Implementation of the imperative / rule based language PROGRES

Albert Zündorf[1]

Department of computer science III
Aachen University of Technology
Ahornstraße 55, 5100 Aachen, Germany

e-mail: albert@rwthi3.informatik.rwth-aachen.de
phone: ++49/+241/80-21312 fax: ++49/+241/80-21329

Abstract

The work reported here is part of the PROGRES (**PRO**grammed **G**raph **R**ewriting **S**ystems) project. PROGRES is a very high level multi paradigm language for the specification of complex structured data types and their operations. The data structures are modelled as **di**rected, **a**ttributed, **n**ode and **e**dge labelled graphs (diane graphs). The basic programming constructs of PROGRES are graph rewriting rules (productions and tests) and derived relations on nodes (paths and restrictions). These basic operations may be combined to build partly imperative, partly rule based, complex graph transformations by special control structures which regard the nondeterministic nature of graph rewriting rules. PROGRES offers its users a convenient, partly textual, partly graphical concrete syntax and a rich system of consistency checking rules for the underlying calculus of programmed diane-graph rewriting systems.

This paper presents the key techniques used for the execution of PROGRES programs. We will discuss a simple approach to control flow backtracking which is done in a conventional imperative programming language by using a certain programming style. Additionally, the semantics of PROGRES demands that we are able to restore arbitrary previous graph states during backtracking. This adds a new quality to the problem of backtracking which is addressed using a nonstandard database management system and its undo / redo mechanism. We believe that the techniques presented here are not specific for the language PROGRES but are useful for the implementation of a whole new family of multiple paradigm languages.

1 *Introduction*

Today's most wide spread programming languages normally support only one programming paradigm. The whole Pascal family of languages ranging from Algol 60 to modern languages like Ada or Modula-2 or object oriented languages like Eiffel, C++, Oberon, or Modula-3 mainly support the imperative programming paradigm. This paradigm is well-suited for algorithmic problem solving. In the artificial intelligence world, rule based and logic programming languages like Ops-5 and Prolog (cf. [SteSha 86]) are used. In these languages, a problem is

1. Supported by *Deutsche Forschungsgemeinschaft.*

described by a set of rules. A solution for the problem is found by automatically selecting and applying appropriate rules.

Multi paradigm languages try to combine the advantages of several programming families. The most important multiple paradigm languages for our approach are *Leda* and *btC*.

Leda, cf. [Budd 91], combines the *imperative* and the *relational* programming paradigm. Relational programming here means querying a database in a Prolog like manner. As in Prolog, implicit knowledge is derived from the explicit knowledge modelled by facts and clauses using SLD resolution and unification. On success, the result of unification is passed to the calling imperative parts. In order to get all possible variable bindings, one may force a failure and the relational part will perform backtracking in order to find another solution. It is possible to use imperative parts in relations. But side effects like assignments to global variables and manipulation of global data structures are *not undone* during backtracking in Leda.

btC, cf. [Liu 91], is an extension of C with backtracking features. Since only very simple backtracking instructions are added, all modifications on data – even on the program's heap memory – are trailable and will be undone during backtracking. This leads to a new kind of data-oriented backtracking mechanism and, in our opinion, this backtracking on data is needed to achieve a real integration of imperative and rule based programming paradigms. But data backtracking is quite difficult to realize and in btC the main amount of implementation efforts and conceptual work is needed for this topic.

In this paper we propose new mechanisms for data backtracking used for the implementation of PROGRES (cf. [Schürr 91, ZünSchü 91, Schürr 91b]). They surpass the control flow oriented backtracking mechanisms of Leda and are easier to implement than those for btC. In chapter 2 we give a short informal introduction to the parts of PROGRES which are necessary to understand our requirements for backtracking. Then we will discuss our approach to control flow backtracking (chapter 3.1) and data backtracking (chapter 3.2). In chapter 4, we propose some generalizations of the used techniques and discuss our results.

2 *The language PROGRES*

For an informal introduction to PROGRES we will use a small specification called the "ferryman's problem". This specification is a precise description as well as a programmed solution for 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 initial situation of "the ferryman's problem" in the form of a **di**rected, **a**ttributed, **n**ode- and **e**dge labelled graph (**diane**-graph).

For the definition of basic operations on diane-graphs PROGRES provides its users with means for the construction of *graph rewriting rules* (productions and tests) and *derived relations* on nodes (paths and restrictions).

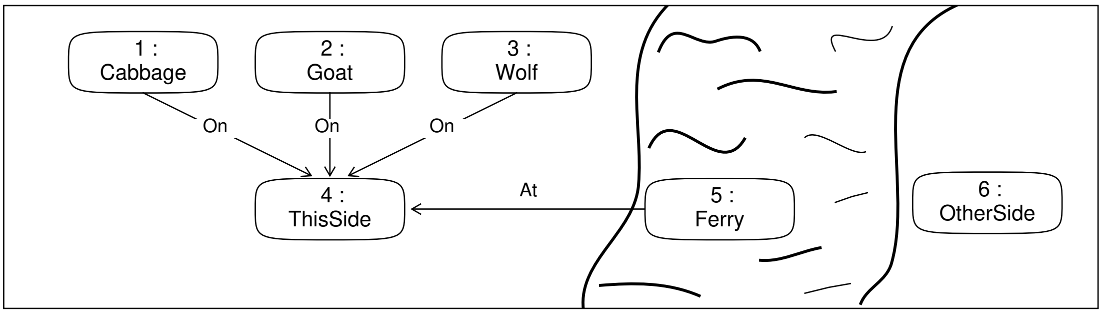

Figure 1: Initial situation for “the ferryman’s problem”

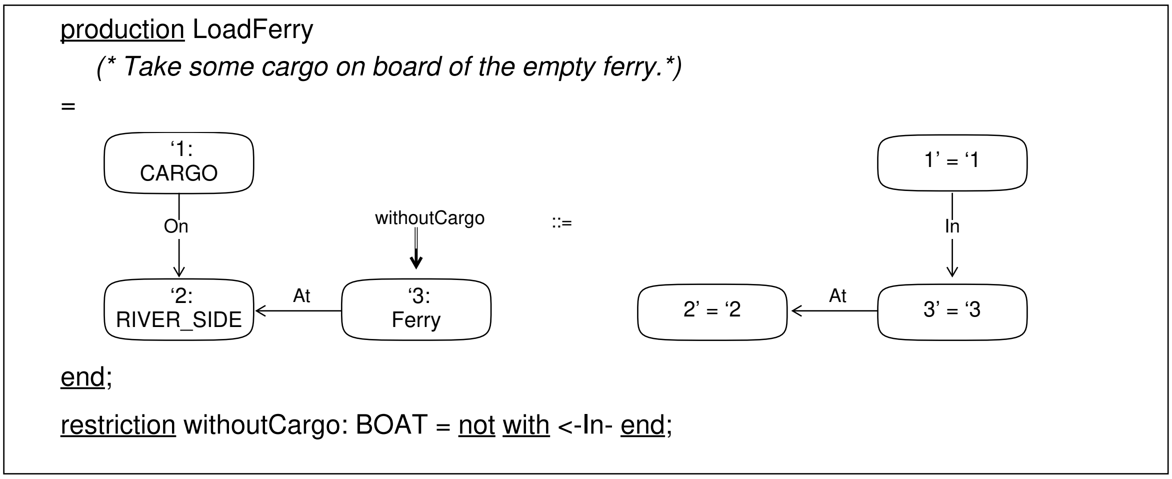

Figure 2: The production LoadFerry

Figure 2 shows the production LoadFerry. Applying this production to the graph in Figure 1 works as follows:

1. First, a (partial) subgraph of the current work graph which is isomorphic to the graph on the left hand side of the production is determined. I.e., a subgraph consisting of two nodes with types of the classes CARGO (Wolf, Goat, or Cabbage) and RIVER-SIDE (ThisSide or OtherSide) respectively, and 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 2. It requires the Ferry node not to be the target of an edge of type In. One subgraph of the graph in Figure 1 matching these conditions consists of the nodes 1, 4, and 5.
2. When a matching subgraph of the work graph has been selected, this subgraph is removed from the work graph and the graph on the right hand side of the production is inserted. A node inscription like “2’ = ‘2” designates 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 in Figure 1 is shown in Figure 3.

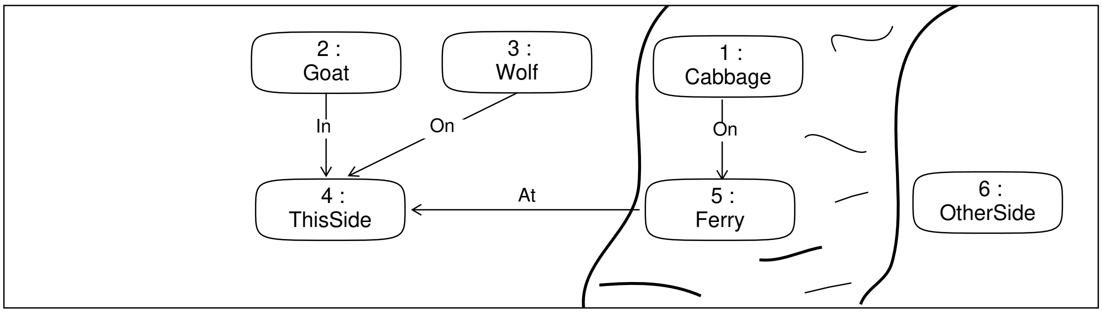

Figure 3: One possible result of the application of LoadFerry to the graph in Figure 1

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

- parameters for nodes, node types, and attribute values,
- application conditions concerning attribute values,
- instructions for changing the embedding of a newly inserted subgraph into the surrounding work graph,
- assignments to attributes of new or identically replaced nodes,
- assignments to return parameters.

These 'basic' (graph rewriting and testing) operations may be used to build complex graph transformations by the control structures of PROGRES. The simplest control structure of PROGRES is the sequence of operations, the &-statement:

LoadFerry & NotSupervisionNecessary & FerryOver

This simple construct already demonstrates the key problems for executing PROGRES. In general, there will be several subgraphs of the current work graph which are isomorphic to the left hand side of the applied production. So we have to choose one of the possible subgraphs for the application of the rewriting rule. This choice might influence the execution of the following operations dramatically. In PROGRES, the semantics of the &-statement ensures that - whenever possible - the interpreter chooses a subgraph for replacement which allows the successful execution of all following operations in the sequence. If it is not possible to execute the whole sequence, the graph remains unchanged.

In our example, the subgraph consisting of the nodes 1, 4, and 5 (cf. Figure 1, Figure 3) was a bad choice for replacement, because you should never leave wolf and goat without supervision on the same river side. The procedure NotSupervisionNecessary tests if the ferry may leave the current RIVER_SIDE without danger. Thus, the wrong choice of CARGO in procedure LoadFerry will cause the following operation NotSupervisionNecessary to fail. Only if we choose nodes 2, 4, and 5 for replacement by LoadFerry, the subsequent test succeeds and the whole sequence is executable. To achieve this behaviour, our deterministically working interpreter will have to perform some kind of backtracking. How we are going to do this is discussed in the following chapter.

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

3 *Implementation strategies*

As stated in the previous chapter, the execution of PROGRES involves the solution of the following backtracking problems. We assume a huge, complex structured, abstract data object, the current work graph, which may be changed by the execution of PROGRES productions. Since there may be several subgraphs matching the left hand side of an applied production, we have to choose one of them for rewriting. This choice in general will be wrong and will cause subsequent actions to fail. In order to execute all actions successfully, we have to go back to an earlier state of execution where we made a wrong decision and try another alternative. Going back to an earlier state of execution means 1) *backtracking of control* flow, i.e. going back to the program statement where a choice was made – even if this statement resides in a procedure which already has finished execution. And 2) *backtracking on data structures*, i.e. we must restore all data to its old state, even complex data objects like the work graph in PROGRES. Especially the necessity of data backtracking separates our approach from languages like Prolog and gives a new quality to the problem of backtracking.

3.1 *Control flow backtracking*

Within an imperative language, control flow backtracking demands that one is able to go back to a previous instruction which may be encapsulated in (maybe nested) control structures or resides (directly or indirectly) in a previously called subroutine which already has terminated.

The WAM (Warren **A**bstract **M**achine [Warren 83, Aït-Kaci 90]), the most famous abstract machine for backtracking languages, addresses this problem by using a technique named *environment protection*. Disregarding the technical details of this strategy, the current call frame on the machine's internal procedure call stack is not released on procedure return. Instead it is kept in memory together with its static and dynamic links and the current program counter value. On backtracking, the old execution state is retrieved by restoring the old procedure call stack using old frames and by just setting the program counter to its old value. The return address needed to jump back to the calling procedure is recovered together with the old call frame. However, this technique operates on the internal organization of the procedure call stack. If one does not want to implement this mechanism himself, he will have to use the WAM (or an equivalent abstract machine) directly or indirectly with the help of an *intermediate language* like Prolog or Leda. This involves the problem of implementing imperative control structures within these languages. Another alternative is to write or use an interpreter for the execution of the backtracking parts of an application. This may be crucial with respect to time efficiency (beside the problem of implementing such an interpreter).

Assuming that the main parts of an application are deterministic and implemented in a standard imperative programming language like Eiffel, C++, Ada, or Modula-2/-3, you may want to include some backtracking parts implemented in the same language. In order to use the described mechanism directly in a standard imperative language, one would have to modify the compiler and / or the runtime system to achieve the desired behaviour of environment protection.

In our solution, we just use a certain style of programming to achieve the same effects. The main idea is adopted from the implementation of btC, [Liu 91]. First we will discuss the translation of PROGRES productions to Modula-2 procedures. Then we will show how backtracking into these procedures works.

Every nondeterministic choice is translated into a loop containing a case statement controlled by a special status variable, cf. Figure 4. If the loop is executed for the first time

```
PROCEDURE LoadFerry( VAR Graph : Graphnumber; VAR BtInfo : BacktrackInfo);
VAR
  BtInfo_2, BtInfo_3, BtInfo_4 : BacktrackInfo;
  Node_1, Node_2, Node_3   : Nodenumber;
  Set_1, Set_2, Set_3      : SetType;
  SVar_BOOLEAN             : BOOLEAN;
BEGIN
  LOOP
     EvalStat( Graph, NotInnermost, BtInfo, BtInfo_2, Node_1, Set_1);
     CASE (BtInfo.Stat) OF
     | ExecInitial:
        GetAllNodesOfClass(Set_1, Graph, CARGO);
     | ExecInner:
        LOOP
           EvalStat( Graph, NotInnermost, BtInfo_2, BtInfo_3, Node_2, Set_2);
           CASE (BtInfo_2.Stat) OF
           | ExecInitial:
              GetAllNodesOfClass(Set_2, Graph, RIVER_SIDE);
           | ExecInner:
              LOOP
                 EvalStat(Graph, Innermost, BtInfo_3, BtInfo_4, Node_3,Set_3);
                 CASE (BtInfo_3.Stat) OF
                 | ExecInitial:
                    GetAllNodesOfClass(Set_3, Graph, BOAT);
                 | ExecInner:
                    TestEdge(SVar_BOOLEAN, Graph, Node_1, Node_2, On);
                    IF (SVar_BOOLEAN) THEN
                       TestEdge(SVar_BOOLEAN, Graph, Node_3, Node_2, At);
                       IF (SVar_BOOLEAN) THEN
                          SVar_BOOLEAN := NOT Empty( withoutCargo( Graph, Node_3));
                       END (* IF *);
                    END (* IF *);
                    SetBtInfo(SVar_BOOLEAN, BtInfo_4);
                 ELSE  EXIT;
                 END (* CASE *);
              END (* LOOP *);
           ELSE  EXIT;
           END (* CASE *);
        END (* LOOP *);
     ELSE  EXIT;
     END (* CASE *);
  END (* LOOP *);
  IF StatIsSuccess(BtInfo) THEN
     DeleteEdge( Graph, Node_3, Node_2, On);
     CreateEdge( Graph, Node_2, Node_1, In);
  END (* IF *);
END LoadFerry;
```

Figure 4: Modula-2 implementation of production LoadFerry

(BtInfo.Stat = Initial), the ExecInitial branch of the case statement is executed. Here, a set of candidates for the current choice is computed. In our example, this is done by the procedure GetAllNodesOfClass which will find three candidates: Wolf, Goat, and Cabbage. In the next iteration of the loop, the EvalStat procedure, cf. Figure 4, examines this set of candidates. If it is not empty,

In-Values			Out-Values		
BtInfo.Stat	SubBtInfo.Stat	InnermostFlag	BtInfo.Stat	SubBtInfo.Stat	Actions
Initial			ExecInitial	Initial	CreateTrailEntry(BtInfo.Trail);
ExecInitial			ExecInner /Failed (cf. Choose)	Initial	Choose(Set, Node, BtInfo.Stat);

ExecInner	Failed		ExecInner
			/Failed
			(cf. Choose)
				Initial	Choose(Set, Node, BtInfo.Stat);
					IF BtInfo.Stat = Failed THEN
					KillTrailEntry(BtInfo.Trail);
					END;

	Success		Success	Success	Trail(BtInfo.TrailNo, Set, Node, SubBtinfo.TrailNo);
NewChoice		NotInnermost	ExecInner	NewChoice	Restore(BtInfo.Trail, Set, Node, SubBtInfo.Trail);
		Innermost	ExecInner /Failed (cf. Choose)	Failed	RestoreSet(Set); Choose(Set, Node, BtInfo.Stat);

```
PROCEDURE Choose(VAR Set : SetType; VAR Node : Nodenumber;
                 VAR Status : ChoiceStatusType);
BEGIN
  IF NOT Empty( Set) THEN
     Node := Elem( Set); RemoveElem( Set, Elem);
     Status := ExecInner;
  ELSE
     Status := Failed;
  END (* IF *);
END Choose;
```

Figure 4: Behaviour of EvalStat

a candidate is chosen nondeterministically (Node_1) and the status of an inner loop (here BtInfo_2.Stat, in Figure 4 SubBtInfo.Stat) becomes Initial. Within the ExecInner branch, we test if the chosen candidate meets all required additional conditions and allows successful choices in all enclosed loops. If all matching conditions are fulfilled, the code of the ExecInner branch will set the SubBtInfo.Stat variable to Success. If one of the matching conditions fails, SubBtInfo.Stat is set to Failed. In this case, in the next iteration, EvalStat will choose another node from the current set of candidates and execute the ExecInner branch again. If no candidate is left, the loop terminates with status Failed and the surrounding loop will make a new choice calling the current loop again with status Initial. Eventually, the whole procedure will terminate with status Failed. If for every loop a candidate is found such that all requirements are fulfilled, the innermost status is set to Success and all loops terminate (one after the other propagating the success) so that the whole procedure terminates with status Success (after modifying the graph). Additionally some controlling variables are trailed for later use, cf. chapter 3.2.

If a procedure terminates successfully, its (nondeterministic) choices may cause a subsequent procedure to fail. In our example, we could have loaded the Cabbage within the procedure LoadFerry and this will cause the following test NotSupervisionNeccessary to fail. So we have to back-

```
PROCEDURE OneFerryStep( VAR Graph : Graphnumber; VAR LoadFerryBtInfo : BacktrackInfo);
VAR
  NotSupervisionNecessaryBtInfo : BacktrackInfo;
  FerryOverBtInfo : BacktrackInfo;
BEGIN
  LOOP
     EvalConcStat( Graph, LoadFerryBtInfo, NotSupervisionNecessaryBtInfo);
     CASE LoadFerryBtInfo.Stat OF
     | Initial, NewChoice : LoadFerry( Graph, LoadFerryBtInfo);
     | ExecInner :
        LOOP
           EvalConcStat(Graph, NotSupervisionNecessaryBtInfo, FerryOverBtInfo);
           CASE NotSupervisionNecessaryBtInfo.Stat OF
           | Initial, NewChoice : NotSupervisionNecessary( Graph, NotSupervisionNecessaryBtInfo);
           | ExecInner : FerryOver( Graph, FerryOverBtInfo);
           ELSE EXIT;
           END (* CASE *);
        END (* LOOP *);
     ELSE EXIT;
     END (* CASE *);
  END (* LOOP *);
END OneFerryStep;
```

Figure 5: Implementation of OneFerryStep

track into the procedure LoadFerry into all search loops and try to load another piece of CARGO. Therefore, we will call the procedure LoadFerry again with BtInfo.Stat = NewChoice. If we call a procedure with status NewChoice, the EvalStat procedure sets BtInfo.Stat of the current loop directly to ExecInner, skipping the Initial branch of the case statement which computes the candidate set. Then it restores the old values of the set of candidates and the currently chosen node, cf. chapter 3.2, and the status of the next enclosed loop is again set to NewChoice. This is done for all but the innermost loop. The innermost loop also restores the candidate set, but then it directly chooses a new element of this set as the new current node in order to finally find another matching subgraph for replacement.

This mechanism is utilized within procedure OneFerryStep (cf. Figure 5). This procedure tries to find a way for the successful execution of all subactions within our sample sequence

LoadFerry & NotSupervisionNecessary & FerryOver

It uses the same mechanism as the procedure LoadFerry. Only the EvalConcStat procedure, c.f. Figure 6, is less complex, because we don't have to examine a set of candidates. Called with status Initial, the Initial branch of the outermost case statement is executed. If successful, its ExecInner branch is called with NotSupervisionNecessaryBtInfo.Stat = Initial. NotSupervisionNecessaryBtInfo is passed to procedure NotSupervisionNecessary. In our example, this call will fail if LoadFerry did not load the Goat. This causes the inner loop to exit with failure so that the EvalConcStat call of the outer loop sets LoadFerryBtInfo.Stat to NewChoice. So, LoadFerry is called again but with status NewChoice and thus will choose another piece of CARGO for replacement. If the Goat is loaded, all subsequent actions will succeed.

Again, procedure OneFerryStep may be combined with other procedures to build a more complex graph operation like

OneFerryStep <u>or</u> FerryOver

In-Values		Out-Values	
BtInfo.Stat	SubBtInfo.Stat	BtInfo.Stat	SubBtInfo.Stat
Initial		Initial	Initial
Success	(* Initial/NewChoice*)	ExecInner	Initial
Failed	(* Initial/NewChoice*)	Failed	Failed
ExecInner	Success	Success	Success
ExecInner	Failed	NewChoice	Initial

Figure 6: Behaviour of EvalConcStat

If OneFerryStep is called with status NewChoice, then EvalConcStat sets LoadFerryBtInfo.Stat directly to ExecInner[1] and the enclosed NotSupervisionNecessaryBtInfo.Stat to NewChoice. Thus, LoadFerry and then NotSupervisionNecessary are skipped and we directly call FerryOver with status NewChoice. So, FerryOver will try to find another subgraph for rewriting. In our example, this will fail. Then the innermost EvalConcStat sets NotSupervisionNecessaryBtInfo.Stat to NewChoice calling procedure NotSupervisionNecessary in order to try another subgraph. This, again, will fail. Thus, the innermost loop terminates with failure and the outer loop will call LoadFerry with status NewChoice. As far as as our example proceeded, this causes LoadFerry to take the last possible CARGO, the Cabbage. Since NotSupervisionNecessaryBtInfo.Stat has been set to Initial in the previous loop iteration, NotSupervisionNecessary is called with this status, making a completely new attempt to execute the action sequence from this point. Due to the construction of our example, there is no other possibility to execute the sequence successfully. So, at least OneFerryStep will terminate with failure. This failure again may be handled by the calling procedure using the described mechanisms.

As you see, the proposed mechanism implements a forward chaining, depth first search in order to find a successful execution sequence for all subactions. The problem of infinite search paths and other search strategies are not addressed yet. This is future work.

Until now, we discussed how backtracking of control flow is managed in our approach just by using a special programming technique. In the following, data backtracking is considered.

3.2 *Data backtracking*

As described in the previous chapter, we are able to restore the state of control flow of a previous choice in order to try another alternative. But to perform backtracking as demanded in our example, we also have to restore the old values and states of our data structures. First of all, the underlying graph is modified by the execution of a production like LoadFerry. Additionally, the data used to control our control flow backtracking like the candidate sets and the current nodes

1. <u>NOT</u> to NewChoice

have to be restored in order to continue the search with the old search state (calling with status NewChoice).

Especially the necessity to restore arbitrary old states of a possibly very large graph during backtracking makes it impossible to use Prolog or comparable languages like Leda as implementation or intermediate language in our approach. In Prolog, all changed variables are reset to an undefined value during backtracking and not to an arbitrary previous value[1]. A clause loading the ferry would look like:

```
LoadFerry( OldGraph, NewGraph) :- . . .
```

It is called with the current graph state as first argument and an undefined variable as second. As result, unification will assign NewGraph one of the possible following graph states. So, in Prolog the whole data structure is copied in every modification step (cf. [SteSha 86] p. 286, the wolf, goat, and cabbage problem). In our approach, this would lead to serious time and space efficiency problems, since we often deal with huge graphs and perform only local modifications. In Prolog, large data structures are often modelled by fact clauses and modified by assert and retract clauses. But this mechanism has the disadvantage that the effects of assert and retract clauses are *not* undone during backtracking and therefore is not adequate for out purpose too.

btC refines the technique of Prolog by using a separate data trail stack. If a data value is modified between two backtracking points its original value (together with its size and address) is pushed to the data trail. During backtracking, this information is read and the old values are restored. This technique also covers data manipulation via pointers on the program's heap. Especially complex data with many pointer structures and allocate / deallocate operations is the key problems in the implementation of btC. Every manipulation of data – even through long pointer chains and using the typical C pointer arithmetic (ptr++) beside implicit type conversion mechanisms – has to be traced and trailed. In general, this technique needs an enormous amount of implementation efforts within the (pre-) compiler for btC.

In our approach, we use the nonstandard database management system GRAS (**GRA**ph **S**torage) [LewSchü 88] to implement data backtracking. GRAS has been developed at our department and is used for the implementation of graph like data structures for highly integrated software development environments [EnJaSchä 88]. The language PROGRES was developed for the specification of these data structures and the operations on them. Beside being an operational specification language, PROGRES may be considered as a manipulation and query language for GRAS. Now, GRAS offers sophisticated undo / redo mechanisms based on forward / backward executable graph deltas [Westf 89]. Every graph modifying operation is stored together with its reverse operation in a command list. To retrieve a previous graph state, this command list is traversed backwards executing the reverse operations. Thus, the costs of backtracking and data trailing operations are proportional to the modifications made and not to the size of the whole data structure.

Thus, in our approach it was an obvious idea to realize backtracking on data structures using this undo / redo mechanism of GRAS. On success, the innermost EvalStat causes GRAS to mark the current graph state for later retrieval. The candidate sets, the current nodes, and these marks are trailed with the help of a conventional trail stack. The trail stack entries are created by the EvalStat procedure if status is Initial and stored as part of the BtInfo variables, cf. Figure 4. On Success, the backtracking data is stored in this trail stack entry. If the procedure is called

1. In Prolog, (logical) variables receive defined values only during the unification process. The language does not support the assignment of new values to variables with already defined values.

with status NewChoice, the BtInfo parameter allows access to the correct data trail entry. This BtInfo variable and those of the enclosed loops are restored on the way to the innermost loop by the EvalStat procedures. The innermost EvalStat procedure additionally causes GRAS to undo all graph changes until the previously marked state is restored again. Now backtracking to the previous program execution state is done and execution proceeds again, searching for another matching subgraph.

4 Conclusions

There are several possibilities for the generalization and utilization of our approach. The afore mentioned form of the control flow backtracking mechanisms we proposed are specific for the language PROGRES. Nevertheless, it is possible to transfer this idea to arbitrarily nested control structures containing a choice point. Skipping parts of statement sequence is done by encapsulating them within if statements. These if statements are controlled by additional status variables. Backtracking into the correct branch of a case statement is accomplished by trailing its selector expression. This technique also works for other control structures, if all controlling conditions and assignments are trailed. Procedures get an additional parameter propagating the backtracking status.

The data backtracking mechanisms could also be generalized. The easiest way of implementing them is to use a database management system which supports undo to any previously marked state. This is the mechanism we used. Even a database system offering only a transaction concept (which allows an arbitrary number of open / pending transactions) would suffice. Marking a data state is done by starting a new transaction and using a counter for the number of open transactions. Backtracking is done by aborting a sufficient number of transactions. This will work but in some way the transaction mechanism is abused, since transactions will never be committed as long the execution of the program proceeds (all pending transactions are committed when the current session is closed, killing all backtracking states). And the number of open transactions grows proportional to runtime and not proportional to static control structure nesting (and recursion depth). A backtracking point within a loop causes a new open transaction in every iteration cycle. So the database must support an arbitrary number of open transactions.

So far, we have implemented a compiler (and an interpreter) for PROGRES productions which generated the contents of Figure 4 (slightly simplified by hand). The compiler for PROGRES control structures is under work. The compiler / interpreter is part of the PROGRES environment which also comprises a structure oriented, textual / graphical editor, an (fully implemented) incremental analyzer, and a graphical browser for the work graph. The whole system consists of about 240 KLOC in common basic modules and about 60 KLOC PROGRES-specific modules.

In order to avoid the usage of a time consuming database management system, it is possible to add an undo / redo feature to all used abstract data type modules. Therefore, all operations on the abstract data type must be able to denote their reverse operations. These reverse operations are trailed in a list with marks for special states. During backtracking, this list is executed backwards by some kind of interpreting mechanism. The implementation of such a data trail for an abstract data type should be more simple (and efficient) than the mechanism of btC, since it is realized on the abstraction level introduced by the abstract data type and its operations and does not need to keep track of arbitrary pointer manipulations.

In general, our approach is a low cost solution for the implementation of imperative / rule based languages or of rule based extensions to existing imperative programming languages or just for a rule based component of a mainly imperative program system. In our opinion, control flow backtracking is sufficiently efficient. The use of a database management system for data backtracking may be crucial but is the easiest solution. Alternatively, an undo mechanism may be implemented in the program's main memory as described above.

To summarize, in our opinion data backtracking is the key aspect of integrating imperative and rule based (logic) programming paradigms in a uniform language. The proposed techniques allow to push Prolog like rule based languages from the single assignment concept to the more imperative multiple assignment programming. Vice versa, an easy implementation of backtracking problems in Pascal like languages is made possible, even if complex data structures are involved. Since the imperative paradigm is very wide spread, these mechanisms may help us to exploit the knowledge and techniques of artificial intelligence within this area.

Appendix A: References

[Aït-Kaci 90] Aït-Kaci H.: The WAM: A (Real) Tutorial; Technical Report, 90/5, Digital Equipment Corporation (1990)

[Budd 91] Budd T.A.: Blending imperative and relational Programming; in: IEEE Software, 58-65, IEEE Computer Society Press (1991)

[EhKrRo 91] Ehrig H., Kreowski H.-J., Rozenberg G. (Eds.): Proc. 4th Int. Workshop on Graph-Grammars and their Application to Computer Science; LNCS 532, Berlin, Springer Verlag (1991)

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

[LewSchü 88] Lewerentz C., Schürr A.: GRAS, a Management System for Graph-like Documents; in: Proc. 3rd Conf. on Data and Knowledge Bases, 19-31, Morgan Kaufmann Publ. Inc., San Matheo, California (1988)

[Liu 91] Liu Y.: btC: a backtracking procedural language; Technical Report 203, University of Queensland, Key Centre for Software Technology (1991)

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

[Schürr 91b] Schürr A.: PROGRESS: A VHL-Language Based on Graph Grammars; in: [EKR 91], 641-659

[SchmiBer 92] Schmidt G., Berghammer R. (Eds.): Graph-Theoretic Concepts in Computer Science; LNCS 570, Berlin, Springer-Verlag (1992)

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

[Warren 83] Warren D.H.D.: An Abstract Prolog Instruction Set; Technical Report 309, SRI International, Menlo Park, CA (1983)

[Westf 89] Westfechtel B.: Extension of a Graph Storage for Software Documents with Primitives for Undo / Redo and Revision Control; Aachener Informatik-Berichte 89/8 (1989)

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

[ZünSchü 91] Zündorf A., Schürr A.: Nondeterministic Control Structures for Graph Rewriting Systems; in: [SchmiBer 92], 48-62