Programmed Graph Transformations and Graph Transformation Units in GRACE

Andy Schürr

Lehrstuhl für Informatik III, RWTH Aachen,
Ahornstr. 55, D-52074 Aachen, Germany

Abstract. Rule-based languages attract more and more attention as a high-level mechanism for the description of complex transformation processes on graph-like data structures. Unfortunately, pure rule-based approaches are not well prepared for expressing any kind of procedural knowledge. Therefore, various extensions were proposed which regulate the application of rewrite rules. This paper compares already existing regulation mechanisms and proposes a new approach based on control flow graphs. This approach is the first one, where complex control structures inherit all the properties of single rewrite rules, thereby allowing a smooth transition from a rule-oriented to an imperative programming paradigm. Finally, we will show how all reviewed regulation mechanisms may be defined using a very small set of basic concepts and a recently developed new fixpoint theorem. Having such a common formal background offers the opportunity to combine different regulation mechanisms within the future multi-paradigm graph grammar programming environment GRACE.

1 Introduction

Graphs play an important role within many application areas of computer science. Furthermore, rule-based systems are often used to describe complex transformation or inference processes on a very high level. Although graphs and rule-based systems are quite popular, their combination in the form of *graph rewriting systems* or *graph grammars* were more or less unknown to the majority of computer scientists for a long time.

Nowadays the situation is gradually improving with the appearance of system implementations like AGG [11], GraphED [8], PAGG [7], and our system PROGRES [16, 17, 22]. There exist even plans to combine the efforts of different groups in Bremen, Aachen, Berlin, and Leiden to develop a *GRaph Centered Environment*, called *GRACE* [9, 10], which encompasses at least the functionality of the systems AGG and PROGRES and supports various graph grammar approaches within a common framework. The dream behind GRACE is to simplify the selection of a "suitable" graph grammar approach for a given application domain and to allow even the combination of different approaches within a single specification if necessary. It is our hope that such an environment makes the potentials of graph grammars and graph rewriting systems more visible and applications of them more popular.

Within this future environment and within currently existing systems like PAGG and PROGRES, additional means are or should be offered which allow us to regulate the application of a large set of graph rewrite rules and to divide such a set of rewrite rules into manageable subsets with well-defined interfaces and hidden implementation details.

Many quite different proposals exist how to *control application of rewrite rules*, with each of them having its specific (dis-)advantages [1, 14, 4, 13, 19], as for instance:

- Apply rules as long as appropriate or as long as possible in any order; this is the *standard semantics* of rewriting systems.
- Introduce *rule priorities* and prefer applicable rules with higher priorities at least in the case of overlapping matches (redices).
- Use *regular expressions* or even complex graph rewriting programs to define permissible derivation sequences.
- Draw *control flow graphs* and interpret them as graphical representations of rule controlling programs.

Taking the goals of GRACE into account it seems to be useful to develop a *common framework* for the definition and comparison of *rule regulation mechanisms*. This is a necessary prerequisite for offering a number of rule regulation mechanisms in GRACE, which complement each other, and for defining the semantics of specifications which use a combination of offered mechanisms. In our opinion, recursively defined control structures of programmed graph grammars are the most powerful approach to regulate rewriting processes [5, 7, 19]. Their semantics are definable by means of so-called "*basic control flow (BCF) operators*" and a recently developed new fixpoint theorem [15, 18]. Unfortunately, these programs emphasize the imperative programming paradigm and have a textual representation. Both properties are inconsistent with the overall goal of visual and rule-oriented programming with graph rewriting systems.

Within this paper, we will first recall the definition of BCF operator expressions and their accompanying fixpoint semantics definition introduced in [18]. Furthermore, we will compare various rule controlling approaches and discuss their relationships to BCF expressions. Finally, we will show how BCF expressions may even be used to define the semantics of a very powerful new kind of graph rewriting control flow graphs. These *control flow graphs* have about the same expressiveness as graph rewriting programs in [19], and preserve the rule-oriented as well as visual character of pure graph rewriting systems as far as possible.

2 Graph Transactions and Transformation Units

So-called *programmed graph grammars* were already suggested many years ago in [1, 14]. Nowadays, they are a fundamental concept of the graph grammar specification languages PAGG [7] and PROGRES [17, 19]. Experiences with using these languages and their predecessors for the specification of software engineering tools [6, 20] showed that their control structures should possess the following properties:

(1) *Boolean nature:* the application of a programmed graph transformation either succeeds or fails — like the application of a single graph rewrite rule — and, depending on success or failure, execution may proceed along different control flow paths.

(2) *Atomic character:* a programmed sequence of graph rewrite steps modifies a given host graph if and only if all its intermediate rewrite steps do not fail.

(3) *Consistency preserving:* programmed graph rewriting has to preserve a graph's consistency with respect to a given set of separately defined integrity constraints.

(4) *Nondeterministic behavior:* the nondeterminism of a single rewrite rule — it replaces any match of its left-hand side — should be preserved as far as possible on the level of programmed graph transformations.

(5) *Recursive definition:* for reasons of convenience as well as expressiveness, programmed graph transformations should be allowed to call each other without any restrictions including any kind of recursion.

Without conditions (1) through (4) we would have difficulties to replace a complex graph rewrite rule by an equivalent sequence of simpler rewrite rules, and without condition (5), the manipulation of recursively defined data structures would be quite cumbersome.

In the sequel, programmed graph transformations with the above mentioned properties will be termed *graph transactions*. The usage of the term "transaction" underlines the most important properties of programmed graph transformations: *atomicity* and *consistency*. The usually mentioned additional properties of transactions, *isolation* and *duration*, are irrelevant as long as parallel programming is not supported and backtracking cancels the effects of already successfully completed transactions (cf. section 3).

As the reader may imagine, it is very difficult to develop a sound formal definition of graph transactions. Until now, proposals for programmed graph rewrite systems compromised either their boolean nature [14, 13] or their atomic as well as nondeterministic character [7], or they were not able to provide a meaningful semantics for all constructible programs [17]. It is even more difficult to develop efficiently working implementations of graph transactions. Our system PROGRES is the first and as far as we know the only representant of a programmed graph rewrite system implementation with the above mentioned properties [19, 21, 22].

Still missing in any (implemented) graph rewrite system are additional means for the afore mentioned decomposition of rule sets and rule controlling programs into separate components. Therefore, proposals have been made to introduce so-called *graph transformation unit*s as the fundamental concept of GRACE [10]. These units realize binary relations between graphs in a similar way as graph rewrite rules and graph transactions. Furthermore, they have a hidden body, which consists of a set of own graph rewrite rules, a set of imported (graph rewrite rules from other) transformation units, and so-called control conditions which regulate the application of rewrite rules.

A precise definition of syntax and semantics of GRACE control conditions is still subject to ongoing research activities [9]. Currently, rule priorities, graph transactions, and control flow graphs are considered as possible candidates. To summarize, graph transactions of PROGRES and application conditions of transformation units in GRACE share many properties, and the same approach should be used to define their intended semantics.

3 Basic Control Flow Operators

The definition of a fixed set of control structures for graph transactions is complicated by contradicting requirements. From a *theoretical point of view* the set of offered control structures should be as small as possible, and we should be allowed to combine them without any restrictions. But from a *practical point of view* control structures are

<Transaction>	::=	<TransactionId> "=" <BCFExpr> ;
<BCFExpr>	::=	<BasicAction> | <ActionCall> | <BCFTerm> ;
<BasicAction>	::=	"skip" | "loop" ;
<ActionCall>	::=	<RuleId> | <TransactionId> ;
<BCFTerm>	::=	"def" "(" <BCFExpr> ")" | "undef" "(" <BCFExpr> ")" |
		"(" <BCFExpr> ";" <BCFExpr> ")" |
		"(" <BCFExpr> "[]" <BCFExpr> ")"
		"(" <BCFExpr> "&" <BCFExpr> ")";

Fig. 1: Syntax of graph transactions and BCF expressions

needed which are easy to use and to understand, which cover all frequently occurring control flow patterns, and the application of which may be directed by a number of context-sensitive rules.

Therefore, it was quite natural to distinguish between basic control flow operators of an underlying theory of recursively defined graph transactions and more or less complex control structures of a higher level programming language. Starting with a formal definition of basic control flow operators, in the sequel termed *BCF operators*, we should then be able to define the meaning of arbitrary control structures by translating them into equivalent BCF expressions.

Figure 1 contains the definition of BCF expressions and of graph transactions as functional abstractions of BCF expressions. It distinguishes between *basic actions* like

- skip, which represents the always successful identity operator and relates a given graph G to itself, and
- loop, which neither succeeds nor terminates for any given graph and represents therefore “crashing” or forever looping computations,

calls of basic rewrite rules or other graph transactions, and finally between two unary and three binary *BCF operators* with

- def(A) as an action which succeeds applied to a given graph G, whenever A applied to G produces a defined result, and returns G itself,
- undef(A) as an action which succeeds applied to a graph G, whenever A applied to G terminates with failure, and returns G itself,
- (A ; B) as an action which is the sequential composition of A and B, i.e. applies first A to a given graph G and then B to any “suitable” result of B,
- (A [] B) as an action which represents the nondeterministic choice between the application of A or B, and
- (A & B) as an action which returns the intersection of the results of A and B, which requires an equality or isomorphy testing operator on graphs.

Note that the operators suggested above are intentionally similar to those proposed by Dijkstra [3] and especially to those presented by Nelson [15] with one essential difference: due to the boolean nature of basic graph rewrite rules and complex graph transactions, we are not forced to distinguish between side effect free boolean expressions and state modifying actions. This has the consequence that complex guarded commands of the form

$$(Cond_1 \rightarrow Body_1 \; [] \ldots [] \; Cond_n \rightarrow Body_n)$$

are no longer necessary (such a command executes a nondeterministically selected body with a valid boolean condition; it fails if all conditions are invalid). They may replaced by expressions like

(def($Cond_1$) ; $Body_1$) [] … [] (def($Cond_n$) ; $Body_n$)

where "def($Cond_i$)" tests either the applicability of a single graph rewrite rule or of a graph transaction without modifying the given input graph. It is even possible to write

($Cond_1$; $Body_1$) [] … [] ($Cond_n$; $Body_n$)

if $Cond_1$ through $Cond_n$ are side effect free expressions or if graph modifying side effects of their execution are even required.

Furthermore, BCF expressions offer almost all possibilities for combining binary relations, the semantic domain of graph transactions. There are operators for intersection, union, and concatenation. The *missing difference operator* is not supported due to the following reasons: A \ B1 is a subrelation of A \ B2, if B2 is a subrelation of B1. As a consequence, the nonmonotonic difference operator had to be excluded in order to be able to come up with a sound definition for recursively defined graph transactions (cf. section 4). It would be possible to define a difference operator where refinements of its second argument are simply prohibited. This corresponds to the requirement that the termination of the evaluation of its second argument must be guaranteed in advance. But such an extensions seems to be unnecessary, taking into account that we are able to define the semantics of all aforementioned rule regulation mechanisms without any needs for building differences.

Having motivated our reasons for the selection of two unary and three binary BCF operators, we are now prepared to discuss the intricacies of their intended semantics. A first problem comes with the definition of the meaning of "(A ; B)" as "apply B to *any suitable* result of A". Let us assume that the application of a graph transformation A to a graph G has three possible results named G1, G2, and G3, respectively. Furthermore, let us assume that the graph transformation B applied to G1 fails but applied to G2 and G3 succeeds. In this case we may either select G2 or G3 but not G1 as a suitable result of the application of A. This means that we need knowledge about future states of an ongoing graph transformation process in order to be able to discard those possible results of a single transformation step, which cause failure of the overall transformation process.

It should be quite obvious that, in general, this kind of clairvoyant nondeterminism requires either a breadth-first search implementation or a depth-first search implementation with backtracking out of "dead-ends". Note that the realization of a *breadth-first search* strategy requires the maintenance of eventually very large sets of graphs, where each graph may itself be a large storage consuming data structure. Therefore, we were forced to adhere to a *depth-first search* semantics while implementing the control structures of the language PROGRES, although breadth-first search strategies are better prepared to deal with certain termination problems.

Furthermore, we were even forced to vote for a *depth-first search semantics* for the BCF operators of this section. Choosing a breadth-first search semantics in theory and a depth-first search semantics in practice results in formal definitions which cannot be used at all for reasoning about or explaining the termination behavior of implemented graph transactions.

Another problem comes with the definition of expressions like "(A [] B)" where A loops forever applied to a certain graph G but B has a well-defined set of possible results. Having a depth-first search semantics in mind, we are forced to define the outcome of the expression "(A [] B)" as being either a nonterminating computation or any defined result produced by B.

This means that the kind of *nondeterminism* we are going to define is not "angelic" but more or less "*erratic*". Using backtracking we are able to discard (nondeterminstic) selections which lead to defined failures of basic actions or graph rewrite rules but not selections which cause nonterminating computations.

4 Fixpoint Semantics for Graph Transactions

After an informal introduction of graph transactions as named BCF expressions we will now define their intended semantics: this is a *semantic function* from the syntactic domain of BCF expressions onto the range of (extended) binary relations over graphs. In order to be able to deal with recursion and nondeterminism in the presence of an atomic sequence operator ";"[1] we had to follow the lines of [15]. There, a new form of the fixpoint theorem is used to give an axiomatic definition of nondeterministic commands.

For the sequel, a certain class of (abstract) graphs G and a set of rewrite rules R is expected to be defined elsewhere together with an equality/isomorphy testing operator "$\cong$". Furthermore, we demand the existence of a function S: $R \rightarrow G \times G$ which maps rewrite rules onto binary relations over graphs and which has the following meaning:

(G1, G2) $\in S[\![r]\!] :\Leftrightarrow$ *r applied to G1 may produce a graph G2.*

$\neg \exists$ *G: (G1, G2)* $\in S[\![r]\!] :\Leftrightarrow$ *r is not applicable to G1*

Note that we assume for simplicity reasons only that the attempt to apply a rewrite rule r to a given graph G always terminates with either returning an arbitrary graph out of the set of all possible results or aborting without any results. This is no longer true for graph transactions which may either return or loop forever applied to a given graph. Therefore, we need an appropriate *extension of the semantic domain* of binary relations over graphs:

$$D := 2^{G \times (G \cup \{\infty\})}.$$

The semantics of a graph transaction is a binary relation between graphs, where the symbol "∞" in a second component is intended to represent *potentially nonterminating computations*. The word "potential" includes computations with unknown effects, i.e. computations which either return still unknown results or abort or loop forever. A relation S_∞ for instance, which maps any given graph G onto "∞", represents a computation with completely unknown outcomes.

Now, we are ready to define a semantic function S from the syntactic domain of BCF expressions of section 3 onto the semantic domain D inductively:

Definition 1 (semantics of BCF expressions)
With A, B $\in$ *<BCFExpr> our function S: <BCFExpr>* $\rightarrow$ *D is inductively defined as follows:*

1. The sequence operator is not (chain) continuous in its 2nd argument. Therefore, the original version of the fixpoint theorem, proved in [12], does not work in our case.

(1) $(G, G') \in S⟦ \underline{skip} ⟧ :\Leftrightarrow \quad G = G'$
(2) $(G, G') \in S⟦ \underline{loop} ⟧ :\Leftrightarrow \quad G' = \infty$.
(3) $(G, G') \in S⟦ \underline{def}(A) ⟧ :\Leftrightarrow$
$\exists\, G'' \neq \infty : \; ((G,G'') \in S⟦ A ⟧ \;\wedge\; G = G') \vee ((G,\infty) \in S⟦ A ⟧ \;\wedge\; G' = \infty)$.
(4) $(G, G') \in S⟦ \underline{undef}(A) ⟧ :\Leftrightarrow$
$((\neg \exists\, G'' : (G,G'') \in S⟦ A ⟧) \wedge \; G = G') \vee ((G,\infty) \in S⟦ A ⟧ \;\wedge\; G' = \infty)$.
(5) $(G, G') \in S⟦ (A \,;\, B) ⟧ :\Leftrightarrow$
$(\exists\, G'' \neq \infty : (G,G'') \in S⟦ A ⟧ \wedge \; (G'',G') \in S⟦ B ⟧) \;\vee ((G,\infty) \in S⟦ A ⟧ \;\wedge G' = \infty$
).
(6) $(G, G') \in S⟦ (\, A \,[]\, B) ⟧ :\Leftrightarrow$
$(G, G') \in S⟦ A ⟧ \;\vee\; (G, G') \in S⟦ B ⟧$.
(7) $(G, G') \in S⟦ (\, A \,\&B) ⟧ :\Leftrightarrow$
$(G' = \infty \wedge ((G, \infty) \in S⟦ A ⟧ \;\vee\; (G, \infty) \in S⟦ B ⟧)$
$\vee ((G, G') \in S⟦ A ⟧ \;\wedge \exists\;\; G'' : (G, G'') \in S⟦ B ⟧ \wedge \;\; G' \cong G'')$. ❑

The definitions above are rather straightforward with the exception of the treatment of the operators <u>def</u> and <u>undef</u>. The expressions <u>def</u>(A) and <u>undef</u>(A) loop forever if A returns not a single defined result but loops forever. Furthermore, <u>def</u>(A) may return its input if A returns at least one defined result, even if A may loop forever. It terminates with failure if A terminates with failure. On the other hand, <u>undef</u>(A) returns its input if and only if A fails, and it fails if and only if A has at least one defined result and may not loop forever.

Therefore, <u>undef</u> is *stricter* than <u>def</u> with respect to the treatment of looping computations. From a practical point of view, this distinction may be justified as follows:

- Often, we would like to know whether A computes at least one result without evaluating all possible execution paths of A after a successful path has been found.
- On the other hand, answering the question whether A fails is not possible without taking all execution paths of A into account, thereby running into any nonterminating execution branch of A.

Nevertheless, a *strict version* of <u>def</u> might be useful, too, which checks whether a subexpression returns *always* a defined result. Its definition is no longer independent from the definition of <u>undef</u> but may be given as follows:

Definition 2 (strict version of def operator)
A strict version of the <u>def</u> operator may be defined as follows with A ∈ <BCFExpr>:
$(G, G') \in S⟦ \underline{def}^{S}(A) ⟧ :\Leftrightarrow \;\; (G, G') \in S⟦ \underline{undef}(\underline{undef}(A)) ⟧$
$\Leftrightarrow \quad (\neg \exists\, G'' : (G, G'') \in S⟦ \underline{undef}(A) ⟧) \wedge \;\; G = G'$
$\vee \quad (G, \infty) \in S⟦ \underline{undef}(A) ⟧ \;\wedge\; G' = \infty$
$\Leftrightarrow \quad \exists\, G'' \neq \infty : (G, G'') \in S⟦ A ⟧) \wedge \;\; (G, \infty) \notin S⟦ A ⟧ \wedge \;\; G = G'$
$\vee \quad (G, \infty) \in S⟦ A ⟧ \;\wedge\; G' = \infty$. ❑

For the proof that all defined BCF operators are monotonic with respect to a suitable partial order on D — a necessary prerequisite for any fixpoint theorem — the reader is referred to [18]. Based on this proof we are able to define a fixpoint semantics for a given set of graph transactions, which may call each other in an arbitrary manner.

Please note that the treatment of *n transactions* instead of one requires the introduction of a more complex semantic domain D^{n} and a corresponding semantic function. This function takes n binary relations as input, which are already computed approxima-

tions of n transactions; it produces better approximations in the form of n binary relations as output. Applying fixpoint theory to this extended setting the semantics of the i-th transaction is defined as the i-th component of their common least fixpoint, an n-dimensional vector over D. For the proof of the *existence of unique least fixpoints* the reader is again referred to [18].

Definition 3 (fixpoint semantics of graph transactions)
Let E_1 to $E_n \in$ <BCFExpr> be BCF expressions which contain at most calls to graph rewrite rules in R and calls with transaction identifiers t_1 through t_n. These expressions may be used to define n (mutually recursive) graph transactions $t_1 = E_1, \dots, t_n = E_n$. The common unique least fixpoint $(S[\![t_1]\!], \dots, S[\![t_n]\!]) \in D^n$ of these graph transactions defines their semantics and may be approximated as follows:

$S^0 := (S_\infty, \dots, S_\infty) \in D^n$ *with S_∞ being the relation $G \times \{\infty\}$,*

$S^{k+1} := (S[\![E_1]\!] [S_1^k, \dots, S_n^k], \dots, S[\![E_n]\!] [S_1^k, \dots, S_n^k])$,

where $S[\![E_i]\!] : D^n \to D$ takes already computed approximations for $t_1, \dots, t_n$ as input and yields a new approximation for t_i by applying definition 1 to expression E_i. ❑

The definition above completes the presentation of BCF expressions and their abstraction to graph transactions, which have the five *required properties* of section 2:

(1) The operators "def" and "undef" allow us to test success and failure of any subcomputation and to continue along different execution paths depending on their results.
(2) The definition of the operator ";" guarantees that the execution of any graph transaction succeeds or fails as a whole.
(3) Furthermore, ";" guarantees that failing graph transactions do not return an inconsistent intermediate computation result but keep their input graph unmodified.
(4) The operator "[]" supports the construction of nondeterminstic computations on the level of control structures.
(5) And graph transaction may call each other without any restrictions.

5 Fixpoint Semantics for Various Programming Approaches

Having presented a hopefully minimal as well as sufficiently large set of BCF operators, we still have to show their appropriateness for the definition of *complex control structures*. For this purpose, figure 2 contains a selection of typical examples and their

semantic definitions. The first five lines define slightly simplified control structures of the language PROGRES [19]:

abort	$\hat{=}$	undef(skip) .
do A or B end	$\hat{=}$	(A [] B) .
do A and B end	$\hat{=}$	((A ; B) [] (B ; A)) .
if A then B else C end	$\hat{=}$	((def(A) ; B) [] (undef(A) ; C)) .
while A do B end	$\hat{=}$	T = ((def(A) ; (B ; T)) [] undef(A)) .
begin A; B end	$\hat{=}$	((def(A) [] def(B)) ; (A [] undef(A)) ; (B [] undef(B))) .
try A else B end	$\hat{=}$	if A then A else B end .
repeat A end	$\hat{=}$	while A do A end.
(A > B)	$\hat{=}$	repeat try A else B end end .

Fig. 2: Proposals for complex control structures

- The first line introduces a new basic action “abort”, which never succeeds but always terminates, as a complement of “skip”.
- The first proposed complex control structure “do A or B end” executes either A or B and fails if neither A nor B are applicable.
- The following complex control structure “do A and B end” tries to execute A and B in any possible order and fails if neither A before B nor B before A succeeds.
- The next control structure “if A then B else C end” tests whether A would be applicable; it executes B if A is applicable and executes C in case of defined failure of A.
- The fifth line defines conditional iteration: B is executed as long as A is applicable. The iteration process terminates successfully as soon as A is no longer applicable, it fails whenever A is applicable but B not, and it loops forever if A never fails.

The next three lines of figure 2 are an attempt to define the semantics of control structures in PAGG [7]. Its sequential composition operator *compromises* the required *atomic character* of graph transactions in favor of an efficiently working implementation. They have to be read as follows[2]:

- The construct “begin A; B end” requires sequential application of A followed by B. Its definition is rather complicated since its execution fails only if both A and B fail. Otherwise, the failing subexpression is skipped and the execution process continues.
- The construct “try A else B end” is a convenient shorthand for “if A then A else B end” which may even be used with more than two arguments in the general case.
- The construct “repeat A end” is a convenient shorthand for “while A do A end”. It may also be used with more than one argument between “repeat” and “end”, but this is just another shorthand for nesting a “begin/end” statement within “repeat/end”.

Using the nonatomic sequence statement and offering no means which are equivalent to the previously introduced BCF operators def and undef reduces the expressiveness of PAGG significantly, but has the advantage that an implementation has no needs for backtracking and undoing already performed graph modifications.

---

2. The original definition uses “sapp” instead of “begin”, “capp” instead of “try” and “wapp” instead of “repeat”.

Another recently developed graph grammar programming approach [5] has about the same properties and deficiencies as PAGG and is, therefore, not explicitly mentioned in figure 2. It offers merely a combination of the already presented or of PROGRES, the if-then-else with a restriction to boolean conditions, and finally the nonatomic begin/end of PAGG.

Finally, the last line of figure 2 defines the semantics of rule priorities with "(A > B)" as apply "A and B as often as possible and prefer A if both A and B may be applied". This *definition of rule priorities* is just an *approximation* of their usual semantics. According to their definition over here, the execution of B is blocked as long as A is executable, even in the case were A and B modify disjoint parts of a given graph. Unfortunately, a more liberal definition of rule priorities is beyond the capabilities of BCF expressions. This is a principle problem of our approach, since enlarging an already nonempty set of redices (matches) within a graph for a rule A may reduce the set of allowed redices of another graph rewrite rule B with lower priority. This is a kind of nonmonotonic behavior, we are not able to deal with.

Beside these more or less graph grammar oriented programming approaches many other approaches for regulating the application of rewrite rules do exist. The most popular ones are presented in [4] together with an in-depth discussion of their expressiveness and related decidability problems. So-called *matrix grammars* offer means for defining rewriting processes with the following flow of control:

$T = (((r_{1,1} ; \dots ; r_{1,k(1)}) [] \dots [] (r_{n,1} ; \dots ; r_{n,k(n)})) ; (T [] \underline{skip}))$,

where all $r_{i,j}$ are simple rewrite rules. They combine the usual "apply as long as needed" semantics of unregulated rewriting systems with the idea that a sequence of rewrite rules instead of a single rewrite rule should be applied. The reader should notice that the semantics of sequential application of matrix grammars fulfills our atomicity requirement, but all other required properties for control structures in section 2 are still missing.

So-called *programmed grammars* in [4] are another step towards really useful control structures. They are equivalent to sets of graph transactions as they were defined over here, where each transaction T has the following form:

$T = (r ; (S_1 [] \dots [] S_n) [] (\underline{undef}(r) ; (F_1 [] \dots [] F_n))$

with r being a simple rewrite rule, and $S_1, \dots S_m$, $F_1, \dots, F_n$ being other transactions. In this way programmed grammars allow the definition of sequences of rule applications and they even support branching in the flow of control depending on success or failure of single rewrite rules. But without any support for functional abstraction they do not allow branching in the flow of control depending on success or failure of complex rewriting processes.

6 Fixpoint Semantics for Control Flow Graphs

In the previous section typical exponents of various programmed rewriting approaches were presented on a more or less informal level. In all cases denotational semantics definitions were sketched by translating their control structures into sets of graph transactions with a well-defined fixpoint semantics definition. But all discussed approaches — except graph transactions themselves and their more readable form in the language PROGRES — had severe deficiencies. And even graph transactions with their low level text representation compromise the *visual nature* of programming with graph rewrite rules.

An obvious solution for this problem is to replace text-oriented control structures by some kind of *control flow graphs*. But until now, all suggested variants of control flow graphs were either less expressive than our graph transactions or came without a sound denotational semantics definition:

- Bunke proposed in [1] control flow graphs, which are not allowed to call each other in an arbitrary manner, i.e. do not properly support functional abstraction.
- Nagl introduced control flow graphs in [14], where a node may be a call to another control flow graph, but where branching depending on success or failure of subgraph calls is not supported.
- And the author of this paper suggested in [17] control flow graphs without the above mentioned deficiencies, but also without a nonoperational semantics definition.

In the sequel, we will show how BCF expressions may be used to define a fixpoint semantics for the control flow graphs of [17]. These control flow graphs are equivalent to graph transactions with strict $\underline{def}^{s}$ operators and without the intersection operator &. Figure 4 contains an example of a control flow graph with two subgraphs and their textual counterparts. Their main properties are:

- Any node in a subgraph should be reachable from its unique start node and should have a path to its unique stop node.
- The execution of a (sub-)graph begins at its start node and ends at its stop node.
- All remaining nodes have an inscription which is either nop (for no operation has to be performed) or the call of a simple rewrite rule or another control flow subgraph.

Control flow graphs for S = repeat try A else B end end
A = do A1 and A2 end

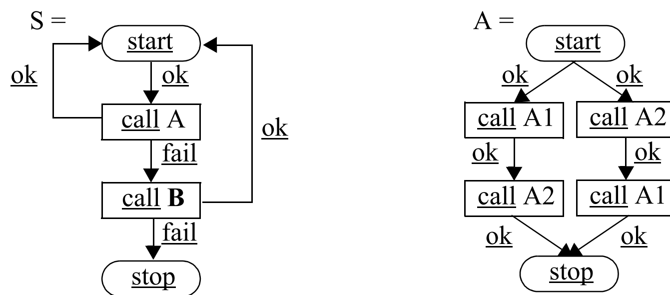

Fig. 3: Examples of control flow graphs

- The execution of a node either succeeds or fails; executions of start, stop, and nop nodes always succeed without any graph modifying effects.
- After a successful execution of a node we have to follow one of its outgoing ok edges; multiple ok edges reflect nondeterministic computations and absence of ok edges partially defined computations.
- After a failing execution of a node we have to follow one of its outgoing fail edges; multiple fail edges reflect again nondeterministic computations and absence of fail edges again partially defined computations.

For a more detailed explanation of this kind of control flow graphs, including an operational semantics definition, the reader is referred to [17]. A *denotational semantics definition* may be provided by translating a control flow graph into an equivalent set of graph transactions. Any node of the control flow graph will be translated into a transaction with calls to other transactions reflecting its outgoing ok and fail edges:

Definition 4 (semantics of control flow graphs)
Any control flow graph node n with label "call A" or "start" or "nop", outgoing ok edges to nodes $n_1, \dots, n_j$, and outgoing fail edges to nodes $m_1, \dots, m_k$ is translated into:

$n = (S \,;\, (n_1 \,[]\, \dots \,[]\, n_j)) \,[]\, (\underline{undef}(S) \,;\, (m_1 \,[]\, \dots \,[]\, m_k))$, *if* $j > 0, k > 0$
and into $n = (S \,;\, (n_1 \,[]\, \dots \,[]\, n_j))$, *if* $j > 0, k = 0$
$n = (\underline{undef}(S) \,;\, (m_1 \,[]\, \dots \,[]\, m_l))$, *if* $j = 0, k > 0$
and into $n = abort := \underline{undef}(\underline{skip})$, *if* $j = 0, k = 0$
where $S = A$, *if inscription of node is "*call A"
and $S = \underline{skip}$, *otherwise.*
Finally, any node n with inscription "stop" is translated into the transaction
$n = \underline{skip}$. ❑

Theorem 5 (equivalence of graph transactions and control flow graphs)
Any set of graph transactions without the intersection operator & and with a strict $\underline{def}^{\,s}$ *operator (cf. def. 1 and def. 2) may be translated into an equivalent set of control flow graphs according to definition 4 and vice versa.*

Proof. The translation of any control flow graph into an equivalent BCF expression is an essential part of definition 4. The reverse direction, the translation of BCF expressions without & and def occurrences, is defined by the graph rewrite rules in fig. 4. These rewrite rules transform a node which has the definition of a graph transaction as its label into an equivalent control flow graph. Rewrite rules are specified in a PROGRES like notation with double dashed nodes representing embedding rules. The rewrite rule TranslateUndef redirects for instance any adjacent ok and fail edge of the rewritten node "undef(A)" to the new node "call T". ❑

Please note that the given proof is just a sketch. Still missing is (the more or less trivial) part which checks (by induction) that the output of the translation of any BCF ex-

pression E into a control flow graph followed by the translation back into another BCF expression E' preserves indeed its semantics, i.e. that $\mathcal{S}[\![E]\!] = \mathcal{S}[\![E']\!]$.

7 Conclusion

This papers constitutes a *new framework* for defining and comparing various approaches which support programming with (graph) rewrite rules. Based on this framework, advantages and disadvantages of previously suggested programming formalisms were discussed. The results of this discussion are summarized in table 1 with column headlines referring to required rule programming properties in section 2.

Motivated by apparently existing deficiencies of reviewed approaches, a new type of control flow graphs was introduced, which allows a smooth transition from rule-oriented programming to the imperative programming paradigm and which supports *visual programming with graph rewrite rules* on a very high level.

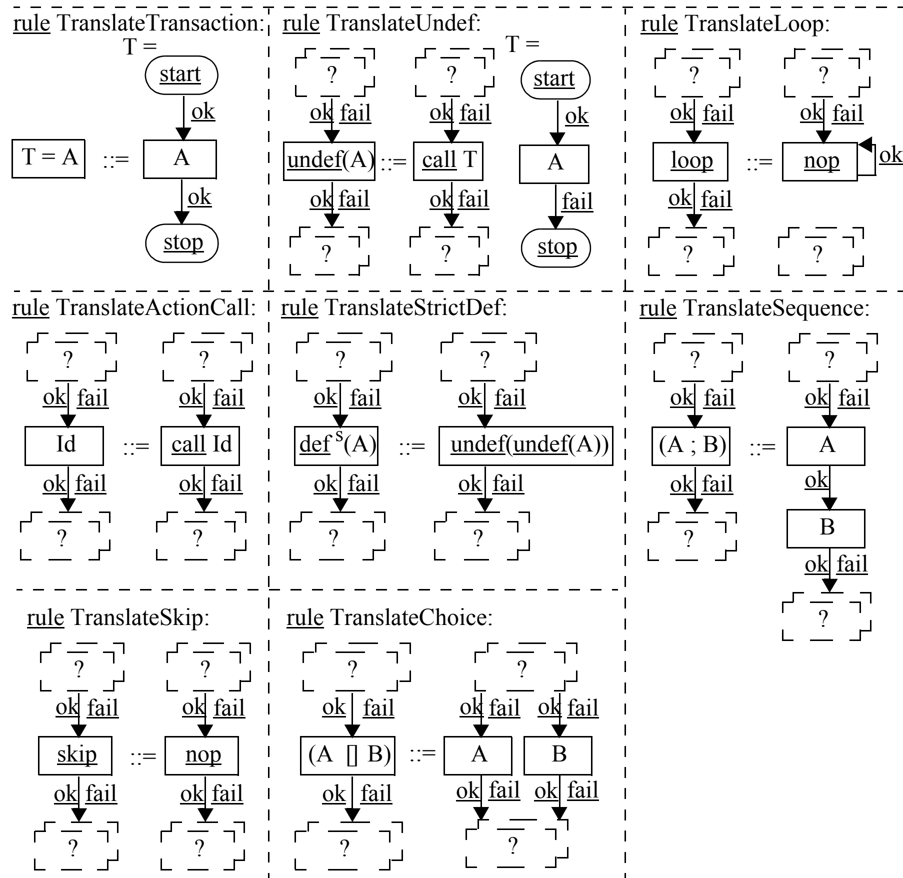

Fig. 4: Translation of graph transactions into control flow graphs

Based on the definition of BCF expressions (graph transactions) and their fixpoint semantics we were even able to relate all listed rule regulation approaches — with the exception of those presented in [12] — to the formal notation of BCF expressions:

- The last line of fig. 2 sketches the translation of simple rule priorities into BCF expressions.
- The translations of matrix grammars as well as programmed grammars in [4] are part of the running text in section 5.
- The lines 6, 7, and 8 of fig. 2 deal with control structures of programmed attributed graph grammars in [7].
- The translation of PROGRES control structures into control flow graphs was already presented in [17, 19].

	Sequential Control Flow	Nondeterm. Control Flow	Success & Failure Test	Recursive Calls	Paradigm & Notation
Simple Rule Priorities (here)	no	no	no	no	purely decl. & text-oriented
Matrix Grammars [4]	atomic sequence	no	no	no	decl./imperative & text-oriented
Programmed Grammars [4]	nonatomic sequence	no	rules only	tail recursion	decl./imperative & text-oriented
Prog. Attributed Graph Grammars [7]	nonatomic sequence	no	rules only	yes	imperative & text-oriented
Prog. Graph Rewriting Systems [5]	nonatomic sequence	yes	no	yes	imperative & text-oriented
PROGRES [19] & BCF Expressions [18]	yes	yes	yes	yes	imperative & text-oriented
Prog. Deriv. of Rel. Structures [13]	yes	yes	no	yes	parallel imper. & text-oriented
Control Flow Diagrams in [2]	yes	yes	no	no	imperative & graphical
Control Flow Diagrams in [14]	yes	yes	rules only	yes	imperative & graphical
New Control Flow Diagrams (here & [17])	yes	yes	yes	yes	imperative & graphical

Table 1: A comparison of various regulated rewriting approaches

- The lines 2 through 5 of fig. 2 sketch how programmed graph transformation systems in [5] must be handled.
- The translation of our new type of control flow graphs into BCF expressions was presented in def. 4.
- And all remaining forms of control flow graphs in [1, 2], [14], and [17, 19] are just subsets of the new type of control flow graphs.

Please note that all presented work within the previous sections had its focus on *sequential rewriting processes*. This is no longer true for so-called “programmed derivations of relational structures” in [13] which are also mentioned in table 1. This approach supports *parallel programming with rewrite rules*, but does not offer any constructs for

testing whether a complex graph transformation returns a defined result or not. Parallel composition of subtransformations as well as testing their applicability are both very valuable means and require completely different formal treatments. Parallel composition, on one hand, with its "interleaving semantics" excludes the definition of a program's semantics as a function of the semantics of its subprograms. Testing success or failure of subprograms, on the other hand, enforces the introduction of a special symbol "∞" for nonterminating computations and the usage of a rather complicated partial order for the resulting semantic domain. Further investigations are necessary to check whether a combination of both approaches is possible or not.

Finally, we should mention that all presented results in this paper are valid for any kind of rewriting approach, where single rewrite rules define binary relations over a given domain of objects. Nevertheless, we have used the terms "graph" and "graph rewrite rules" throughout this paper — instead of more generic terms — in order to emphasize the graph oriented background of our current research activities. Having developed such a common framework for rather different rule regulation mechanisms, which is independent from a specific (graph) data model or a selected (graph) rewriting approach, offers the opportunity to combine different regulation mechanisms as needed within a future *multi-paradigm graph grammar environment* GRACE [9, 10].

References:

1. Bunke H.: *Sequentielle und parallele programmierte Graphgrammatiken*, Dissertation, Technical Report IMMD-7-7, Universität Erlangen, Germany (1974)
2. Bunke H.: *Programmed Graph Grammars*, in: Proc. Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology, LNCS 73, Springer Verlag, Germany (1979), 155-166
3. Dijkstra E.W.: *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*, in CACM, no. 18, acm Press, USA (1975), 453-457
4. Dassow J., Paun G.: *Regulated Rewriting in Formal Language Theory*, EATCS 18, Springer Verlag, Germany (1989)
5. Dörr H.: *Efficient Graph Rewriting and Its Implementatiion*, Dissertation, LNCS 922, Springer Verlag, Germany (1995)
6. Engels G., Lewerentz C., Nagl M., Schäfer W., Schürr A.: *Building Integrated Software Development Environments*, in: ACM TOSEM, vol. 1, no. 2, acm Press, USA (1992),135-167
7. Göttler H.: *Graphgrammatiken in der Softwaretechnik*, IFB 178, Springer Verlag, Germany (1988)
8. Himsolt M.: *GraphED: An Interactive Graph Editor*, in: Proc. STACS 89, LNCS 349, Springer Verlag, Germany (1988), 532-533
9. Kreowski H.J., Kuske S.: *On the Interleaving Semantics of Transformation Units - A Step into GRACE*, in: Proc. 5th Int. Workshop Workshop on Graph-Grammars and Their Application to Computer Science, same volume

10. Kreowski H.J.: *Graph Grammars for Software Specification and Programming: An Eulology in the Praise of GRACE*, in: Proc. Colloquium on Graph Transformations and its Applications in Computer Science, Technical Report B-19, Universitat de les Illes Balears, Departament de Ciencies Matematiques i Informatica (1994), 55-62
11. Löwe M., Beyer M.: *AGG - An Implementation of Algebraic Graph Rewriting*, LNCS 690, Proc. 5th Int. Conf. on Rewriting Techniques and Applications, Springer Verlag, Germany (1993)
12. Manna Z.: *Mathematical Theory of Computation*, New York: McGraw-Hill, USA (1974)
13. Maggiolo-Schettini A., Winkowski J.: *Programmed Derivations of Relational Structures*, in: Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science, LNCS 532, Springer Verlag, Germany (1991), 582-598
14. Nagl M.: *Graphgrammatiken*, Vieweg Verlag, Germany (1979)
15. Nelson G.: *A Generalization of Dijkstra's Calculus*, in ACM Transactions on Programming Languages and Systems, vol. 11, no. 4, acm Press, USA (1989), 517-561
16. Schürr A.: *PROGRES: A VHL-Language Based on Graph Grammars*, in: Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science, LNCS 532, Springer Verlag, Germany (1991), 641-659
17. Schürr A.: *Operationales Spezifizieren mit programmierten Graphgrammatiken*, Dissertation, RWTH Aachen, Deutscher Universitätsverlag, Germany (1991)
18. Schürr A.: *Logic Based Programmed Structure Rewriting Systems*, appears in: Special Issue on Graph Transformation Systems, Fundamenta Informaticae, North-Holland
19. Schürr A., Zündorf A.: *Nondeterministic Control Structures for Graph Rewriting Systems*, in Proc. WG'91 Workshop in Graphtheoretic Concepts in Computer Science, LNCS 570, Springer Verlag, Germany (1992), 48-62
20. Westfechtel B.: *Revisionskontrolle in einer integrierten Softwareentwicklungs-Umgebung*, Dissertation, RWTH Aachen, Informatik-Fachberichte 280, Springer Verlag, Germany (1991)
21. Zündorf A.: *Implementation of the Imperative/Rule Based Language PROGRES*, Technical Report AIB 92-38, RWTH Aachen, Germany (1992)
22. Zündorf A.: *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungs-Systeme*, Dissertation, RWTH Aachen (1995)