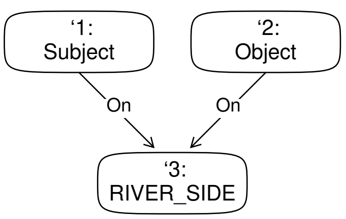
Definition of functions for attribute expressions:

(40) . . .

Appendix B: Specification of the Ferry Man's Problem

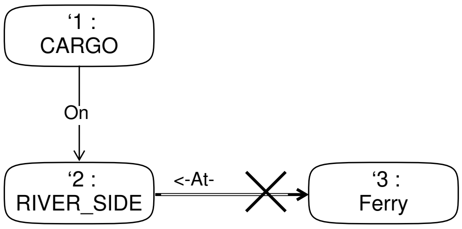
```
spec FerryMansProblem;
   (* An example PROGRESS specification for the following problem:                  *)
   (* A cabbage, a goat, and a wolf have to be ferried over a river. But there are several   *)
   (* restrictions to be observed:                                                  *)
   (* (1)   The ferry is not able to carry more than one piece of cargo at the same time.   *)
   (* (2)   The goat and the cabbage or the wolf and the goat must never be left alone on   *)
   (*       any side of the river (you can imagine why).                             *)
   type import BOOLEAN, Cardinal;
   function import
   (* Predefined functions for the types BOOLEAN and Cardinal. *)
      'not':  (BOOLEAN)              -> BOOLEAN,
      inc:    (Cardinal)             -> Cardinal,
      '<=':   (Cardinal, Cardinal)   -> BOOLEAN,
   transaction SolveFerryMansProblem
      (*  The problem is solved by examining all legal sequences of basic actions        *)
      (*  in order to find the shortest one which leads to the desired solution.         *)
      =
      use  NoOfActions: Cardinal := 11 (* There is no solution with less than 11 actions     *)
      in
         CreateProblem
            (* Creates the initial situation: wolf, goat, cabbage, and ferry on this river side.  *)
       & loop   (* Determines the minimal number of actions that is necessary.             *)
            when not def( SolutionWith_N_Actions(NoOfActions) )
            then  NoOfActions := inc(NoOfActions)
         end
       & SolutionWith_N_Actions(NoOfActions)
         (* Again computing a shortest solution (there are two). For demonstration purpose only. *)
       & RemoveSolution (* If all cargo reached the other river side, the graph is cleaned up. *)
      end
   end;  (* SolveFerryMansProblem *)
   transaction SolutionWith_N_Actions (NoOfActions: Cardinal)
   =  (* Search for a solution using maximal NoOfActions graph rewritings! *)
      use  Steps: Cardinal := 0
      in  loop
            when not ProblemSolved (* Not all cargo reached the other river side.          *)
            then   Steps := inc(Steps)
               &  valid Steps '<=' NoOfActions (* An additional action is still permitted.  *)
               &  (   LoadFerry       (* This action requires, that the ferry is empty and   *)
                                         (* there is some cargo on the ferry's river side        *)
                   or  (not SupervisionNecessary & FerryOver)
                                         (* If situation allows, change the river side           *)
                   or  UnloadFerry)   (* Possible if there is something in the ferry.          *)
         end
      end
   end;  (* SolutionWith_N_Actions *)
```

```
section GlobalGraphscheme;  (* Description of the used graph model.                  *)
    node class THING end;
    edge type On: THING -> RIVER_SIDE;
    node class RIVER_SIDE end;
    node class CARGO is a THING end;
    edge type In: CARGO -> BOAT;
    node class BOAT is a THING end;
    edge type At: BOAT -> RIVER_SIDE;
end; (* GlobalGraphscheme *)

section SpecialObjectTypes;
    (* Specialization of the global graph scheme. The concrete object/node types are given *)
    node type ThisSide: RIVER_SIDE end;
    node type OtherSide: RIVER_SIDE end;
    node type Ferry: BOAT end;
    node type Cabbage: CARGO end;
    node type Goat: CARGO end;
    node type Wolf: CARGO end;
end; (* SpecialObjectTypes *)

section DerivedRelations; (* This section defines some derived relations and restrictions *)
    path myCargo: BOAT -> CARGO =  <-In- end;
    path nearbyFood: CARGO -> CARGO
    =
        nearby<Wolf, Goat> or nearby<Goat, Cabbage>
    end;
    path nearby<Subject, Object: CARGO>: CARGO -> CARGO
        (* Two pieces of CARGO are 'nearby' each other, if they are on the same RIVER_SIDE*)
    =   '1 => '2 in
```




```
    end;
```

restriction withoutSupervision: CARGO *(* There is no supervision if the ferryman is absent. *)*
= ‘1 in



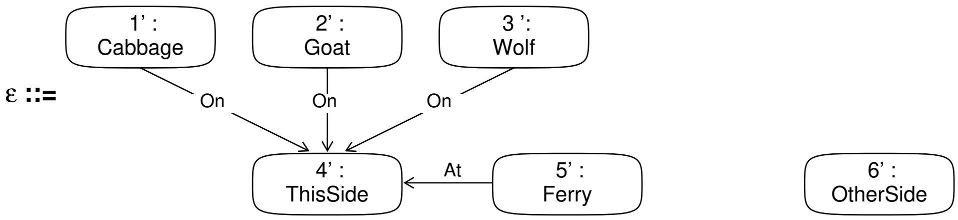
end;

restriction withoutCargo: BOAT = not with <-In- end;

end; *(* DerivedRelations *)*

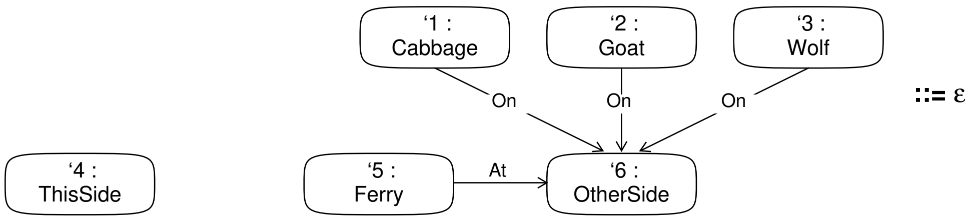
section BasicOperations; *(* Definition of the basic graph rewriting operations. *)*

production CreateProblem *(* Creates the start situation, everything is on ThisSide. *)*
=



end;

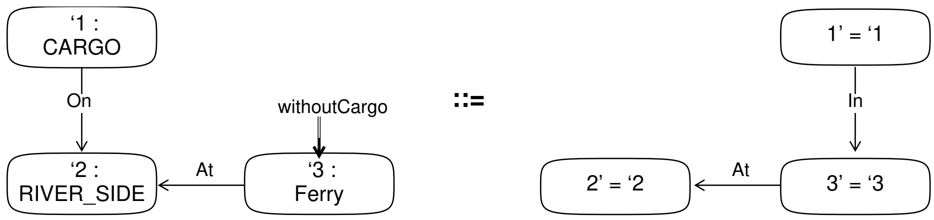
production RemoveSolution *(* If everything is ferried over, the graph will be cleaned up.*)*
=



end;

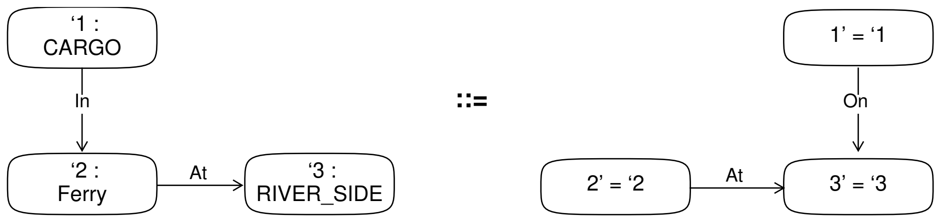
query ProblemSolved *(* Tests, whether the correct final situation has been reached. *)*
=
def(RemoveSolution)
end;

production LoadFerry
(Take some cargo on board of the empty ferry.*)*
=



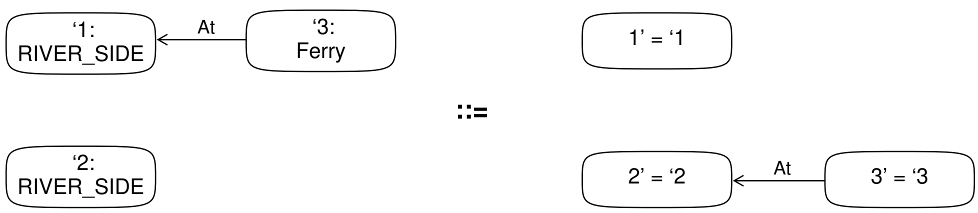
end;

production UnloadFerry
(Put the ferry's cargo ashore.*)*
=



end;

production FerryOver
(The ferry and its load are crossing the river. *)*
=

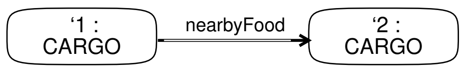


end;

production ReducePopulation
(That should not happen! Probably the result of an erroneous decision. *)*
=

end;

test SupervisionNecessary
= *(* If an animal is nearby its preferred food. *)*



end;

end; *(* BasicOperations *)*

end. *(* FerryMansProblem *)*