Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Prof. Goos

Embedding the graph rewrite system GrGen.NET into C#

Diploma thesis by Moritz A. Kroll

August 2008

Supervisor:
Dipl.-Inform. Dr. Rubino Geiß

Responsible supervisor:
Prof. em. Dr. Dr. h.c. Gerhard Goos

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Kurzfassung

Für die Entwicklung komplexer graphersetzungsgestützter Anwendungen benötigt man eine Programmierplattform, mit der man komfortabel und typsicher arbeiten kann. Neben einer geeigneten Darstellung der Graphersetzungsregeln ist dazu aber vor allem auch die Programmierschnittstelle zwischen der verwendeten Hochsprache und dem Graphersetzungssystem von entscheidender Bedeutung.

Anhand von typischen Anwendungsfällen wird GrGen.NET bezüglich Komfort und Typsicherheit mit der bereits existierenden eingebetteten Graphersetzungssprache XL verglichen, die bereits über eine sehr gute Programmierschnittstelle verfügt. Da XL jedoch für die Modellierung von Pflanzen entwickelt wurde und die Graphelemente immer eine Verbindung zu einer "Wurzel" benötigen, ist XLs Graphmodell nicht für allgemeine Anwendungen geeignet.

Die durch den Vergleich mit XL gewonnenen Erkenntnisse werden dann zuerst dazu verwendet, die Programmierschnittstelle von GrGen.NET in Angriff zu nehmen und bezüglich Komfort sowie Typsicherheit deutlich zu verbessern. Mit der darauf aufbauenden Einbettung (Embedding) von domänen-spezifischen Konstrukten von GrGen.NET (⇒ Domain-Specific Language (DSL)) in die Programmiersprache C# erhält man eine universelle Programmiersprache mit einem übersichtlichen Zusammenspiel zwischen Graphdatenhaltung, -ersetzung und -manipulation. Insbesondere können damit Variablen aus dem C#-Kontext in Graphersetzungsregeln verwendet werden; auch die umgekehrte Richtung ist möglich. Der Nutzen dieser Einbettung wird anhand von Beispielen aus dem Übersetzerbau gezeigt.

Contents

Chapter 1

Introduction

The basic task of a graph rewrite system[1] is to apply graph rewrite rules, which search for a pattern (left-hand-side of the rule, LHS) in a graph (called host graph) and replace it by another pattern (right-hand-side of the rule, RHS). The patterns are described using nodes, edges and possibly other properties like type and attribute conditions or negative application conditions. A graph model[2] specifies the type hierarchy of the node and edge types including their attributes and may also describe some parts of the structure of the graph.

Complex applications whose core data is naturally represented as graphs are good candidates for benefiting from the concise and declarative specifications of graph rewrite rules. However, other parts of the program like database communication or a GUI may be more suitably formulated in an imperative general purpose language. But as most general purpose languages like Java and C# are very limited in their syntax, it is not possible to use the domain-specific notations of the graph rewrite system to describe graph models and patterns without the use of external files. This does not only lead to very scattered code which is difficult to understand, but may also make the code less type-safe if the API of the graph rewrite system is inappropriate, as is the case with GrGen.NET 1.3.1.

In this work I develop a convenient programming interface between the graph rewrite system GrGen.NET and the object-oriented general purpose language C#. As a part of this, the API of GrGen.NET 1.3.1 has been progressively improved and implemented in the versions 1.4 and 2.0. To gain first insight into the problems, two motivating examples are presented after a short introduction to GrGen.NET.

1.1 GrGen.NET

GrGen.NET is an application domain neutral graph rewrite system based on the SPO approach [GBG+06, Kro07, Gei08]. It uses attributed and typed multigraphs with multiple inheritance on node and edge types and offers an expressive and easy to learn rule

[1]Also called graph transformation system.

[2]Also called meta-model.

specification language. In spite of its expressiveness it is one of the fastest automatic graph rewrite systems [SNZ08] with the help of search plan driven graph pattern matching [Bat06, BKG08], which can also be used to adapt the pattern matching to changing structures of the graph. To quickly develop simple graph rewrite applications, GrGen.NET offers a high-level interface to programmed rule application called eXtended Graph Rewrite Sequences (XGRS) supporting logical and iterative sequence control and nested transactions. Although the rule applications together with the XGRSs are Turing-complete and even have been used to develop a simple compiler [Den07], they have not been designed for complex applications forcing the developer back to the API of GrGen.NET.

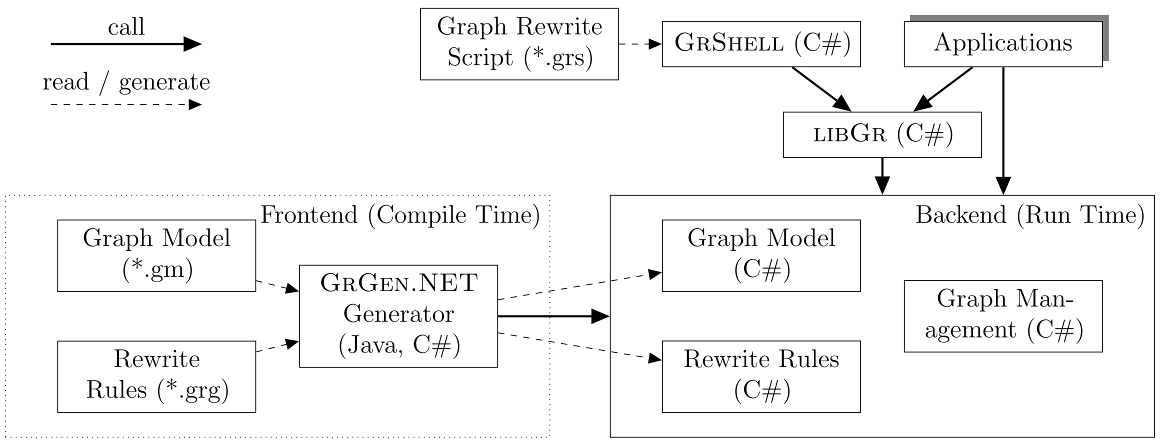

Figure 1.1: GrGen.NET system components

Figure 1.1 shows the basic architecture of GrGen.NET. The graph models are specified in .gm files, while the graph rewrite rules reside in .grg files referring to the graph models. The GrGen.NET Generator, written in Java and C#, compiles them to .NET assemblies which form a graph backend when combined with a graph management library. Working on the backend, libGr is a generic graph API which offers some important C# interfaces:

• *IGraphModel*:	A graph model.
• *ITypeModel*:	A set of node or edge types.
• *IType*:	A node or edge type.
• *IGraph*:	A graph consisting of nodes and edges of a graph model.
• *IGraphElement*:	A node or an edge.
• *INode*:	A node.
• *IEdge*:	An edge.
• *IAction*:	A graph rewrite rule.
• *IActions*:	A set of graph rewrite rules.
• *IMatch*:	A match of a pattern consisting of nodes and/or edges.
• *IMatches*:	A set of matches.

The backend provides implementations for these interfaces, the most important ones being *LGSPGraph* for *IGraph*, *LGSPNode* for *INode*, and *LGSPEdge* for *IEdge*. While the GrShell, GrGen.NET's command line shell, only uses the generic libGr API, user applications may also access the generated assemblies and their types directly to avoid querying for the objects by their names. So the generated assemblies are also part of the API.

1.2 Motivation

Developing graph transformation based applications with the API of GrGen.NET 1.3.1 is tedious at times, as the following two basic examples show.

Example: Creating and Initializing a Node

If you want to create a node of some type *WriteValue* and initialize its attribute *value* with 7, you have to write the following code (given you already have a graph *graph* of the according model):

```
LGSPNode node = graph.AddNode(NodeType_WriteValue.typeVar);
((INode_WriteValue) node.attributes).value = 7;
```

This small code fragment already reveals several problems:

- Class names generated from the model specification are always prefixed to avoid name clashes. This leads to unnecessarily long type names, even if the user does not need them as e.g. the model type names are already self-explaining.
- To access an attribute you first have to read the *attributes* field of the graph element (member of *LGSPNode* and *LGSPEdge*) and then cast it to the appropriate element type. This is not only unacceptable for users of the API but also requires many casts leading to slow attribute access. It reveals implementation details of the library which are of no interest to most users.
- Type safe handling of graph elements is not possible as only the *attributes* fields has a special type. When whole graph elements (element + attributes) are needed, only the general interfaces *IGraphElement*, *INode* and *IEdge* or the general backend implementation classes *LGSPNode* and *LGSPEdge* can be used.

Example: Simple Pattern Matching

Consider another very basic example where you have a node n of type *Process* and want to find an outgoing edge e of type *Request* to a node of type *Resource* whose *amount* attribute is greater than some not statically known value i. If there is no such edge, e should be null at the end. With GrGen.NET 1.3.1 you have two reasonable choices:

Using a graph rewrite rule and the API: Before GrGen.NET 2.0 rules are not able to take non-graph-element parameters, so you have to use a dummy graph element with an accordingly typed attribute to pass values to rules. As no modification of the graph is needed here, a test can be used instead of a rule, which is just a graph rewrite rule with the RHS implicitly declared as the LHS without application conditions:

```
test GetResource(n:Process, dummy:Dummy) : (Resource)
{
    n -:Request-> res:Resource;
    if { res.amount > dummy.value; }
    return (res);
}
```

A test has no modification part, but internally the return statement belongs to one. In order to get the return value, first the *Match* method must be invoked on the according action object representing the test and then the *Modify* method with a match returned by the former method. Although the 1.3.1 API already had a convenience function *Apply* to match and modify at once, it did not support return values, yet. So the needed C# code for this example looks like this:

```
    IEdge e = null;
    ((INode_Dummy) dummy).value = i;
    LGSPAction action = Action_GetResource.Instance;
    IGraphElement[] pars = new IGraphElement[] { n, dummy };
    LGSPMatches matches = action.Match(graph, 1, pars);
    if(matches.Count != 0)
    {
        IGraphElement[] rets =
            action.Modify(graph, matches.matches.First);
        e = (IEdge) rets[0];
    }
```

This code shows two more places of type unsafety: The parameters and the return values. The unspecific parameter array *pars* has to be filled according to the order of parameters given in the specification of the test. The return array *rets* works analogously. Only at runtime the program fails if the types did not match the signature in the rule specification file.

Using the API only: Another way to get the edge is to match the pattern manually via the API:

```
    IEdge e = null;
    foreach(IEdge curEdge in n.GetCompatibleOutgoing(
            EdgeType_Request.typeVar))
    {
        INode target = curEdge.Target;
        if(!target.InstanceOf(NodeType_Resource.typeVar)) continue;
        if(((INode_Resource) target.attributes).amount <= i) continue;
        e = curEdge;
        break;
    }
```

Compared to the first choice, this is not only shorter, but also much easier to understand. There is no need for a dummy element and no chance to mix up any array indices. But the graph pattern is not clear at first sight anymore and the code is still much more verbose than the rule specification. When it comes to bigger patterns, this way becomes inacceptable for several reasons:

- The developer is responsible for choosing a suitable order for searching the pattern elements, which is not a trivial task for complex patterns. Also he cannot benefit from GrGen.NET's search plan generation to adapt the search strategy to changes in the graph structure.
- Small changes in the pattern may require non-trivial changes to the code to preserve expected performance.
- The expressiveness of the code is very low compared to the rule specification, so there are many possible errors, which could not even be formulated in the rule specification in the first place.
- The obvious API functions for searching a pattern like *INode.GetCompatible-Outgoing* are slow as they require C# enumerators which use closures to maintain their state. The code generated by GrGen.NET is much more efficient for larger patterns because it avoids the enumerators by using low-level API fields. But the resulting code should not be written by hand as it is even more error prone and less maintainable.

1.3 What can be done?

The above examples exposed that the API of GrGen.NET 1.3.1 has some severe problems with type safety, which also negatively affects its usability. So the first important step should be to improve the API accordingly (see section 4.1). Then, there are three options:

Live with the separation of declarative rule specification and imperative C# code. With an improved API the rule applications could just look like method calls. But if you just want to find one edge, the overhead of the declaration of an according pattern and the method call passing the needed context explicitly as parameters is quite huge compared to the pattern itself.

Specify patterns as strings passed together with the needed context to some API function. Although this gives you the locality, it not only takes away type safety inside the rule specifications but also static syntax checking. Only when a rule is executed at runtime, errors will be reported. To circumvent this, a preprocessor could extract all rules at compile time and check them, but to also check the types of the parameters, a type analysis of the host language would be required, making up a good part of a compiler.

Embed patterns into C# as part of some new specialized statements. This way we get locality and type safety and do not have to specify the needed context anymore because it can be done implicitly by the used identifiers. On the downside this requires to develop a full compiler processing the new statements in addition to C#.

Since embedding patterns into C# provides the highest convenience and productivity advantage, this option was chosen for this work, resulting in the new programming language G#. As proof of concept the Mono C# compiler [Mon08a] was extended to support some of the new constructs.

Outline

The rest of this work is structured as follows: In chapter 2 we have a look at related work, chapter 3 consists of an analysis of typical scenarios of the development of an application which makes use of graph rewriting, chapter 4 contains a discussion about possible improvements of GrGen.NET and introduces the special language constructs of G#, and in chapter 5 I describe how a compiler for the new language can be realized by extending the Mono C# Compiler. Finally in chapter 6 the improvements achieved by this work are shown and it is discussed about how the performance of GrGen.NET has changed over several versions released while writing this work, and in chapter 7 I conclude. Appendix A contains a definition of the newly introduced language constructs, appendix B contains some examples working with the current implementation of the developed compiler.

Chapter 2

Related Work

In this chapter we have a look at some related work to see whether it can be of any help to the goals of this work. At first we investigate, whether it is possible to specify graph patterns satisfactorily in C# 3.0 using LINQ or with the help of a simple macro expanding preprocessor. Then two embedded domain-specific languages (EDSLs) are presented as examples, namely Embedded SQL and the graph transformation language XL. At the end we consider three approaches to implement an EDSL for our case.

2.1 Simple Approaches

In this section I argue about two attempts to realize a convenient and type-safe programming interface between graph rewriting and C# without the need of an extended C# compiler.

2.1.1 LINQ

LINQ (.NET Language Integrated Query [BH05]) is an extendable, declarative, general-purpose query facility with compile-time syntax checking and static typing introduced with C# 3.0. A query can filter and calculate complex information from “any `IEnumerable<T>`-based information source” [BH05]. Considering a graph as information source there are enumerables for nodes or edges with given types, and for incoming and outgoing edges with given types from given nodes. The standard query operator ”*where*” filters data according to a predicate. For graph rewriting such a predicate should be able to represent a graph pattern. With the help of lambda expressions, the GrGen graph rewrite rule from listing 2.1 could be represented as shown in listing 2.2. This lambda expression results in an expression tree, which could be translated at runtime to a real graph rewrite rule. Because assignments — and thus variable declarations and initializations — are not allowed in expression trees, all variables and their types have to be specified as parameters of the function representing the expression tree[1]. By using special functions (in this example

[1]The last type parameter of *Func* is the result type of the expression. Here it is used without any special semantics in mind.

Listing 2.1: An example GrGen rule utilizing several syntactic features

```
rule linqRule(Proc:Process, Res:Resource) : (Process)
{
    hom(Proc, Res);
    Proc -req:request- Res <-par:parent-> Proc;
    if { Res.UseCounter > 5; }
    negative {
        Proc <-:request- Proc;
    }
    modify {
        delete(req);
        newProc:Process -:request-> Res;
        eval {
            Res.UseCounter = Res.UseCounter + 1;
        }
        return (newProc);
    }
}
```

param, *hom*, *negative*, *modify*, *assign*, and *returns*), which always have to be declared in the current scope, most if not all features of GrGen can be syntactically emulated. With the help of some special operator overloads for Minus and XOR, and some implicit casts, the expression representation is even internally type-safe. But without a real signature, the actual parameter and return types are unclear at first sight and cannot be checked statically. To use such a graph rewrite expression, we have to use general methods with non-type-safe parameters and return values. So although we get the locality with syntax similar to the GrGen syntax, the type declarations of elements are very cumbersome and we still do not get a type-safe programming interface between the graph rewrite rules and C#.

2.1.2 C Preprocessor

The C preprocessor is a simple macro expansion tool. Macro definitions can have any number of parameters and may refer to previously defined macros. As the macro expansion does not account for the C syntax, the context of a macro usage, such as the current scope and the types of macro arguments, is not considered. Macro names may only be formed out of characters, numbers and underscores, so special characters to represent e.g. edges would not be possible. A representation of a graph pattern with macros would at most abbreviate the definition of a pattern using the API a bit, so it is not suitable for our goals.

2.2 Embedded Domain-Specific Languages

Embedded domain-specific languages (EDSLs) are languages which have been extended by a domain-specific language (DSL) to improve the productivity when working in the

Listing 2.2: A lambda expression to represent the GrGen rule from listing 2.1

```
Expression<Func<Process, Resource, request, parent, request, Process, request, bool>>
    linqExp = (Proc, Res, req, par, noReq, newProc, newReq) =>
        param(Proc, Res) &&
        hom(Proc, Res) &&
        Proc -req- Res ^par^ Proc &&
        Res.UseCounter > 5 &&
        negative (
            Proc ^noReq- Proc
        ) &&
        modify(
            delete(req) &&
            newProc -newReq^ Res &&
            assign(Res.UseCounter, Res.UseCounter + 1) &&
            returns(newProc)
        );
```

domain of the DSL while keeping the full expressiveness of the host language (mostly a general-purpose language (GPL))[2]. As this is exactly what we want with C# as the host language, in this section we have a look at the most famous EDSL, Embedded SQL, to get a basic understanding of what an EDSL is and at the Java-based graph rewriting language XL to see what level of convenience has already been reached in our profession.

2.2.1 Embedded SQL

Embedded SQL is an ANSI standard from 1989 which partially defines the embedding of SQL statements into Ada, C, COBOL, FORTRAN, Pascal, and PL/I [Sau02]. In 1992 the definition was completed by the ISO SQL-2 standard. The embedding brings several benefits:

- An abstraction of the different SQL library APIs.
- Few additional language constructs, thus easy to learn (as long as SQL is known).
- SQL injection can partially[3] be prevented at compile time.
- Easy implementation of a compiler (see below).

The embedded statements always begin with `EXEC SQL` followed by an SQL instruction like `SELECT`, `INSERT`, or `CREATE`. To interact with the rest of the program, these instructions

[2]There are very different definitions of EDSLs. Here I stick with the interpretation used by MetaBorg [BV04], a method for implementing EDSLs (see also section 2.3.1).

[3]SQL injection may still be possible, when dynamical SQL statements for use with the `EXECUTE` or `PREPARE ... FROM` command are built by concatenating commands and data carelessly without the use of SQL parameters.

can read and write specially marked variables of the surrounding program, so called *host variables*. Processing multiple rows resulting from an SQL `SELECT`-query is done in an iterative way using *cursors*. Errors, warnings or non-applicable statements which might occur at any SQL statement can be handled via registered callbacks or loop breaks. A precompiler translates the embedded SQL program into the host language, which can then be compiled using a normal compiler for the language.

Listing 2.3 shows a small example with C as the host language printing the columns `a` and `b` of all rows from the table `tab` for which `a` is greater or equal to 12. As this example shows, the SQL statements do not fit to the style of the language. In C no two keywords follow each other and blocks are always written with braces. Also the declare sections appear unnecessary.

There are probably two reasons for this: Firstly, Embedded SQL was meant to be applied to many languages with next to no changes, so people would know how to work with it in any language, when they learned it for one. Secondly, the simple syntax allows to use a compiler which only has to remember a very small context (the declared host variables) and can directly emit host source code while parsing each line of the input source code as no further analysis (especially no type analysis) is required. If the line does not start with “EXEC SQL” and it is not inside a “DECLARE SECTION”, the line can just be written out without modifications.

Listing 2.3: Embedded SQL in C example

```
EXEC SQL BEGIN DECLARE SECTION;
int val_a, val_b, min_a = 12;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur CURSOR FOR SELECT a, b FROM tab WHERE a >= :min_a;
EXEC SQL OPEN cur;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
while(1)
{
    EXEC SQL FETCH NEXT FROM cur INTO :val_a, :val_b;
    printf("%i, %i\n", val_a, val_b);
}
EXEC SQL CLOSE cur;
```

2.2.2 XL

XL [Leh08] is a Java-based modelling language which offers concise formulations of graph rewriting rules and graph queries. It uses Lindenmayer systems which have been extended to rooted graphs with typed, attributed nodes and colored, directed simple edges. If a node is not weakly connected to the root (anymore), it is not part of the graph and can therefore not be matched as part of a graph rewrite rule. The nodes represent Java objects, whereas edges are not much more than 32-bit integers. Limited support for multiedges and multiple inheritance on edge types can be realized by interpreting the bits of the integer as edge indicators of different edge types. Multiple inheritance on node types is only indirectly possible using interfaces. While GrGen.NET has built-in support for multiple inheritance, in XL the developer has to manually implement the interfaces in the according classes. As an undocumented feature, it is possible to "abuse" node types to instantiate edges, which not only makes it possible to have typed, attributed edges but also to have multiedges. But it is not possible to use a name for an edge in a graph pattern, so it is unclear how its attributes can be accessed.

While in chapter 3 some more details about XLare presented, listing 2.4 tersely illustrates some of its features:

Listing 2.4: XL syntax example adapted from [Leh08]

```
 1 public void derivation() [
 2     Axiom ==> F(1) RU(120) F(1) RU(120) F(1);
 3     F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
 4 ]
 5
 6 public int distance(Individual a, Individual b) { return 1; }
 7 public boolean isRelevant(F f) { return true; }
 8
 9 public void printLength(Individual c) {
10     System.out.println(sum(
11         (*
12             d:Individual,
13             ((d != c) && (distance(c, d) < 2)),
14             d ( -(branch|successor)-> )* f:F,
15             (isRelevant(f))
16         *).length));
17 }
```

Graph transformations are specified in so called transformation blocks delimited by brackets ("[...]") which either act as a statement or as a whole method body like in lines 1–4. Each transformation rule is given by a left hand side (LHS), a right hand side (RHS), and a production operator indicating a mode of the transformation. In line 2 all nodes of type *Axiom* are replaced by a sequence of *F* and *RU* nodes implicitly connected by *successor* edges. The production operator "==>" indicates that implicit embedding should be used, i.e. all incoming/outgoing edges of the textually first/last node of the LHS are to be connected to the textually first/last node of the RHS, respectively.

The LHS of a rule is a search pattern which can also contain any Boolean expressions as conditions enclosed in parentheses including function calls (see lines 13 and 15) or transitive closures of binary relations (see line 14 matching a path made of any combination of branch and successor edges leading from d to a node of type F).

In the Java context the LHS of a rule can also stand for itself as a so called *query expression*. The query expression in lines 11–16 yields all nodes f (as it is the textually last node mentioned in a pattern context ("f:F")) fulfilling this pattern. The result of a query expression can be given to a new kind of `for` loop, which iterates over all yielded values, or to two new types of methods: *aggregate methods* compute a single value from the yielded values, whereas *filter methods* are used to yield only a part of them. By using the

aggregate method *sum* in line 10 in conjunction with line 16 the *length* attributes of all yielded F nodes are added up.
Summarizing the advantages of XL, we see:

- A very concise syntax with embedded graph transformations and queries yielding a high productivity,
- aggregate methods abbreviating loops over graph query results for common and user-defined tasks,
- graph element constructors for data initialization directly at element creation,
- support for object-oriented development by allowing the user to add methods to node types,
- and (undocumented) visited flags.

But for the development of complex applications there are also several disadvantages:

- Due to the rooted graphs the developer has to be very careful when he deletes graph elements. If he deletes the wrong element, the whole graph may vanish. Or he has to make sure, that each element is connected to the root node, which further complicates the graph rewrite rules.
- Edges cannot be referred to in graph rewriting rules and queries, so even with the nodes abusing trick it is not possible to access the attributes without using the low level API.
- XL uses many unintuitive and cryptic constructs which makes learning XL very difficult.
- As the syntax of XL bases on Java 1.4, it does not support generics which have become an important tool against code duplication and type-insafety.

Because of these disadvantages XL seems to be inappropriate for general complex applications utilizing graphs in this form and thus legitimates the development of the solution presented in this work.

2.3 Ways to Implement an EDSL

In this section I discuss some ways of implementing an EDSL with C# as a host language and explain which way has been chosen for this work.

2.3.1 MetaBorg

"MetaBorg" introduced in [BV04] is "a method for providing concrete syntax for domain abstractions to application programmers" [BV04]. It uses SDF [SDF08], a modular syntax definition formalism, and Stratego/XT [Str08], a language and toolset for program transformation, to embed a domain-specific language into a general-purpose language and to assimilate the embedded code into the host language (source to source transformation). With SDF it is even easy to integrate multiple different extensions into one host language as described in [BV07], as long as you have the according SDF grammars. Sadly, a C# grammar is not available for SDF, yet, whose creation would probably have taken too much time for this work, considering the ambiguous grammar and the required type attribution.

2.3.2 C# Parser

"C# Parser" supplied at [DE08] is a fast, handwritten C# parser, which supports writing the abstract syntax tree back to a C# source file. With the extensions to support the G# language it would have been possible to also do source to source transformations. But there were several problems: The parser was barely able to parse C# 2.0. After some work on improving this situation until it passed all test cases of the Mono C# compiler, I realized that the associativity of the expressions was wrong in most situations. Christoph Mallon and I then reimplemented the expression and statement parser, so that all test files of the Mono C# compiler passed even with explicit marking of associativity using parentheses in the unparsed output. While working on this parser and developing the G# language, I began to realize that some type analysis would be required for a semantic analysis of the new constructs to ensure type-safety. But a type analysis was not even partly implemented in the parser. And considering that the project was already dead since May 2007[4], this project finally also failed to qualify for this work. Newer versions of C# like the current 3.0 would probably never be supported by it.

2.3.3 The Mono C# Compiler

Finally the choice fell on the C# compiler of Mono 1.9 [Mon08b], because Mono is an actively developed project supported by a large community and a big sponsor (Novell). According to [Mon08a] it fully supports C# 2.0, while the C# 3.0 features are still being developed. In contrast to the C# parser project, as a full compiler it provides type analysis usable for semantically checking the new constructs. But on the downside it is rather slow and provides no way of emitting the processed source code back into a file making debugging the compiler more difficult. The .NET decompiler *Reflector* [Lut08] attenuates the latter drawback, as the generated code can be easily inspected in form of C# code. Also the compiler is not designed to be extended, further complicating the implementation

[4]Recently (in August 2008), this project has been revived and our changes have been applied to the project. So it was not a complete waste of time after all.

as a G# compiler. Still I considered this the only chance to get a working implementation after I had already lost so much time on the “C# Parser”-project.

Chapter 3

Scenario and Problem Analysis

In this chapter several typical scenarios for graph application development are presented. For each scenario, implementations using GrGen.NET 1.3.1 are discussed and compared to solutions written in XL, the previously only embedded graph rewriting language known to me, to find out advantages and disadvantages of both systems. Although the presented scenarios were chosen to be general and representative, there may be some typical scenarios I missed. But by basing the language developed in this work on a general purpose language, it would still be possible to solve these scenarios. Just not necessarily as convenient as it could be, if special support was available.

The examples in this chapter are based on the description of the AntWorld case study proposed by Albert Zündorf for the GraBaTs 2008 tool contest [Zü08]. Although it is just a toy example, it is able to cover most scenarios. Because of its simple graph model, the scenarios can be easily described. Chapter 6 contains an evaluation of the results on the basis of a real world application.

3.1 Model Specification

The first thing we need when developing a graph rewrite application is a graph model (also known as meta model). It describes the type hierarchy of the graph elements, their attributes, and in some systems enforced or unenforced assertions about the structure of a graph model instance (i.e. the structure of a graph).

As an example consider a graph which contains a quadratic grid of *GridNode*s connected by *GridEdge*s with additional diagonal edges towards the center. All (more or less) radial edges are special *GridEdge*s called *PathToHill*. The center of the grid shall be an *AntHill* modelled as a special *GridNode*. For each *GridNode* we want to know how much *food* and *pheromones* it contains, which should be initialized as empty. *AntHill* nodes however start with 8 *food* units. Since an ant hill without ants is missing something, we also have *Ant*s located on the grid, who may carry some food and whose positions are modelled by *AntPos* edges pointing to the according *GridNode*. To define an order on the *Ant*s, we connect them via a singly linked list using *NextAnt* edges.

GrGen.NET: GrGen.NET's graph models are specified in .gm files. Although they support assertions about the structure of a graph in form of so called *connection assertions*, they are too simple to exactly specify the above described graph model, because they can only describe the local structure of a graph. E.g. it is not possible to state, that there is exactly one list of *Ant*s or that the grid is really quadratic. So the following graph model specifies a super set of the one described above:

```
node class GridNode {
    food:int;
    pheromones:int;
}
node class AntHill extends GridNode {
    food = 8;
}
node class Ant {
    hasFood:boolean;
}
edge class GridEdge connect GridNode[*] --> GridNode[*];
edge class PathToHill extends GridEdge connect inherited;
edge class AntPos connect Ant[1] --> GridNode[*];
edge class NextAnt connect Ant[0:1] --> Ant[0:1];
```

The interpretation of this graph model is straightforward: *GridNode* is a node type with the two integer attributes *food* and *pheromones*. *AntHill* is a subtype of *GridNode* and initializes the *food* attribute with 8 instead of the default value 0. For (virtual) multiple inheritance on node or edge types more than one super type separated by commas can be added behind `extends`. *GridEdge* is an edge type which may connect *GridNode*s without restrictions on the out and in degree. Its subtype *PathToHill* inherits the connection assertions from *GridNode*. Ants are represented as *Ant* nodes with a Boolean attribute *hasFood*. Their positions are modelled as *AntPos* edges from each *Ant* to the according *GridNode*. While any number of *Ant*s may stand on one *GridNode* (unlimited in-degree `[*]` which is equivalent to `[0:*]`), one *Ant* stands on exactly one *GridNode* (out-degree `[1]` which is equivalent to `[1:1]`). The connection assertions are not enforced but can be validated on demand to check the integrity of the graph. If the expressiveness of the connection assertions is not sufficient, a Turing-complete mechanism can be used applying an XGRS to a copy of the graph. The graph is then said to be valid, if the XGRS succeeds.

If the filename of the above listing is “AntWorld.gm”, a model class with the name *de.unika.ipd.grGen.models.AntWorld.AntWorldGraphModel* will be generated. But an actual graph is represented by a general *LGSPGraph* instance whose *Model* property just points to an instance of the according model class. Thus the compiler cannot statically distinguish graphs of different models (not type-safe) and no convenience methods for creating graph elements of the according model can be made available. It is possible though to manually define subclasses of *LGSPGraph* using the according models to achieve this.

XL: XL neither has a concept of explicit graph models nor of constraints on the graph structure. Any Java class inheriting from *de.grogra.graph.impl.Node* can be used as a node type. All other types, like `String`, `int`, or a class not inheriting from the node implementation class, are implicitly wrapped when used in graph patterns, and thus can also be used as node types. Edge types are just bit masks represented by integer constants[1].

The simplest way of programming with XL is using an .rgg file: It holds the whole project in an implicit class derived from the XL API class *RGG*, whose instances represent graphs. In such a file, the element types required for our example could be declared like this:

```
class GridNode {
    int food, pheromones;
    GridNode() {}
    GridNode(int food) {
        this.food = food;
    }
}
class AntHill extends GridNode {
    AntHill() {
        super(8);
    }
}
class Ant {
    boolean hasFood;
}
const int RealGridEdge = EDGE_0;
const int PathToHill = EDGE_1;
const int GridEdge = RealGridEdge | PathToHill;
const int AntPos = EDGE_2;
const int NextAnt = EDGE_3;
```

The Java classes *GridNode*, *AntHill* and *Ant* can be used as node types, while the *RealGridEdge*, *PathToHill*, *GridEdge*, *AntPos*, and *NextAnt* constants can be used as edge types. The *EDGE_n* constants represent single bits of the edge integers, which can be used by the user. When searching for a *GridEdge* between two nodes, the corresponding edge integer is masked with *GridEdge* and such an edge is said to be found, when the result is not null. So this would be true for both a *RealGridEdge* and a *PathToHill* edge. On the other side, when we want to create a real *GridEdge* (i.e. not a *PathToHill* edge), we may not create a *GridEdge*, but must use a *RealGridEdge* as it does not contain the bit of *PathToHill*.

The *GridEdge* could also be interpreted as an edge type inheriting from both *RealGridEdge* and *PathToHill*. To search for an exact *GridEdge* between two nodes a and b, the pattern "`a -GridEdge-> b, (a.getEdgeBitsTo(b) == GridEdge)`" would have to be used, which is very inconvenient to write.

[1]Keep in mind, that XL does not support real edge attributes.

To simplify the declaration and usage of node types, XL provides special syntax for declaring them, so called *modules*:

```
module GridNode(int food) {
    int pheromones;
}
module AntHill(super.food) extends GridNode {
    { food = 8; }
}
module Ant(boolean hasFood);
const int RealGridEdge = EDGE_0;
const int PathToHill = EDGE_1;
const int GridEdge = RealGridEdge | PathToHill;
const int AntPos = EDGE_2;
const int NextAnt = EDGE_3;
```

The first three lines declare the $GridNode$ node type with the integer attributes $food$ and $pheromones$ and the constructors $GridNode()$ and $GridNode(int\ food)$. The following three lines declare the $AntHill$ node type as a subtype of $GridNode$ with the constructors $AntHill()$ and $AntHill(int\ food)$ where the latter is supposed to set the $food$ attribute to the given value. But as we want the initial value of $food$ for an $AntHill$ to always be 8 and are too lazy to use the second constructor, we have to initialize $food$ in an instance initialization block, what effectively makes the $AntHill(int\ food)$ constructor useless for the moment. Still the module version of $GridNode$ is more concise than the pure Java approach.

Another advantage of using the module declarations is, that you can use the specified constructor not only for constructing new elements of this type, but also for matching them. Writing $AntHill(5)$ on the LHS of a rule would only match $AntHill$ nodes with the $food$ attribute being 5. On the other hand $AntHill(f)$ on the LHS matches any $AntHill$ node and assigns the value of the $food$ attribute to a new local variable f, if f does not exist as a local variable, yet. If such a local variable already exists, the node is restricted to those with the $food$ attribute value being f. To restrict the $food$ attribute to the value of a field f, one would have to use an expression which is not a single identifier like $AntHill((f))$ or $AntHill(this.f)$.

While modules can implement any number of interfaces using the "implements" keyword, they cannot extend more than one other module. This makes multiple inheritance on node types much more verbose, as the interfaces have to be implemented manually in all according modules. Of course, the same applies to the Java class way of node types. But as modules are transformed to Java classes, it is also possible to declare methods in them well supporting object-oriented development.

As mentioned above, a graph is represented as an *RGG* instance. If multiple (different) graphs are required, you should probably[2] use an .xl file, which is similar to an .rgg file

[2]It is unclear, if this is the best way to do it. The XL distribution does not contain an example using multiple graphs, just like GrGen.NET.

but has neither implicit embedding into an *RGG* subclass nor implicit imports. Thus you can manually define different *RGG* subclasses to be able to statically distinguish between different types of graphs. But it is not possible to state which element types are allowed in which graph type. All element types are allowed in any graph type, so it is type-safe with respect to different graphs. However, due to scoping you would have to qualify an element type of another graph type inside a pattern, so it is not too easy to make a mistake here.

Considering the GrGen.NET graph models and the XL modules, we can extract these advantageous features:

- An explicit graph model (GrGen.NET)
- Straightforward support of (virtual) multiple inheritance (GrGen.NET)
- Constructors usable in both patterns and rewrite parts (XL)
- Element methods (XL)
- Support for graph structure assertions (GrGen.NET)

3.2 Rule Specification

With the graph model defined, we can now specify rules to work on the graph. Generally a graph rewrite rule describes a pattern to look for (the left-hand-side, LHS), another pattern which shall replace the LHS (the right-hand-side, RHS), and possibly a mode, how this rule is to be applied.

As an example consider we have a given *Ant* which should go one step towards the hill, if it carries food and place a given number of units of pheromones at the old position.

GrGen.NET 1.3.1: Graph rewrite actions are specified in .grg files. GrGen.NET knows two kinds of actions: Rules and tests. Rules are normal graph rewrite rules with a LHS and a RHS, whereas tests only have a LHS and are meant to only check for the existance of a pattern. Although both action kinds can receive and return any number of graph elements (as specified by the declaration, of course), they do not support values like integers, strings, or object references. So in order to pass the number of pheromones to an action, we have to use a wrapper graph element. Here we will use a new node type *DummyInt* with the integer attribute *value*.

A rule for this example is shown in listing 3.1. The rule named *GoHome* with the parameters *curAnt* of type *Ant* and *dummy* of type *DummyInt* consists of a LHS made of one attribute condition (line 3) and one graphlet (line 4), and the RHS which modifies the LHS (lines 6 – 12). The LHS says, that the *hasFood* attribute of *curAnt* must be true and that *curAnt* must be connected via an *AntPos* edge to the old *GridNode* which must have a *PathToHill* edge to the *GridNode* the *Ant* has to

Listing 3.1: The example rule written in the rule specification language of GrGen.NET.

```
rule GoHome(curAnt:Ant, dummy:DummyInt)
{
    if { curAnt.hasFood; }
    curAnt -oldPos:AntPos-> old:GridNode -:PathToHill-> new:GridNode;

    modify {
        eval {
            old.pheromones = old.pheromones + dummy.value;
        }
        delete(oldPos);
        curAnt -:AntPos-> new;
    }
}
```

move to. If the LHS is found, the RHS will add *dummy.value* units of pheromones to the old *GridNode* and transfer the *Ant* from the old to the new *GridNode*, by deleting the old *AntPos* edge and creating a new one between *curAnt* and *new*. So again, this is straightforward.

Instead of a "modify" part, also a "replace" part could be used where all elements of the LHS not mentioned on the RHS are deleted.

XL: As mentioned in the previous chapter, in XL rules are specified inside *transformation blocks* delimited by brackets ("[...]") acting as normal Java statements. All rules in a transformation block are applied in a way, as if they were executed simultaneously. All matches and the according changes are saved, but do not take effect until a so called *derivation step* is performed, which is explained in section 3.3.

A rule consist of a graph query specifying the pattern (LHS), a production operator determining the mode of the rule, and so called *graph statements* defining the replacement (RHS). XL supports three production operators:

"==>>": This declares a normal rule, where the LHS is replaced by the RHS with SPO-like semantics (remember that XL uses rooted graphs).

"==>": This results in a normal rule with implicit embedding, which means, that all "incoming (resp. outgoing) edges of the textually leftmost (resp. rightmost) nodes of" the LHS are connected "to the textually leftmost (resp. rightmost) nodes of" the RHS ([KKBS05]).

"::>": An execution rule executes a statement given as the RHS for each found match of the LHS.

The above example could be implemented as shown in listing 3.2. It declares a normal Java method *GoHome* with the parameters *curAnt* of type *Ant* and *incrPher* of type *int*. The braces of the method body block can be omitted when it would only contain

Listing 3.2: An XL version of the example rule.

```
1 public void GoHome(Ant curAnt, int incrPher)
2 [
3     (curAnt.hasFood)
4     curAnt -AntPos-> (* old:GridNode -PathToHill-> next:GridNode *)
5     ==>>
6     curAnt -AntPos-> next { old.pheromones += incrPher; };
7 ]
```

a transformation block. The transformation block consists of a single rule without implicit embedding. Line 3 in the transformation block is a condition predicate which must evaluate to true for the LHS to match. As such a predicate is an expression, it can also contain any method calls. While this can be very convenient, it may cause unexpected behaviour if the called methods have side-effects. Line 4 describes the graph pattern and additionally states that both *GridNode*s and the *PathToHill* edge belong to the *context* of the pattern, i.e. they should not be deleted, if they are not mentioned on the RHS, unless they are also mentioned outside of context parentheses. The RHS is pretty straightforward: As the old *AntPos* edge is not mentioned on the RHS it is removed, and a new *AntPos* edge to the next *GridNode* is created. Additionally the pheromones counter of the old *GridNode* is incremented.

When we compare the GrGen.NET and the XL solution, we see, that the slightly more verbose version of GrGen.NET is intuitively understandable, while it is unclear what XL's context parentheses and the "==>>" production means without reading the sparse documentation and several examples. However, treating a rule as a language statement or as a method seems very useful for an imperative (object-oriented) language.

3.3 Simple Rule Execution

After specifying a rule, the next important question is, how to execute them.

GrGen.NET 1.3.1: GrGen.NET uses search plan driven graph pattern matching as described in [Bat06] and [BKG08]. At run-time, the user can trigger an analysis of the current graph structure and create new rule instances with potentially better search plans to speed up rule execution (see line 12 and 13 of the listing below). These dynamically generated rule instances can be accessed through the according *LGSPActions* rule container (see line 17).

Because the initial graph is always empty and an analysis on an empty graph is not very expedient, GrGen.NET uses so called *static search plans* as initial rule instances. To adumbrate the expected graph structure for single rules, the user may annotate some elements of the patterns with a search priority. For example "$hill[prio = 10000] : AntHill;$" in a pattern indicates that this *AntHill* is a very good point to start searching. Using these annotations the user can sometimes avoid dynamically creating search plans to save some time and simplify the use of the rule. The *GoHome* rule instance with a static search plan for the example from the previous section can be accessed as "*Action_GoHome.Instance*" (see line 9).

Whenever the matcher of a rule is invoked, you can specify, after how many matches the matcher shall stop searching. The high-level API of GrGen.NET 1.3.1 only uses two amounts, one and all possible (infinity), thus allowing two modes of rule execution:

- The normal mode searches for one match and rewrites it (see line 9).
- The "all at once"-mode searches for all matches in the graph, and then rewrites them one after the other ignoring any conflicts (see line 17). These conflicts include adding edges to a node, which is deleted by rewriting another match, or negative application conditions triggered by changes caused by another rewrite. Here the user is responsible for preventing conflicts.

```
 1 // Initialize graph and the actions container
 2 LGSPGraph graph = new LGSPGraph(new AntWorldGraphModel());
 3 AntWorldActions actions = new AntWorldActions(graph);
 4
 5 ...  // Some initialization, also declaring LGSPNode curAnt
 6
 7 // Apply GoHome rule to one occurrence of the pattern
 8 // using the static search plan
 9 Action_GoHome.Instance.Apply(graph, curAnt, dummyInt);
10
11 // Generate a new search plan for the GoHome rule
12 graph.AnalyzeGraph();
13 actions.GenerateSearchPlans(Action_GoHome.Instance);
14
15 // Apply GoHome rule to all occurrences of the pattern at once
16 // using the dynamically generated search plan
17 actions.GetAction("GoHome").ApplyAll(graph, curAnt, dummyInt);
```

But as shown in the introduction, return parameters can only be accessed via the low-level API in GrGen.NET 1.3.1. Consider a rule with the following header:

```
rule RuleA(a:Ant, -p:PathToHill->) : (GridNode, PathToHill)
```

To apply this rule once, you have to use something like this:

```
INode ant;
IEdge homeWay;
...
INode destGridNode = null;
IEdge nextWayHome  = null;
LGSPAction ruleA = actions.GetAction("RuleA");
IGraphElement[] pars = new IGraphElement[] { ant, homeWay };
```

```
LGSPMatches matches = ruleA.Match(graph, 1, pars);
if(matches.Count != 0)
{
    IGraphElement[] rets = action.Modify(graph, matches.matches.First);
    destGridNode = (INode) rets[0];
    nextWayHome  = (IEdge) rets[1];
}
```

The C# compiler has no chance to verify, that the types and the order of the rule parameters and return values are correct. If the developer mixes them up or changes the rule declaration, this will only be noticed at runtime and because the return values are also only casted to unspecific types, an error may occur at a much later place.

Although in the version developed here the high-level API has been adapted to make handling return parameters easier, the new version of the above example is only a bit more type-safe:

```
IAnt ant;
IPathToHill homeWay;
...
IGridNode   destGridNode = null;
IPathToHill nextWayHome  = null;
LGSPAction ruleA = actions.GetAction("RuleA");
object[] rets = ruleA.Apply(graph, ant, homeWay);
if(rets != null)
{
    destGridNode = (IGridNode) rets[0];
    nextWayHome  = (IPathToHill) rets[1];
}
```

Still, both the parameters and the return values are not type-safe (the *Apply* method takes any number of *object* instances after the first parameter), but if the return value types are wrong, it will crash there immediately instead of somewhere else.

Another way to apply the rule in GrGen.NET 1.3.1 is to use *extended graph rewrite sequences* (XGRS)[3], which supply logical and iterative sequence control over several rules and support nested transactions. To communicate between the caller and the XGRS and between rules inside the XGRS, named but typeless variables are used which are managed by the graph. So when *ant* and *homeWay* are already stored in graph variables, we can formulate the example like this:

```
INode destGridNode = null;
IEdge nextWayHome  = null;
if(actions.ApplyGraphRewriteSequence("(dest, next)=RuleA(ant, homeWay)"))
{
    destGridNode = (INode) graph.GetVariableValue("dest");
    nextWayHome  = (IEdge) graph.GetVariableValue("next");
}
```

[3] "Extended" compared to the old C implementation of GrGen.

This XGRS executes the rule *RuleA* once (normal mode) with the parameters *ant* and *homeWay* and stores the results in the graph variables *dest* and *next* if the pattern has been found (otherwise they stay untouched). *ApplyGraphRewriteSequence*, which should probably be renamed to *ApplyXGRS* for brevity, returns the result of the XGRS, which is true for a rule, if the pattern was found. The all-at-once mode version of the XGRS would be "`(dest, next)=[RuleA(ant, homeWay)]`".

Of course, the string of the XGRS is only evaluated at runtime, so — additionally to order mix-ups — this may lead to parsing errors and use of unknown variables at runtime. The latter isn't even an error because of questionable semantics of rule applications with undefined parameters. Last but not least parsing and interpreting the string causes additional runtime overhead.

XL: As described in the previous section, in XL no changes to the graph induced by transformation blocks take effect until a derivation step, which is triggered by a call to *derive*. The derivation mode decides what happens:

- In *PARALLEL_MODE* all changes take effect, ignoring any conflicts.
- In *PARALLEL_NON_DETERMINISTIC_MODE* almost all changes take effect: If one node is deleted by several applications, only one of these applications is chosen randomly.
- In *SEQUENTIAL_MODE* only the first application is used.
- In *SEQUENTIAL_NON_DETERMINISTIC_MODE* one application is chosen randomly.

Per default the derivation mode additionally has an *EXCLUDE_DELETE_FLAG* set indicating, that no node used by an application may have been deleted or it's changes will not be applied.

Note, that a transformation block can contain multiple rules, so with the last mode and some loops you can easily simulate AGG's [ERT99] rule layers, where all rules in one layer are applied in a random order until none of the rules are applicable anymore.

Given a *curAnt* variable of type *Ant* and a *incrPher* variable of type *int*, executing the *GoHome* rule as defined in the last section in XL is not more than:

```
GoHome(curAnt, incrPher);
derive();
```

However, if you need a rule, which returns more than one element or value, you have to use either an object or static fields. Let us have a look at the XL version of the *RuleA* application: First we have to note, that we cannot access edges in patterns and that patterns only support simple edges (in contrast to multiedges). Although we could use the API to traverse all adjacent edges to find the correct edge object, we will just let *RuleA* return the source and the target nodes of the edge.

```
module RuleATriple(GridNode dest, GridNode nextSrc, GridNode nextTgt);

public RuleATriple RuleA(Ant a, GridNode pSrc, GridNode pDest) {
    ...
    if(!found) return null;
    derive();
    return new RuleATriple(dest, nextSrc, nextTgt);
}

...

GridNode destGridNode = null;
GridNode nextWayHomeSrc = null, nextWayHomeTgt = null;
RuleATriple res = RuleA(a, pSrc, pDest);
if(res != null) {
    destGridNode = res.dest;
    nextWayHomeSrc = res.nextSrc;
    nextWayHomeTgt = res.nextTgt;
}
```

Although the edges are not handled very nicely and the user has to manually manage the result object, this solution is still much better than the GrGen.NET solutions, as it is inherently type-safe. Although the **module** notation already saves much time, it would be nice to use a generic class instead. Sadly, XL does not support them though.

Comparing GrGen.NET and XL we see, that the normal rule application mode of GrGen.NET is the same as XL's *SEQUENTIAL_MODE* when the *derive* method is called after each rule, and the all-at-once mode is similar to the *PARALLEL_MODE* without the *EXCLUDE_DELETE_FLAG*. For the AntWorld test case an additional rule execution mode has been implemented in the proposed version of GrGen.NET, which I just mention here because it is not a significant feature for this work: The random-match-selector mode first finds all matches of a pattern and then randomly selects n to be applied. An XGRS randomly selecting 4 matches of *RuleA* is written as "`(dest, next)=$4[RuleA(ant, homeWay)]`". This new mode is comparable to XL's *SEQUENTIAL_-NON_DETERMINISTIC_MODE* for which this n would always be 1.

Looking at how rules encapsulated in methods (or similar) are applied, we saw that neither GrGen.NET nor XL comes up with a type-safe *and* convenient solution for rules returning more than one element or value. But at least GrGen.NET's way of specifiying multiple return values for an action is convenient.

3.4 Executing Many Rules

In complex graph transformation scenarios many rules have to be executed, partially depending on the results of other rules. Here results are not only the return values but also whether or not they matched at all. Rules may need to be executed iteratively for a given number of times or as long as possible.

As an example we take the sequence control of the GrGen.NET AntWorld solution submitted to the GraBaTs 2008 except for the fair random choosing of the ant's ways: The simulation is executed for a given number of rounds (here 250). In every round first each ant is moved, then it is checked, whether an ant has reached the border of the grid making it necessary to extend it, then all dropped food units in the ant hill are transformed to ants, and at last the pheromones on each grid node are reduced.

An ant's move is determined as follows: if the ant can take food, it does so and starts going back to the ant hill dropping pheromones on the path, which ends this round for the ant. If it already carries food, it also goes towards the ant hill ending the round for the ant. If the ant is already at the ant hill, it drops the food and continues searching. When searching, the ant tries to follow a pheromone path away from the ant hill. But if no such exists, it just heads for any direction.

When the grid has to be extended, first one node of the outer grid is extended, then all others are extended and connected to the according previous new node. Lastly, the first and the last new nodes are connected. As the grid nodes in the corners are extended by three new nodes rather than just one, they are handled by special rules.

GrGen.NET 1.3.1: This is an ideal case for graph rewrite sequences (XGRS):

```
 1  actions.ApplyGraphRewriteSequence(
 2        "(curAnt=firstAnt && "
 3      + "(("
 4      + "    TakeFood(curAnt) | GoHome(curAnt) ||"
 5      + "    DropFood(curAnt) | (SearchAlongPheromones(curAnt) ||"
 6      + "                        SearchAimless(curAnt))"
 7      + ") && (curAnt)=GetNextAnt(curAnt))*"
 8      + "| ((cur)=ReachedEndOfWorld"
 9      + "    && ((cur, curOuter)=GrowWorldFirstNotAtCorner(cur)"
10      + "         || (cur, curOuter)=GrowWorldFirstAtCorner(cur))"
11      + "    && ((cur, curOuter)=GrowWorldNextNotAtCorner(cur, curOuter)"
12      + "         || (cur, curOuter)=GrowWorldNextAtCorner(cur, curOuter))*"
13      + "    && GrowWorldEnd(cur, curOuter))"
14      + "| (curAnt)=Food2Ant(curAnt)*"
15      + "| [EvaporateWorld]"
16      + ")[250]");
```

Lines 2–7 iterate over each ant and move them, lines 8–13 expand the grid if required, line 14 emerges one ant per food unit, line 15 lets the pheromones evaporate a bit, and line 16 closes the main loop iterating through 250 rounds.

As mentioned in the previous section, the string is only checked at runtime which is a big disadvantage. Also the numerous string concatenations add much unnecessary noise. Using multi-line verbatim strings would help, but is not yet supported by the XGRS parser because of the new-line characters.

XL: Assuming that all rules are declared like the rules in the GrGen.NET implementation, return false or null if no match was found, and that the *GrowWorld...Corner* rules return a *GridTuple* containing two *GridNode*s, it could be implemented in XL as:

```
 1 for(1:250) {
 2     Ant curAnt, nextAnt = firstAnt;
 3     do {
 4         curAnt = nextAnt;
 5         TakeFood(curAnt) | GoHome(curAnt) ||
 6         DropFood(curAnt) | (SearchAlongPheromones(curAnt) ||
 7                             SearchAimless(curAnt));
 8     }
 9     while((nextAnt = GetNextAnt(curAnt)) != null);
10     GridNode cur = ReachedEndOfWorld();
11     if(cur != null)
12     {
13         GridTuple gt = GrowWorldFirstNotAtCorner(cur);
14         if(gt == null) gt = GrowWorldFirstAtCorner(cur);
15         GridNode curOuter;
16         do {
17             cur = gt.first;
18             curOuter = gt.second;
19         } while((gt = GrowWorldNextNotAtCorner(cur, curOuter)) != null
20              || (gt = GrowWorldNextAtCorner   (cur, curOuter)) != null);
21         GrowWorldEnd(cur, curOuter);
22     }
23     while((curAnt = Food2Ant(curAnt)) != null) {}
24     EvaporateWorld();
25 }
```

In contrast to Java, XL allows expressions to be used as statements like in C. This makes it possible to connect rules with Boolean operators very similar to GrGen.NET's XGRSs (see lines 5–7). While the XGRSs only assign values to the given return variables if the rule succeeds, assignments like in line 9 always assign a value. So, to implement the XGRS "$(curAnt) = GetNextAnt(curAnt)$" an additional variable *nextAnt* is needed, which is only assigned to *curAnt* once after the declaration and when *GetNextAnt* succeeded (see line 4).

GrGen.NET's XGRSs provide an easy way to control the execution of many rules using Boolean and iterative operators, but they can quickly become difficult to understand if the expressions are too deeply nested and badly indented. On the other side the implementation in XL is not necessarily more understandable. Especially because of the problems with multiple return values the code is more verbose.

3.5 Graph Modification

When working with graphs, we sometimes want to change the graph unconditionally and perhaps at a-priori known places. We can distinguish between four scenarios here: The creation and initialization of single elements, the deletion of single elements, and more complex irregular and regular graph modifications.

3.5.1 Creation and Initialization of Single Elements

Consider you want to create a new *Ant* already carrying food at the *AntHill* given as *hill* and store it in the local variable *ant*.

GrGen.NET 1.3.1: Defining and executing a graph rewrite rule induces quite some overhead, so for this small example using the API is shorter:

```
LGSPNode ant = graph.AddNode(NodeType_Ant.typeVar);
((INode_Ant) ant.attributes).hasFood = true;
graph.AddEdge(EdgeType_AntPos.typeVar, ant, hill);
```

As mentioned in the introduction, this shows some problems:

- Compared to the informal description of this example, this solution is very verbose, making it more difficult to understand while skip-reading it. The domain-specific syntax of GrGen.NET would be much better.
- The *ant* variable has a very unspecific type, prohibiting type-safe use across several functions and easy access to the attributes. Accessing attributes in this type-unsafe way can introduce errors, which will only be noticed at runtime, when an invalid cast exception occurs.

XL: In XL it's as simple as:

```
Ant ant;
[ ==> a:Ant(true) -AntPos-> hill, { ant = a; }; ]
```

At first sight it's clear what's happening, as long as you use a naming convention clearly differentiating between element types and local variables, and remember that we defined the *Ant* constructor to take the initial value for the $hasFood$ attribute. The only thing spurious here is the explicit assignment from the graph variable to the local variable.

3.5.2 Deletion of Single Elements

Now we want to delete an *Ant* given as *ant* with all its adjacent edges.

GrGen.NET: Again using the API leads to the shortest implementation for this example:

```
graph.RemoveEdges(ant);
graph.Remove(ant);
```

First we have to remove the adjacent edges and then remove the *Ant* node itself. As the *Remove* method is a low-level method, this is unnecessarily verbose. Probably it should be renamed to *RemoveNode* and a new *Remove* method should be introduced calling *RemoveEdges* and *RemoveNode*.

XL: Here we get:

```
[ ant ==>>; ]
```

This is as short as it can get. But we should note here, that an RGG unit (the class in which an .rgg file is embedded) always has an associated graph and that all transformation blocks operate on this graph. To operate on another graph one would probably have to call a custom method in the other RGG unit. Also it seems, that all nodes have to be reachable by a so called *RGGRoot* node. So if *ant* was the only connection of the rest of the graph to the *RGGRoot* node, the graph will be empty afterwards. If the "==>" production operator is used instead though, all outgoing edges from *ant* will be redirected to come from the *RGGRoot* node, so the elements reachable over these edges will still be part of the graph. This very unexpected behaviour makes it much more difficult to work with XL as a graph rewrite system.

3.5.3 More Complex Graph Modifications (irregular)

As a somewhat more complex, but irregular example we construct "The House of St. Nik's" out of *GridNode*s and *GridEdge*s and assign 100 food units to the bottom nodes.

GrGen.NET 1.3.1: For this example it is better to declare a rule in a .grg file:

```
rule BuildStNikHouse {
    modify {
              a:GridNode;
        b:GridNode; c:GridNode;
        d:GridNode; e:GridNode;

        eval { d.food = 100; e.food = 100; }

        d -:GridEdge-> b -:GridEdge-> a -:GridEdge-> c -:GridEdge-> b
          -:GridEdge-> e -:GridEdge-> d -:GridEdge-> c -:GridEdge-> e;
    }
}
```

And invoke it from C# code:

```
Action_BuildStNikHouse.Instance.Apply(graph);
```

Here we can use the static version of the action to apply the rule to the graph, as it does not depend on any search plan anyway (empty pattern).

XL: In XL it is again just one transformation block:

```
[ ==>
            a:GridNode,
    b:GridNode,       c:GridNode,
    d:GridNode(100),  e:GridNode(100),

    d -GridEdge-> b -GridEdge-> a -GridEdge-> c -GridEdge-> b
      -GridEdge-> e -GridEdge-> d -GridEdge-> c -GridEdge-> e;
]
```

Comparing both solutions, they look very similar. The XL solution uses the nice custom constructors to initialize the bottom *GridNode*s, while GrGen.NET has to initialize them manually in one or more **eval** blocks. While the advantage of the constructors may not become very clear in this example, consider what the GrGen.NET solution would look like for a RHS with 39 graph elements and 35 attribute initializations[4]. As XL's transformation blocks are normal statements, there is no need to split rule declaration and rule execution.

3.5.4 More Complex Graph Modifications (regular)

For a regular example let us attach 8 new *Ant*s to an *AntHill* given as *hill* and connect them as a singly linked list using *NextAnt* edges. We want the first *Ant* of the list to be stored in the local variable *firstAnt* and the last one in *lastAnt*. Because the function which initiates the creation of the ants is already complex enough, the graph modification should be written somewhere else.

GrGen.NET 1.3.1: One rule creates all elements manually:

```
rule CreateAnts(hill:AntHill) : (Ant, Ant) {
    modify {
        a1:Ant -:AntPosition-> hill <-:AntPosition- a2:Ant;
        a3:Ant -:AntPosition-> hill <-:AntPosition- a4:Ant;
        a5:Ant -:AntPosition-> hill <-:AntPosition- a6:Ant;
        a7:Ant -:AntPosition-> hill <-:AntPosition- a8:Ant;

        a1 -:NextAnt-> a2 -:NextAnt-> a3 -:NextAnt-> a4
           -:NextAnt-> a5 -:NextAnt-> a6 -:NextAnt-> a7 -:NextAnt-> a8;

        return (a1, a8);
    }
}
```

And the C# code calls the rule and extracts the return values:

[4]See the "InitExample" rule from the UML2CSP example shipped with GrGen.NET. The initialization of the attributes was not only tedious but also very error prone, so I forgot to initialize one element and initialized one element twice when I wrote it down the first time. With element constructors this could not have happened in the first place.

```
INode firstAnt = null, lastAnt = null;
LGSPAction action = Action_CreateAnts.Instance;
LGSPMatches matches = action.Match(graph, 1, new IGraphElement[] { hill });
if(matches.Count != 0)
{
    IGraphElement[] rets = action.Modify(graph, matches.matches.First);
    firstAnt = (INode) rets[0];
    lastAnt  = (INode) rets[1];
}
```

For small graph initializations this may be acceptable, but for larger ones using the API makes more sense:

```
public void CreateAnts(INode hill, out INode firstAnt, out INode lastAnt)
{
    firstAnt = lastAnt = null;
    for(int i = 0; i < 8; i++)
    {
        INode a = graph.AddNode(NodeType_Ant.typeVar);
        graph.AddEdge(EdgeType_AntPos.typeVar, a, hill);
        if(lastAnt != null)
            graph.AddEdge(EdgeType_NextAnt.typeVar, lastAnt, a);
        else
            firstAnt = a;
        lastAnt = a;
    }
}

...

INode firstAnt, lastAnt;
CreateAnts(hill, out firstAnt, out lastAnt);
```

XL: Again we need a special class to return two elements in a type-safe way:

```
module TupleAntAnt(Ant first, Ant second);

public TupleAntAnt CreateAnts(AntHill hill) {
    Ant first = null, last = null;
    [ ==>
        for(1:8) (
            a:Ant -AntPos-> hill,
            if(last != null)
                (last -NextAnt-> a)
            else
                { first = a; },
            { last = a; }
        );
    ]
    return new TupleAntAnt(first, last);
}

...

TupleAntAnt res = CreateAnts(hill);
Ant firstAnt = res.first;
Ant lastAnt = res.second;
```

Here we can use a **for** and an **if** graph statement on the RHS of the rule to build the 8 ants and handle the singly-linked list. Besides **for** and **if**, XL also provides `switch`, **do-while**, **while** and **synchronized** as imperative control statements for the RHS of a rule.

The XL API solution looks very similar to the GrGen.NET 1.3.1 API solution, but is clearer by using domain-specific syntax for element creation. The special support for the imperative control statements does not seem to be necessary and also does not fit to the declarative character of the GrGen.NET rules.

3.5.5 Conclusion

As we saw in this section, domain-specific syntax embedded in the host language is especially important for small to medium changes, where you do not want to explicitly call a rule defined somewhere else and loose all context. Here the easy propagation of graph elements (and other values) to local variables and vice versa is a must. For medium to large modifications it is absolutely legitimate to put them somewhere else in most situations.

3.6 Working with Matches

In several situations you need to do something with a match of a rule what cannot be done on the RHS of the rule. You may need to

- do complicated attribute calculations,
- do further checks whether the match is really what you want,
- choose some of the available matches, or
- start looking for matches of another rule depending on the current match.

As a simple example we want to move an *Ant* given as *ant* to a surrounding *GridNode* with the most pheromones. To avoid requiring quadratic time to find this *Ant*, the solution should iterate over the matches and remember the current match with the most pheromones[5].

[5]This could be done automatically when this condition was directly formulated in the pattern with the help of a negative application condition, but GrGen.NET is not able to optimize this and I doubt XL can do this.

GrGen.NET 1.3.1: For this example we are going to find all matches, then select one with the highest amount of pheromones, and finally apply the change to that one. So, first, we have to define a rule:

```
rule MoveAnt(ant:Ant) {
    ant -p:AntPos-> :GridNode <-:GridEdge-> nextNode:GridNode;
    modify {
        delete(p);
        ant -:AntPos-> nextNode;
    }
}
```

Then we specify the C# code finding all possible ways (see line 4 below), selecting our favorite (see lines 5–14) and moving the *Ant* to the best *GridNode* (see line 16):

```
1  int maxPheromones = -1;
2  LGSPMatch maxMatch = null;
3  LGSPAction moveRule = actions.GetAction("MoveAnt");
4  LGSPMatches matches = moveRule.Match(graph, 0, new IGraphElement[] { ant });
5  foreach(LGSPMatch match in matches)
6  {
7      INode nextNode = match.nodes[Rule_MoveAnt.NodeNums.nextNode];
8      int curPheromones = ((INode_GridNode) nextNode.attributes).pheromones;
9      if(curPheromones > maxPheromones)
10     {
11         maxPheromones = curPheromones;
12         maxMatch = match;
13     }
14 }
15 if(maxMatch != null)
16     moveRule.Modify(graph, maxMatch);
```

Of course, we could also use two rules for finding and rewriting the match to make it a bit clearer, but it would require more code.

XL: The XL code is very similar to the GrGen.NET 1.3.1 solution, but uses two embedded rules for the small patterns:

```
int maxPheromones = -1;
GridNode bestNode = null;
[
    ant -AntPos-> GridNode -GridEdge- g:GridNode
    ::>
    {
        if(g.pheromones > maxPheromones)
        {
            maxPheromones = g.pheromones;
            bestNode = g;
        }
    }
]
```

```
if(bestNode != null)
[
    ant -AntPos-> (* GridNode *)
    ==>
    ant -AntPos-> bestNode;
]
```

Again the embedded domain-specific code is more concise and makes the intention of the code much clearer.

3.7 Graph Traversal

In compiler construction it is often important to apply optimization rules in a specific order to avoid optimizing unreachable code, for instance. Therefore, the intermediate representation (IR) graph is typically traversed in depth-first-search (DFS) order using pre- and/or post-walkers, which execute user-specified code before and after visiting a node, respectively. Sometimes this code starts another DFS e.g. to search previous instructions writing to the same memory location as a current store instruction. To simulate a similar case in the AntWorld environment, the exercise is to start a DFS at the *AntHill* searching the subgraph induced by *PathToHill* edges against their orientation. For each found *Ant*, another DFS should be started at the *Ant*'s position to find a path back to the *AntHill* with no *Ant* obstructing the way. If no such path is found, the *Ant* is to be removed.

GrGen.NET 1.3.1: Version 1.3.1 neither supports high-level graph traversals nor visited flags at all, so this example has to be implemented using the low-level API and hash maps to keep track of visited elements. First we create a class which can be used for generic DFS pre-walking:

```
public enum WalkerResult
{
    Proceed, Skip, Abort
}

public delegate WalkerResult PrewalkHandler(INode curNode);
public delegate void AddChildrenHandler(INode curNode, Stack<INode> nodeStack);

public class DFS
{
    public static void DoDFS(INode startNode, PrewalkHandler prewalk,
                             AddChildrenHandler addChildren)
    {
        Dictionary<INode, object> visitedSet = new Dictionary<INode, object>();
        Stack<INode> nodeStack = new Stack<INode>();
        nodeStack.Push(startNode);

        while(nodeStack.Count > 0)
        {
            INode curNode = nodeStack.Pop();
            if(visitedSet.ContainsKey(curNode)) continue;
            visitedSet[curNode] = null;

            WalkerResult res = prewalk(curNode);
            if(res == WalkerResult.Skip) continue;
            if(res == WalkerResult.Abort) break;

            addChildren(curNode, nodeStack);
        }
    }
}
```

The *DFS.DoDFS* method receives the node where to start the DFS, a pre-walker handler, and a handler responsible for choosing the children of the according current node. The pre-walker handler determines, whether the current node's children should be visited or not or the whole DFS should abort here. The *Dictionary* (representing a hash map) prevents a node from being visited more than once.

The example shall start searching for *Ants* at the *AntHill* (line 1) along incoming *PathToHill* edges (lines 23–27). If it finds any *Ants* (line 8) which cannot reach the *AntHill* without being blocked by other *Ants* (line 10), the *Ants* at the current *GridNode* shall be removed from the graph (lines 12–17):

```
 1 DFS.DoDFS(antHill, Prewalk, AddChildren);
 2
 3 ...
 4
 5 WalkerResult Prewalk(INode curNode)
 6 {
 7     // If the curNode has an incoming AntPos edge, we found one or more Ants
 8     if(curNode.GetExactIncoming(EdgeType_AntPos.typeVar).GetEnumerator().MoveNext())
 9     {
10         if(!CheckReachable(curNode, antHill))
11         {
12             foreach(IEdge antpos in curNode.GetExactIncoming(EdgeType_AntPos.typeVar))
13             {
14                 INode ant = antpos.Source;
15                 graph.RemoveEdges(ant);
16                 graph.Remove(ant);
17             }
18         }
19     }
20     return WalkerResult.Proceed;
21 }
22
23 void AddChildren(INode curNode, Stack<INode> nodeStack)
24 {
25     foreach(IEdge edge in curNode.GetExactIncoming(EdgeType_PathToHill.typeVar))
26         nodeStack.Push(edge.Source);
27 }
```

The *CheckReachable* method starts another DFS looking for an unobstructed way to the *AntHill*:

```
 1 bool CheckReachable(INode from, INode to)
 2 {
 3     bool found = false;
 4     DFS.DoDFS(from,
 5         /* Prewalk */ delegate(INode curNode)
 6         {
 7             if(curNode == to)
 8             {
 9                 found = true;
10                 return WalkerResult.Abort;
11             }
12
13             if(curNode != from && curNode.GetExactIncoming(EdgeType_AntPos.typeVar).
14                     GetEnumerator().MoveNext())
15                 return WalkerResult.Skip;
16
17             return WalkerResult.Proceed;
18         },
19         /* AddChildren */ delegate(INode curNode, Stack<INode> nodeStack)
20         {
21             foreach(IEdge inEdge in curNode.GetCompatibleIncoming(
22                                                 EdgeType_GridEdge.typeVar))
23                 nodeStack.Push(inEdge.Source);
24             foreach(IEdge outEdge in curNode.GetCompatibleOutgoing(
25                                                 EdgeType_GridEdge.typeVar))
26                 nodeStack.Push(outEdge.Target);
27         }
28     );
29     return found;
30 }
```

If the DFS has found the destination node, it aborts the search and returns true (lines 7 – 11). If it comes to a node, which is not the current start node and which is occupied by an *Ant* (lines 13 – 15), the DFS skips its children, which are defined as any adjacent nodes connected by a *GridEdge* (lines 21 – 26). Using C# delegates for the handlers provides a very good locality for the very simple handler implementations.

XL: XL does not support high-level graph traversals, too, but through an "*Persistence-Manager*" it provides (undocumented[6]) visited "marks" which can be used in our generic DFS pre-walker class instead of a hash set:

```
public abstract class DFS {
    public const int PROCEED = 0;
    public const int SKIP = 1;
    public const int ABORT = 2;
```

[6]Actually I found out about XL's visited marks 1.5 months after I added them to GrGen.NET.

```
    PersistenceManager manager;

    public DFS(PersistenceManager manager) {
        this.manager = manager;
    }

    public void doDFS(Node startNode) {
        int visitedMark = manager.allocateBitMark(false);

        Stack nodeStack = new Stack();
        nodeStack.push(startNode);

        while(nodeStack.size() > 0) {
            Node curNode = (Node) nodeStack.pop();
            if(curNode.setBitMark(visitedMark, true)) continue;

            int res = prewalk(curNode);
            if(res == SKIP) continue;
            if(res == ABORT) break;

            addChildren(curNode, nodeStack);
        }

        manager.disposeBitMark(visitedMark, true);
    }

    abstract int prewalk(Node curNode);
    abstract void addChildren(Node curNode, Stack nodeStack);
}
```

This does the same as the GrGen.NET 1.3.1 version except that it uses those visited marks needing allocation and disposal instead of a hash map.

The first DFS is also very similar:

```
new DFS(getPersistenceManager()) {
    int prewalk(Node curNode) {
        // If the curNode has an incoming AntPos edge, we found one or more Ants
        if(!empty((* curNode <-AntPos- Node *))) {
            if(!CheckReachable(curNode, antHill)) [
                (* curNode <-AntPos- *) Ant ==>;
            ]
        }
        return PROCEED;
    }
    void addChildren(Node curNode, Stack nodeStack) {
        for((* curNode <-PathToHill- n:Node *))
            nodeStack.push(n);
    }
}.doDFS(antHill);
```

Checking for any *Ant*s indicated by incoming *AntPos* edges can be formulated nicer than in the GrGen.NET solution with XL's queries and the aggregation function *empty* (line 4). A query returns all matching elements corresponding to the textually rightmost pattern element (*Node* in this case). Iterating over the children can also be simplified using a query (line 12). The use of an embedded rule makes removing all *Ant*s at *curNode* more concise (line 6).

The *CheckReachable* method also profits from queries, but needs an instance variable to save the *found* result:

```
boolean found;

public boolean CheckReachable(final Node from, final Node to) {
    found = false;
    new DFS(getPersistenceManager()) {
        int prewalk(Node curNode) {
            if(curNode == to) {
                found = true;
                return ABORT;
            }

            if(curNode != from && !empty((* curNode <-AntPos- Node *)))
                return SKIP;

            return PROCEED;
        }
        void addChildren(Node curNode, Stack nodeStack) {
            for((* curNode -GridEdge- n:Node *))
                nodeStack.push(n);
        }
    }.doDFS(from);
    return found;
}
```

The comparison between GrGen.NET and XL again shows the advantages of domain-specific syntax over conventional API calls. With the comments "Prewalk" and "AddChildren" the basic structure of the GrGen.NET version of *CheckReachable* already looks quite nice and does not need an allocation of the *DFS* class. Of course, being able to also specify a post-walker would be nice, too. It was just left out here to keep the example small and to be able to use a single stack to implement it.

Supporting visited flags directly in the graph rewrite system makes it possible to manage these flags more efficiently. For example by saving the flags in the graph elements themselves instead of an external hash map, not only the overhead of the hash value calculation, but also of the additional objects can be saved.

3.8 Conclusion

The different properties of XL and GrGen.NET 1.3.1 presented in this chapter are summarized and rated in table 3.1 with respect to the goals of this work, i.e. convenience (Con) and type safety (TS). While the convenience rating tries to describe how nice it is to work with a criterion (obviously subjective), the static type safety rating states whether the provided types ensure correct use already at compile time. The ratings are + + for perfect, + for good, o for acceptable, - for bad, - - for very bad. × means "not available" or "not supported". In the following subsections the criteria and their ratings for XL and GrGen.NET 1.3.1 are briefly described.

3.8.1 Model

Graph: The expressiveness of graphs.

> **XL**: *Convenience*: Typed, attributed nodes (+). Colored, directed simple edges (o). Support for using node types for edges[7] allowing typed, attributed multiedges (+). While rooted graphs may be useful in some domains, in general they are not (-).
>
> *Type safety:* Everything is allowed as a node or an edge, so there cannot occur any run-time errors. (+)
>
> **GrGen.NET 1.3.1**: *Convenience*: Typed, attributed nodes (+). Typed, attributed, and directed multiedges (+).
>
> *Type safety:* Present (+).

Multiple graph models: Can multiple graph models be handled?

> **XL**: *Convenience*: For multiple graph types the user has to fall back from the convenient *RGG* classes to .xl files to declare differently typed graph classes (o). XL does not have explicit graph models (-). But element types can be declared in the scope of graph types, so that they must be qualified when used in another graph type (o).
>
> *Type safety*: The graph classes are meant to have special types subclassed from *RGG*, so it is type-safe (+).
>
> **GrGen.NET 1.3.1**: *Convenience*: Although it is possible to specify, reuse, and combine different explicit graph models (+), one .grg file can only use one specific graph model combination (×). On the API-level, a general graph type is used, not providing any convenience methods for creating special graph elements[8] (-).
>
> *Type safety*: On the API-level, all graph instances have the same type (-).

[7]Implicitly resulting in an "`-in_edge-> node -out_edge->`" construct.

[8]Note, that this is not needed in XL because of the embedded rules.

	XL		**GrGen.NET** 1.3.1	
	Con	TS	Con	TS
Model:				
Graph	o	+	+	+
Multiple graph models (s/A)	o	+	×/o	×/-
Graph structure assertions	×	×	+	+
Node types (spec/API)	o	+	+	+/-
Edge types (spec/API)	- -	-	+	+/-
Attribute access (spec/API)	++	+	++/- -	+/-
Constructors (node/edge)	+/×	+/×	×	×
Element methods (node/edge)	+/×	+/×	×	×
Embedding	+	+	×	×
Rules:				
Modes	+	+	o	+
LHS and RHS	o	+	o	+
Parameters (spec/API)	++	+	o	+/-
Returns (spec/API)	-	+	o/-	+/-
Outer access to LHS	++	+	-	-
Outer access to RHS	o	+	×	×
Embedding	+	+	×	×
Rule execution:				
Graph queries	+	+	×	×
XGRS	×	×	-	-
Boolean combination	-	+	×	×
- without returns	+	+	-	-
Graph traversal:				
Visited flags	+	-	×	×
DFS	- -	-	×	×
Other traversals	×	×	×	×

Table 3.1: Comparison of different features of XL and GrGen.NET 1.3.1 in terms of convenience (Con) and type safety (TS)

Graph structure assertions: Assertions on the structure of a graph.

XL: Not supported (×).

GrGen.NET 1.3.1: *Convenience*: Simple, yet useful assertions for node-edge-nodebetween two node and one edge type (+).

Type safety: Unaffected (+).

Node types: Properties of node types.

XL: *Convenience*: Node types are attributed and support normal Java inheritance (+), multiple inheritance only manually using interfaces (-). Concise definition of node types with attributes possible (+).

Type safety: All node types are special types (+).

GrGen.NET 1.3.1: *Convenience*: Native attributed node types (+) with multiple inheritance (+).

Type safety: While the specification can distinguish all types (+), on the API-level, there is only one type for all nodes (-).

Edge types: Properties of edge types.

XL: *Convenience*: Edge types, however, do not support attributes (-) and inheritance has to be emulated using bit flags (- -).

Type safety: Any integer could be used as a edge type (-).

GrGen.NET 1.3.1: *Convenience*: Native attributed edge types (+) with multiple inheritance (+).

Type safety: While the specification can distinguish all types (+), on the API-level, there is only one type for all edges (-).

Attribute access: How can attributes be accessed in rules and via the API?

XL: *Convenience*: Intuitive attribute access with the "." operator throughout the language (+ +).

Type safety: Present (+).

GrGen.NET 1.3.1: *Convenience*: Intuitive attribute access with the "." operator in rule specification (+ +), but very inconvenient attribute access via API because of access over *attribute* field (- -).

Type safety: The *attribute* field must be cast to the correct type in order to access the attributes, what can fail at runtime (-).

Constructors: Constructors of graph element types.

XL: *Convenience*: Node types support constructors which can be used both on the LHS and the RHS (+). Edge types do not support them.

Type safety: The constructors are bound to their types and therefore always type-safe (+).

GrGen.NET 1.3.1: Not supported (×).

Element methods: Is it possible to declare methods in element types?

XL: *Convenience*: Node types support element methods (+). Edge types do not.

Type safety: Present (+).

GrGen.NET 1.3.1: Not supported (×).

Embedding: Is it possible to declare graph types in the "normal" source code?

XL: *Convenience*: Yes, either implicitly in .rgg files or explicitly in .xl files (+).

Type safety: Present (+).

GrGen.NET 1.3.1: Not supported (×).

3.8.2 Rules

Modes: Available rule modes.

XL: *Convenience*: Three different, but unintuitive production operators (o). Several derivation modes (+).

Type safety: Present (+).

GrGen.NET 1.3.1: *Convenience*: Only test/rule and modify/replace (o).

Type safety: Present (+).

LHS and RHS: The left and right hand side of graph rewrite rules.

XL: *Convenience*: Concise but unintuitive syntax (o). Transitive patterns (+). Method calls can be used as pattern predicates (+). Edges cannot be refered to (-). No syntactic differentiation between unnamed pattern element declarations and variable accesses (-). No syntactic differentiation between in and out parameters of constructors on the LHS (-).

Type safety: Present as non element types are implicitly wrapped (+).

GrGen.NET 1.3.1: *Convenience*: Straightforward syntax (+). No dynamic patterns (-).

Type safety: Rule specifications are strongly typed (+).

Parameters: Parameters of rules "called" in some way. For XL this means, that the rule is wrapped by a method.

XL: *Convenience*: Natural parameter passing according to method declaration (+). All parameter types are supported (+).

Type safety: Present (+) unless deliberately prevented.

GrGen.NET 1.3.1: *Convenience*: Variadic parameter passing, thus no parameter information available (o). Only graph elements can be passed to a rule (o).

Type safety: Although the rule specification handles parameters type-safe (+), the API receives the parameters in a general graph element array, and thus allows the user to pass an edge for a node parameter and vice versa and to pass a wrong number of parameters (-).

Returns: Return values of rules "called" in some way like above.

XL: *Convenience*: Only one return value possible, more return values have to wrapped by a helper object (-).

Type safety: Type-safe, when the user defines custom helper objects for each combination of returned types (XL bases on Java 1.4 and thus does not provide generics) (+)

GrGen.NET 1.3.1: *Convenience*: Any number of return values supported, but only graph elements can be returned (o). No convenience API available when using return values (-).

Type safety: Again the rule specification is type-safe with respect to return values (+), but the API uses one simple array of graph elements for all returned nodes and edges, which could also be accessed with wrong indices (-).

Outer access to LHS: How can matched graph elements of the LHS of a rule be accessed by imperative code executed before or instead of a RHS without "returning" the elements explicitly?

XL: *Convenience*: Inside the imperative code of an execution rule, the LHS elements can just be accessed via their names (+ +). There is no simpler way than that.

Type safety: The names refer to elements with the correct type (+).

GrGen.NET 1.3.1: *Convenience*: Elements must be manually extracted from element kind specific arrays (-).

Type safety: Array indices cannot be verified at compile-time (-).

Outer access to RHS: How can elements of the RHS of a rule be accessed by imperative code executed after the rule without "returning" the elements explicitly? A typical example would be the creation of a graph element from an embedded rule and its use outside of the rule.

XL: *Convenience*: Elements must be manually assigned to a variable or field of the outer scope (o).

Type safety: Present (+).

GrGen.NET 1.3.1: Not supported (×).

Embedding: Is it possible to embed graph rewrite rules in the “normal” source code?

XL: *Convenience*: Yes, in so called transformation blocks (+).
Type safety: Present (+).

GrGen.NET 1.3.1: Not supported (×).

3.8.3 Rule Execution

Graph queries: Are graph queries supported?

XL: *Convenience*: Yes, together with aggregation and filter functions they can simplify queries in many situations (+).
Type safety: Present (+).

GrGen.NET 1.3.1: Not really supported: enumerations on element types are too simple for this criterion.

XGRS: Support for extended graph rewrite sequences.

XL: Not supported (×).

GrGen.NET 1.3.1: *Convenience*: Supported by API (+). Parameters and return values have to be accessed via graph variables (-). Much noise due to string handling of XGRS (-).
Type safety: String is interpreted at run-time, so besides wrong names even syntactic errors are possible (-)

Boolean combination: Is it possible to combine several “rule applications”[9] directly with Boolean operators like in XGRSs?

XL: *Convenience*: Support for expression statements (+). Boolean combination becomes more verbose when return values are used (-). Return values are also assigned, when a rule fails (-). Rule methods must trigger derivation steps (o).
Type safety: Present (+).

GrGen.NET 1.3.1: Not supported (×). because of missing convenience function for rule application with return values.

Boolean combination without returns: This criterion is similar to “Boolean combination” but here only “rule applications” not returning any values (except success and failure) are considered.

XL: *Convenience*: Support for expression statements (+). Rule methods must trigger derivation steps (o).
Type safety: Present (+).

GrGen.NET 1.3.1: *Convenience*: Dummy variables necessary because C# has no expression statements (-).
Type safety: General graph element array for parameters (-).

[9]Method calls for XL.

3.8.4 Graph Traversal

Visited flags: The existance of visited flags.

XL: *Convenience*: (Undocumented) visited flags (+).

Type safety: Any integer could be used as a handle for a so called "bit mark" and no error is reported at run-time, when the handle is not valid (-).

GrGen.NET 1.3.1: Not supported (×).

DFS: Support for traversing a graph in depth-first order.

XL: *Convenience*: There seems to be undocumented support for some kind of visitors which can be used to traverse the graph in depth-first order, but it is unclear how they work[10] (- -).

Type safety: The visitors receive unspecific *Object* references (-).

GrGen.NET 1.3.1: Not supported (×).

Other traversals: Support for any other graph traversals, like breadth-first-search or some weighted searchs.

XL: Seems not to be supported.

GrGen.NET 1.3.1: Not supported (×).

[10] *de.grogra.graph.Graph.accept* in combination with a *de.grogra.graph.VisitorImpl* subclass and an *de.grogra.graph.EdgePattern* instance. Using these classes like in *de.grogra.graph.impl.Node.cloneGraph* did not work, though. Not a single element was found.

Chapter 4

Proposed Solutions

This chapter contains a discussion about what can be done to improve the GrGen.NET solutions of the typical scenarios of the previous chapter. In the first part of this chapter I describe changes to the API of GrGen.NET 1.3.1, in the second part some new features are added to GrGen.NET, and in the third part an embedded domain-specific language G# extending C# by special constructs for graph rewriting with GrGen.NET is introduced.

4.1 GrGen.NET API Changes

An important weapon against many programming errors is static typing as supported by C#. If an API has adequately diverse types and the parameters of methods are accordingly constricted, most type-related errors can already be caught at compile-time. For the API of GrGen.NET 1.3.1 this is not the case: Sections 3.3 and 3.5.1 indicated that the API has some serious problems with type-safety. But this not only applies to the elements and their attributes, but also to their meta-types and to graph instances. In this section the type-safety is improved also resulting in a much more convenient programming interface.

4.1.1 Assimilating Attributes into Graph Elements

Nodes in GrGen.NET 1.3.1 are represented solely as generic *LGSPNode* objects implementing the *INode* interface (edges analogously as *LGSPEdge*s implementing *IEdge*) (see figure 4.1). Elements store their attributes in an external type-specific object implementing the (empty) *IAttributes* interface. The virtual multiple inheritance of the element types is implemented using .NET's multiple inheritance on interfaces.

Using an external object for the attributes has some advantages:

- Retyping an element can be done by simply exchanging its *ITypeFramework* and *IAttributes* fields representing their type and attributes by new objects according to the new type. Only the common attributes have to be copied, which is implemented using reflection. No adjacent elements need to be adapted.

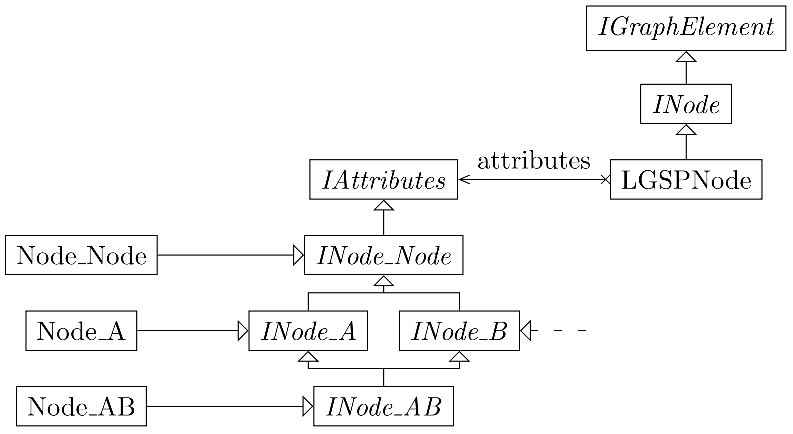

Figure 4.1: The node representation in GrGen.NET 1.3.1

- The size of a graph element object is independent of the number of attributes, so elements with many attributes take less space in the processor cache during a matching process, when the attributes are not touched, than they would take, if the attributes were stored inside the elements.

But accessing the attributes of a new element in C# is quite a hassle:

```
LGSPNode ant = graph.AddNode(NodeType_Ant.typeVar);
((INode_Ant) ant.attributes).hasFood = true;
```

This not only leads to lengthy code but also to runtime cast exceptions, when the programmer mixes up the types. Also an *LGSPNode* could have any node type, so only variable names can help making the code understandable when elements are passed to methods.

Starting with GrGen.NET 1.4, the element attributes are stored directly in the graph elements by merging the type-specific object classes and the according *LGSPNode* or *LGSPEdge* class. With version 2.0 the root element types *Node* and *Edge* do not implement special *INode_Node* and *IEdge_Edge* interfaces anymore (see figure 4.2), which added nothing to the general interfaces anyway. Also the "*Node_*" and "*Edge_*" prefixes have been removed to provide the developer with exactly the names specified in the model. He still has to live with the distinction between the interfaces and the concrete types of the elements, though, because classes do not allow multiple inheritance in C#.
But the new element representation introduces some disadvantages:

- Retyping a graph element creates a new object and requires all adjacent elements to be updated. All references to the retyped element become invalid. The same happens when an element is deleted (just as it was before in GrGen.NET 1.3.1).
- Searching through a list of graph elements with many attributes causes more cache pollution.
- Directly reusing elements deleted by the RHS of a rule is only possible for elements with the exact same type. Thus this optimization is useless for most rules.

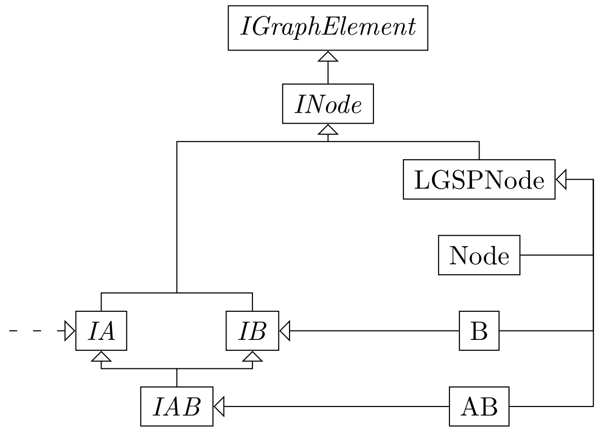

Figure 4.2: The improved node representation in GrGen.NET 2.0

To shorten these disadvantages the following changes have been implemented:

- To recognize invalid elements produced by retyping or deleting, the properties *IGraphElement.Valid*, *IGraphElement.ReplacedByElement*, *INode.ReplacedByNode*, and *IEdge.ReplacedByEdge* have been introduced.
- *LGSPNode/LGSPEdge.Get/SetAttribute* now work without reflection. For each element type custom methods are generated to speed up attribute access via attribute names.
- The use of reflection for element retyping has been replaced by custom methods, too.
- Cloning of elements uses copy constructors with custom code instead of the reflection-based *Object.MemberwiseClone*().
- As direct element reuse during a rewrite of a match is not possible anymore for most rules, element pooling has been introduced, which stores up to 10 deleted elements of each type for later reuse[1]. This makes it even possible to reuse elements beyond rule boundaries.

GrGen.NET 2.0 realizes type-safe element creation over static methods in the element classes creating and adding a new instance of the according element to a given graph. So now you can write the above code much more concisely and type safe:

```
Ant ant = Ant.CreateNode(graph);
ant.hasFood = true;
```

With the not yet implemented constructors this could even be:

```
Ant ant = Ant.CreateNode(graph, true);
```

[1]The value 10 was chosen to avoid keeping too many elements in the cache during the available benchmarks. But a reasonable value is highly problem specific, so an adaptive or user-definable value might be desirable.

4.1.2 Refactoring the Graph Model API Architecture

The graph model architecture of the GrGen.NET 1.3.1 API is not able to statically distinguish between node and edge models, and node and edge types. Type models just inherit from the general *ITypeModel* interface and element meta-types from the *IType* interface (see figure 4.3). Users were able to put edge types into node creation functions. In some cases even without an immediate runtime error!

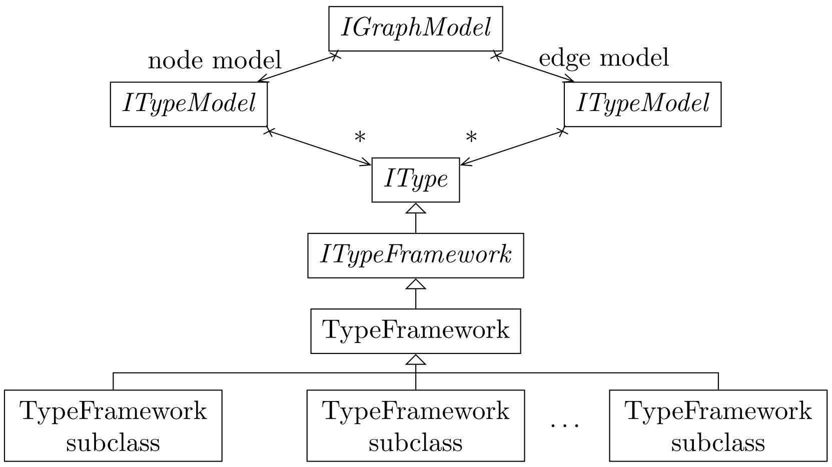

Figure 4.3: The type architecture of GrGen.NET 1.3.1

To solve this problem, the graph model architecture has been restructured in version 1.4. A concrete element type model now implements either `INodeModel` or `IEdgeModel`, and concrete element types inherit either from `NodeType` or `EdgeType` (see figure 4.4). More abstract access is provided through `ITypeModel` and `GrGenType`. To get access to an instance of a given (meta-)type the static *TypeInstance* property of the according element class can be used. This keeps any arbitrary pre- or suffixes of class names off the user.

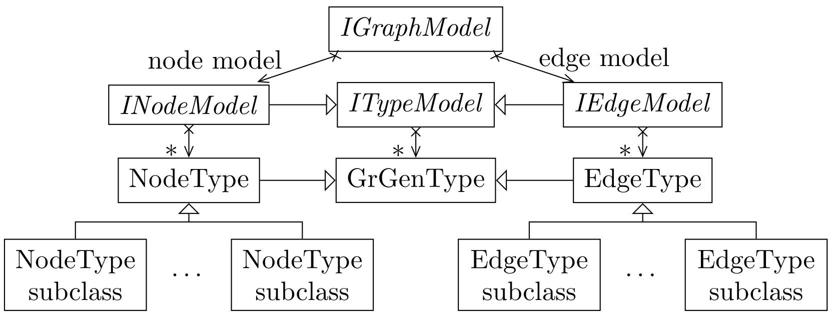

Figure 4.4: The improved type architecture of GrGen.NET 1.4

With the new structure, errors like creating nodes out of edge types are now reported at compile-time. Also the maintainability of both the user and the library source code have been improved by this change, as for method parameters and return values it is now much clearer what you are holding in your hand and what you are allowed to do with it.

4.1.3 Type-safe Handling of Graphs

GrGen.NET 1.3.1 represents concrete graph objects by general *LGSPGraph* instances. Their *Model* property points to the according instance of a specific model class which was used to instantiate the graph. In an application with multiple graphs of different graph models this again leads to problems with too unspecific types: A method receiving an *LGSPGraph* object could get a graph instantiated with any graph model, although it only supports a special one. For example it adds a node whose type only exists in this graph model. Thus it should only be possible to pass graphs of the correct model to the method.

By merging *LGSPGraph* and the specific model classes, GrGen.NET 2.0 again achieves better type-safety: Now the method parameter declaration can constrain which graph model must be used. So, for such methods passing wrong graphs is not possible anymore. Additionally it is now possible to provide model specific convenience methods for element creation, which especially make code completion features of editors more useful.

4.2 New GrGen.NET Features

Some solutions for the problems of the last chapter require new features to be added to GrGen.NET. For example it must be possible to pass/return an integer value easily to/from a graph rewrite rule to allow transparent access to local variables and parts of the graph patterns. In this section I describe these new features.

4.2.1 Element Constructors

To get rid of the sometimes large textual distances between element creation and attribute initialization (cf. section 3.5.3), bodyless constructors are added to element types, whose parameters are just the names of the attributes to be initialized (see first parameter in line 8 in listing 4.1). They take effect as if they were executed after all attribute initializers (like in line 2). Similar to C++ the last parameters may be annotated with default values, so that these parameters can be omitted when the constructor is used on the RHS of a rule (see all other parameters in line 8).

Listing 4.1: A small GrGen.NET model using a constructor

```
 1 node class Font {
 2     fontname:string="Verdana";
 3     height:int=10;
 4     bold:boolean;
 5     italic:boolean;
 6     underlined:boolean;
 7
 8     Font(height, fontname="Arial", bold=false, italic=false, underlined=false);
 9
10     // Alternative constructor
11     // Font(height, fontname="Arial");
12 }
```

But for elements with many default parameters where you only want to set one or two of them depending on the case, this only helps when these parameters are declared as one of the first default parameters. A typical example for an element, for which this is not the case, comes from GUI programming: The *Font* node class in listing 4.1 always requires a height parameter, but the name of the font and whether the font is bold, italic, and/or underlined are optional (see line 8). So if you wanted to instantiate an underlined "Courier" font of height 12 on the RHS, with the C++-like syntax you would have to specify all parameters of the constructor, although most of them would receive their default values:

```
:Font(12, "Courier", false, false, true);
```

By additionally allowing to use named parameters for all attributes not used as positional parameters, this can be avoided and the constructor use can become easier to understand, if there is no sensible order on the attributes. With named parameters the constructor can also be simplified by removing all default parameters whose values equal the default values of the according attributes (see line 11). So now we can write:

```
:Font(12, "Courier", underline=true);
```

A (parameterless) default constructor always implicitly exists. This way the user can always initialize any combination of element attributes with named parameters. The default constructor may be overwritten by a constructor with only optional parameters. If used without parameters, the default constructor may be written with or without parentheses to keep compatibility to previous versions of GrGen.NET.

Analogously to the subpatterns introduced by [Jak08] the constructors can not only be used on the RHS to instantiate an element, but also on the LHS to match one. The default values are ignored, though: only those attributes, which are explicitly given by either a positional or a named parameter, add a constraint to the according pattern element. Otherwise, very confusing errors would be possible, when default parameters were used without care. Thus, with the "Alternative constructor" of listing 4.1 the *Font* constructor in line 1 of listing 4.2 matches any italic font, the one in line 3 matches any font with height 9, and the last constructor in line 4 matches any underlined "Courier" font with height 12.

Listing 4.2: An action using constructors on the LHS

```
1 test FontTest(f1:Font(italic=true))
2 {
3     f2:Font(9);
4     f3:Font(12, "Courier", underline=true);
5 }
```

4.2.2 Non-Graph-Element Parameters and Return Values

To allow reasonable parameterization of graph rewrite rules (i.e. without creating elements dedicated to parameter transfer) and interaction between C# and GrGen.NET, the rules need to be able to get and return variables from the C# context. This can be achieved by using action parameters for C# variables read in an action and using action return values for changed variables. Analogously to the syntactic differentiation between nodes and edges for action parameters in the GrGen.NET syntax, variable parameters are prefixed by the new `var` keyword. Parameter return types are just given directly. Listing 4.3 shows a small example rule with the two variable parameters *i* and *str*: It matches a *Resource* node with an amount greater or equal to the given integer *i* and pointed to by a given *Process* node. For such a match, the resource amount is reduced by *i*, and a string formed from the given string *str* and the new amount is returned.

For the API this change means, that several functions now work on `object` instances instead of `IGraphElement` instances. Especially graph variables may now also be non-graph-element values. With the GrShell the rule from the example could be invoked as shown in listing 4.4: First an initialization rule is called which returns a *Process* node. Then we initialize the variable *intValue* with the integer 7. The example rule can simply be called by providing the needed arguments in an `xgrs` command, where we pass "Remaining donuts" as the string parameter. If the matched *Resource* node had an *Amount* of 10 before rewriting it, the *answer* variable will be set to the string "Remaining donuts: 3".

Listing 4.3: Example of a GrGen.NET rule with variable parameters and return values

```
rule varExample(var i:int, p:Process, var str:string) : (string)
{
    p --> r:Resource;
    if { r.Amount >= i; }
    modify {
        eval {
            r.Amount = r.Amount - i;
        }
        return (str + ": " + r.Amount);
    }
}
```

Listing 4.4: Example showing how to use the rule from listing 4.3 in the GrShell

```
xgrs (proc)=Init
intValue = 7
xgrs (answer)=varExample(intValue, proc, "Remaining donuts")
```

4.2.3 Visited Flags

Visited flags are simple Boolean markers attached to each graph element. As the name suggests they can be used to mark elements as visited while walking a graph, but they may also be used to store any other Boolean values. For nested graph traversals even multiple visited flags may be necessary at the same time. In the previous chapter the visited flags were implemented using hash maps, but in GrGen.NET some visited flags could be hosted directly inside the graph elements as there are still several free bits in a flag variable left. By adding support for visited flags directly into GrGen.NET, no additional memory is needed for up to eight simultaneously used visited flags.

While the visited flags can be made accessible to C# programs over the API without problems, special syntax is needed to access them out of rules, because method calls are not allowed there. Table 4.1 shows what you can do with visited flags and how you do it with the API, the GrShell, and inside graph rewrite actions. To use a visited flag, first of all, you have to allocate one from the graph. The resulting visitor ID can then be used to get and set the according visited flag of given graph elements, to reset the visited flag for all graph elements, and to deallocate the flag again. Graph rewrite actions can access a visited flag with the new *visited*(elem, visitorID) expression. As an example, listing 4.5 searches for *Resource* nodes which have not been visited by the visitor specified by *visID*, yet, and which are pointed to by a given *Process* node. For a found match, the *Amount* attribute of the resource is decremented and the resource is marked as visited by the given visitor.

Listing 4.5: Example of a GrGen.NET rule using a visited flag

```
rule visitedExample(var visID:int, p:Process)
{
    p --> r:Resource;
    if { !visited(r, visID); }
    modify {
        eval {
            r.Amount = r.Amount - 1;
            visited(r, visID) = true;
        }
    }
}
```

You can have an arbitrary number of allocated visited flags at any time. Considering a compiler scenario, you could have one walker visiting all program blocks and another walker visiting any nodes – including program blocks – starting from the block currently visited by the first walker without any interference.

The first eight visited flags are implemented by using unused flags in the graph elements. So they do not need any additional memory and reading and writing the flags is a $O(1)$ operation. Resetting such flags is a $O(|N| + |E|)$ operation, $|N|$ being the number of nodes in the graph and $|E|$ the number of edges, as the flags have to be reset in each graph element. To avoid unnecessary resetting of elements, the graph remembers, whether any nodes and any edges have been marked as visited. So, if you only mark nodes and then reset the flags, only the nodes are actually reset.

When the user allocates more than eight visited flags at a time, hash maps are used for the remaining flags which require $O(|N| + |E|)$ additional memory. Reading and writing them has an expected time complexity of $O(1)$. Resetting such flags also is a $O(|N| + |E|)$ operation, as all buckets of the hash map are cleared by `Dictionary<TKey,TValue>.Clear()`.

When a visited flag is deallocated, the flag is just added to a list of free flags. The flag is not reset to avoid unnecessary runtime overhead. Only when this flag is allocated again, a reset is forced. So visited flag allocation is a $O(|N| + |E|)$ operation, while deallocation is a $O(1)$ operation.

Table 4.1: Visited Flag Tasks with API, GrShell, and Actions

Task	With API
Flag allocation	int *visID* = *graph*.AllocateVisitedFlag();
Get flag value	bool *val* = *graph*.IsVisited(*elem*, *visID*);
Set flag value	*graph*.SetVisited(*elem*, *visID*, *true*);
Reset flag values	*graph*.ResetVisitedFlag(*visID*);
Flag deallocation	*graph*.FreeVisitedFlag(*visID*);

Task	With GrShell
Flag allocation	*visID* = allocvisitflag
Get flag value	*val* = isvisited *elem visID*
Display flag value	isvisited *elem visID*
Set flag value	setvisited *elem visID true*
Reset flag values	resetvisitflag *visID*
Flag deallocation	freevisitflag *visID*

Task	Inside a graph rewrite action
Flag allocation	–
Get flag value	if { visited(*elem*, *visID*); }
Set flag value	eval { visited(*elem*, *visID*) = *true*; }
Reset flag values	–
Flag deallocation	–

4.3 Embedded GrGen.NET: G#

The improvements on the API and the feature set of GrGen.NET are an important part of the way to conciseness, type-safety, and convenience. But the convenience of locally embedded graph patterns and rules, which can access all elements in the current scope and create a new scope with the pattern elements, cannot be reached, yet. In this part of the chapter I introduce the embedded domain-specific language G# to complete the GrGen.NET improvements developed in this work.

G# extends the general-purpose programming language C# by constructs specific to the domain of graph transformation. On the one hand, by extending C# we can use the language the API was designed for and still have the full expressiveness of a general-purpose language required for complex applications. On the other hand, the extensions provide domain-specific notations for graph patterns and graph modifications improving the conciseness and productivity for related tasks enormously. In the following subsections these extensions are introduced.

4.3.1 Model Specification

For the declaration of a graph model in G#, the GrGen.NET model specification has been adapted to C# and encapsulated in a **model** type, which now contains the node and edge class declarations for this graph model. An instance of the model type represents the graph and the model at the same time. In GrGen.NET 1.3.1 every graph had the unspecific *LGSPGraph* type initialized with a graph model, whose type name was the name of the model file without the extension suffixed by "GraphModel". Now the developer can just use the name *he* specified to access the graph/model type. Listing 4.6 illustrates some more additions: To support object-oriented development, it is now possible to declare methods and properties in element types (see line 13) and the graph type (line 27). For the graph type also own fields are allowed (line 26), which may help easily accessing unique graph elements.

Listing 4.6: An extended G# version of the model from section 3.1

```
 1 public model AntWorld {
 2     node class GridNode {
 3         int food, pheromones;
 4 
 5         GridNode(food, pheromones=0);
 6     }
 7     node class AntHill : GridNode {
 8         food = 8;
 9     }
10     node class Ant {
11         bool hasFood;
12 
13         public bool TakeFood(GridNode foodPlace)
14         {
15             if(foodPlace.food == 0) return false;
16             foodPlace.food--;
17             hasFood = true;
18             return true;
19         }
20     }
21     edge class GridEdge connect GridNode[*] --> GridNode[*];
22     edge class PathToHill : GridEdge connect inherited;
23     edge class AntPos connect Ant[1] --> GridNode[*];
24     edge class NextAnt connect Ant[0:1] --> Ant[0:1];
25 
26     private int FoodCounter;
27     public void AssignFood(GridNode newGridNode)
28     {
29         if(--FoodCounter != 0) return;
30         newGridNode.food += 100;
31         FoodCounter += 10;
32     }
33 }
```

4.3.2 Rule Specification

Just like for the model types, new kinds of type declarations are introduced for rules, tests, and subpatterns using the keywords **rule**, **test**, and **pattern**, respectively. To specify which graph model should be used, the according type keyword is suffixed by the model name in angle brackets as shown in listing 4.7. The rest (signature and declaration body) is identical to the rest in GrGen.NET's .grg files.

Listing 4.7: G# versions of rules from sections 3.5.3 and 3.5.4

```
public rule<AntWorld> BuildStNikHouse {
    modify {
                a:GridNode;
        b:GridNode;      c:GridNode;
        d:GridNode(100); e:GridNode(100);

        d -:GridEdge-> b -:GridEdge-> a -:GridEdge-> c -:GridEdge-> b
          -:GridEdge-> e -:GridEdge-> d -:GridEdge-> c -:GridEdge-> e;
    }
}

public rule<AntWorld> CreateLessAnts(hill:AntHill) : (Ant, Ant) {
    modify {
        a1:Ant -:AntPosition-> hill <-:AntPosition- a2:Ant;
        a3:Ant -:AntPosition-> hill <-:AntPosition- a4:Ant;

        a1 -:NextAnt-> a2 -:NextAnt-> a3 -:NextAnt-> a4;

        return (a1, a4);
    }
}
```

4.3.3 Domain-Specific Syntax in Methods

As shown in the previous chapter we want to do the following things directly inside a method:

- Embed small to medium rules,
- iterate over one or more matches of a given pattern and execute some C# code for each match, and
- do simple unconditional modifications to the graph.

Looking at how rules are executed, this is just iterating over matches for the according patterns and do some modifications. So by splitting rules into an iteration and match handling/modification part, we can handle all above cases easily. In the following we will refer to the match handling/modification part as *match handler*.

The match and matchatonce Statements

The new statements **match** and **matchatonce** are responsible for the iteration part: The **match** statement searches a given graph for a match of a given pattern, executes the match handler for this match, and starts searching again up to a given number of times, which defaults to 1. The matching fails, if no match was found. The **matchatonce** statement searches for up to a given number of matches (which defaults to infinity) first, and then executes the match handler for each one sequentially. It fails, if a given minimum number of matches (which defaults to 1) was not found, in which case the match handler is not executed at all. Whenever one of these statements fails, an optional **else** part is executed.

Listing 4.8 illustrates the use of both statements searching for an adjacent *GridNode* with the highest *pheromones* attribute like in section 3.6. The **do** keyword in line 6 specifies, that the match handler is a C# statement, in this case a block. Line 17 uses a **replace** block of a graph rewrite rule instead. It is also possible to use a **modify** block here.

Of course, C#'s block oriented scoping rules also apply to the new statements: the graph element variable g declared in line 4 is available in the match handler, just like the variables *maxPheromones* and *bestNode* could be used in both patterns.

Listing 4.8: A G# version of the example from section 3.6

```
 1 int maxPheromones = -1;
 2 GridNode bestNode = null;
 3 matchatonce(graph) {
 4     ant -:AntPos-> :GridNode <-:GridEdge-> g:GridNode;
 5 
 6     do {
 7         if(g.pheromones > maxPheromones) {
 8             maxPheromones = g.pheromones;
 9             bestNode = g;
10         }
11     }
12 }
13 if(bestNode != null) {
14     match(graph) {
15         ant -:AntPos->;
16 
17         replace {
18             ant -:AntPos-> bestNode;
19         }
20     }
21 }
```

The modify Statement

For unconditional modifications of the graph, using an empty pattern would be much too verbose, so a **modify** statement modifying a given graph directly is what we need. Unconditional modifications are often used to create elements, which are used later. To make the new elements accessible, they have to be declared in a scope surrounding the

modify statement and assigned to by this statement. As this was not intended by the original GrGen rule syntax, the rule syntax is extended to allow setting an already existing name for an unbound pattern element: When you would write *name* : *Type* to declare a new pattern element with the name *name* and the type *Type*, you can now use an already declared variable *declared* for this by writing *declared* := *Type* as shown in Listing 4.9. This avoids any explicit assignments, which are needed by XL.

Listing 4.9: A G# version of the example from section 3.5.1

```
Ant ant;
modify(graph) {
    ant:=Ant(true) -:AntPos-> hill;
}
DoSomething(ant);
```

4.3.4 Treating Rule Applications like Methods

For rule applications it is important to handle parameters and return values type-safely. In XL this is fulfilled, but returning more than one element or value is problematic, as Java does not support **out** or **ref** parameters like C# does. But **out** parameters are probably also not suitable, as we do not want our variables to be changed, if no match was found. However, **ref** parameters require our variables to be initialized, so the compiler cannot check, whether each execution path to a usage of the variables contains *sensible*[2] assignments to them.

A syntax using tuples like in the XGRS would make the code type-safe and much cleaner and would allow to only assign to the return variables, when the rule has actually matched. To find out whether the rule has matched at all, the whole expression could evaluate to a boolean value indicating this:

```
1 IGridNode destGridNode = someOldGridNode;
2 if(!(destGridNode, IPathToHill nextWayHome) ?= graph.RuleA(ant, homeWay))
3     Console.WriteLine("RuleA has not been found");
4 else
5 {
6     bool succeeded = (IAnt ant) ?= graph.DoSomething(destGridNode, nextWayHome);
7 }
```

Let *RuleA* and *DoSomething* be names of actions declared for the graph model implied by the type of *graph*. The above code shows the new "?=" tuple assignment operator for rule applications, which has the precedence of a primary operator. A new token is required here to make the language unambiguous, but as a side effect the question mark also suggests that this assignment is conditional: The variable *destGridNode* in line 2 stays untouched, if the rule does not succeed, so it is still the *someOldGridNode* in line 6. The new variable *nextWayHome* of type *IPathToHill* declared in line 2 is only valid inside the **if**-expression and the "then"- and "else"-part, but might only be initialized in one of the latter parts.

[2]Sensible, in contrast to an axiomatic initialization with null.

Here it is only initialized in the "else"-part because of the negation of the rule result. The scoping rule is analogous to the scoping of variables declared in **for** statements.

So, the tuple assignment operator for rule applications can simplify single rule applications and the handling of especially multiple return values enormously.

4.3.5 Supporting XGRSs

As we have seen in section 3.4, XGRSs are very handy to execute a bunch of rules with some sequence control. Partially XGRSs are already covered by the special rule invocation syntax of the previous section. Together with support for expression statements[3] as in XL, the example from section 3.4 could be written as:

```
bool dummy;
for(int i = 0; i < 250; i++)
{
    Ant curAnt = firstAnt;
    do
    {
            graph.TakeFood(curAnt) | graph.GoHome(curAnt)
        ||  graph.DropFood(curAnt) | (  graph.SearchAlongPheromones(curAnt)
                                     || graph.SearchAimless(curAnt));
    }
    while((curAnt) ?= graph.GetNextAnt(curAnt));
    if((GridNode cur) ?= graph.ReachedEndOfWorld())
    {
            (cur, GridNode curOuter) ?= graph.GrowWorldFirstNotAtCorner(cur)
        ||  (cur,          curOuter) ?= graph.GrowWorldFirstAtCorner(cur);
        while( (cur, curOuter) ?= graph.GrowWorldNextNotAtCorner(cur, curOuter)
            || (cur, curOuter) ?= graph.GrowWorldNextAtCorner(cur, curOuter)) {}
        graph.GrowWorldEnd(cur, curOuter);
    }
    while((curAnt) ?= graph.Food2Ant(curAnt)) {}
    graph.EvaporateWorld.ApplyAll(graph);
}
```

Partly this is even better than the XGRS, as the structure of the code is much clearer due to the **do-while** loop and the **if** . Because of the special return syntax this also looks better than the XL version. But the repeated mentioning of the graph variable is cumbersome and the while loops with the empty body are not convincing either. With a special **exec** statement like in the GrGen.NET rule specifications our example can be formulated this way:

```
for(int i = 0; i < 250; i++)
{
    Ant curAnt = firstAnt;
    do
    {
```

[3] Allowing a simple expression to be a statement like in C, e.g. **true** || **false**;.

```
        exec(graph, TakeFood(curAnt) | GoHome(curAnt) ||
                    DropFood(curAnt) | (SearchAlongPheromones(curAnt) ||
                                        SearchAimless(curAnt)));
    }
    while((curAnt) ?= graph.GetNextAnt(curAnt));
    if((GridNode cur) ?= graph.ReachedEndOfWorld())
    {
        exec(graph,
            (
                (cur, GridNode curOuter) ?= GrowWorldFirstNotAtCorner(cur) ||
                (cur,          curOuter) ?= GrowWorldFirstAtCorner(cur)
            )
            &&
            (
                (cur, curOuter) ?= GrowWorldNextNotAtCorner(cur, curOuter) ||
                (cur, curOuter) ?= GrowWorldNextAtCorner(cur, curOuter)
            )*
            && GrowWorldEnd(cur, curOuter)
        );
    }
    exec(graph, (curAnt) = Food2Ant(curAnt)* | [EvaporateWorld]);
}
```

With the **exec** statement all features of XGRSs like iteration and nested transactions are available, and you do not have to repeat the graph qualification everywhere anymore. But for consistency the tuple assignments also use "?=" as operator instead of the original "=".

4.3.6 Special Syntax for Depth-First Search

With the domain-specific syntax introduced in section 4.3.3 the expressiveness has already been considerably improved. So the question is, whether more special syntax is needed to improve DFS traversal of graphs. To answer this question, let us first have a look at a G# version of the *CheckReachable* method from section 3.7 using the embedded rules:

```
bool CheckReachable(INode from, INode to)
{
    bool found = false;
    DFS.DoDFS(from,
        /* Prewalk */ delegate(INode curNode)
        {
            if(curNode == to)
            {
                found = true;
                return WalkerResult.Abort;
            }

            if(curNode != from)
            {
                match(graph)
                {
```

```
                curNode <-AntPos-;
                do {
                    return WalkerResult.Skip;
                }
            }
        }

        return WalkerResult.Proceed;
    },
    /* AddChildren */ delegate(INode curNode, Stack<INode> nodeStack)
    {
        matchatonce(graph)
        {
            curNode <-GridEdge-> n:Node;
            do {
                nodeStack.Push(n);
            }
        }
    }
);
return found;
}
```

This is already much clearer than before, but special syntax can improve this even further:

```
bool CheckReachable(INode from, INode to)
{
    fordepth(INode curNode in graph at from)
    {
        along {
            curNode <-GridEdge-> nextnode:Node;
        }
        pre do {
            if(curNode == to) return true;

            if(curNode != from)
            {
                bool hasAnt = false;
                match(graph, 1)
                {
                    curNode <-AntPos-;
                    do {
                        hasAnt = true;
                    }
                }
                if(hasAnt) continue;
            }
        }
    }
    return false;
}
```

The new **fordepth** statement declares a DFS through the graph given after **in** starting at the node stated after **at** (see line 3). The **along**-pattern specifies the children of the current node (here *curNode*) by declaring a special node named *nextnode* (line 6). Instead of the **along**-pattern, the user can also specify a **child**-pattern together with a **next**-pattern. The former pattern chooses the first child and the latter pattern all following children of the current node. Both ways result in a very compact and customizable specification of the DFS-tree. Apart from the **pre**-handler, it is also possible to specify a **post**-part, and both **pre**- and **post**-parts can use a **modify**-part or an action call instead of a **do**-part. The **pre**- and **post**-handlers will only be executed for those nodes which are compatible to the type given in the **fordepth** statement (line 3). Of course the incompatible nodes will still be handled by the DFS normally. They will just not be passed to those handlers.

Compared to the XL version this is quite good. Only the *hasAnt* variable is a bit cumbersome. Here XL's use of the aggregate function *empty* with a query is much more concise. The *hasAnt* variable is required, because a **continue** inside the **match** statement would just skip any further processing of the match due to scoping.

Sometimes choosing the children is more complicated[4]. Therefore it is also possible to specify an action call for **along** or **child** /**next** or a **do**-part acting like the body of an enumerator yielding child nodes:

```
along do {
    foreach(IEdge inEdge in curNode.GetCompatibleIncoming(GridEdge.TypeInstance))
        yield return inEdge.Source;
    foreach(IEdge outEdge in curNode.GetCompatibleOutgoing(GridEdge.TypeInstance))
        yield return outEdge.Target;
}
```

4.4 Conclusion

In this chapter several improvements of GrGen.NET and the embedded domain-specific language G# have been introduced. Together with improvements implemented by Edgar Jakumeit [Jak07, Jak08] and Sebastian Buchwald [Buc08] the convenience and type safety of GrGen.NET evolved very positively as shown in the "New" column of table 4.2 extending table 3.1 from the previous chapter. The column "with G#" shows that G# in combination with the proposed changes to GrGen.NET solves the remaining issues. In the following subsections the descriptions of the criteria are repeated and their ratings are described for the new columns.

[4]Not here, the example below is equivalent to the much conciser **along**-part above.

4.4.1 Model

Graph: The expressiveness of graphs.

> **New**: *Convenience*: Typed, attributed nodes (+). Typed, attributed, and directed/undirected [Buc08] multiedges (+ +).
>
> *Type safety:* Present (+).
>
> **With G#**: Unchanged.

Multiple graph models: Can multiple graph models be handled?

> **New**: *Convenience*: For each graph model a special graph class is generated providing special convenience methods for creating graph elements (+).
>
> *Type safety*: Present, because of special graph classes (+).
>
> **With G#**: Unchanged.

Graph structure assertions: Assertions on the structure of a graph.

> **New and with G#**: Unchanged.

Node types: Properties of node types.

> **New**: *Convenience*: Native attributed node types (+) with multiple inheritance (+).
>
> *Type safety*: Present due to custom classes per type (+).
>
> **With G#**: Unchanged.

Edge types: Properties of edge types.

> **New**: *Convenience*: Native attributed edge types (+) with multiple inheritance (+). Support for undirected edge types and (abstract) arbitrarily directed edge types as a common super type of directed and undirected edge types [Buc08] (+).
>
> *Type safety*: Present due to custom classes per type (+).
>
> **With G#**: Unchanged.

Attribute access: How can attributes be accessed in rules and via the API?

> **New**: *Convenience*: Intuitive attribute access with the "." operator in rule specification and via API (+ +)
>
> *Type safety*: The attribute properties are correctly typed (+).
>
> **With G#**: Unchanged.

Constructors: Constructors of graph element types.

> **New**: *Convenience*: Constructors supported for node and edge types (+). Default parameters, which can be used by position and by name (+).
>
> *Type safety*: Present (+).
>
> **With G#**: Unchanged.

	XL		**GrGen.NET**					
			1.3.1		New		with G#	
	Con	TS	Con	TS	Con	TS	Con	TS
Model:								
Graph	o	+	+	+	++	+	++	+
Multiple graph models (s/A)	o	+	×/o	×/-	×/+	×/+	+	+
Graph structure assertions	×	×	+	+	+	+	+	+
Node types (spec/API)	o	+	+	+/-	+	+	+	+
Edge types (spec/API)	- -	-	+	+/-	+	+	+	+
Attribute access (spec/API)	++	+	++/- -	+/-	++	+	++	+
Constructors (node/edge)	+/×	+/×	×	×	+	+	+	+
Element methods (node/edge)	+/×	+/×	×	×	×	×	+	+
Embedding	+	+	×	×	×	×	+	+
Rules:								
Modes	+	+	o	+	+	+	+	+
LHS and RHS	o	+	o	+	+	+	+	+
Parameters (spec/API)	++	+	o	+/-	+	+/-	++	+
Returns (spec/API)	-	+	o/-	+/-	+	+/-	++	+
Outer access to LHS	++	+	-	-	-	-	++	+
Outer access to RHS	o	+	×	×	×	×	+	+
Embedding	+	+	×	×	×	×	+	+
Rule execution:								
Graph queries	+	+	×	×	×	×	×	×
XGRS	×	×	-	-	-	-	+	+
Boolean combination	-	+	×	×	-	-	+	+
- without returns	+	+	-	-	-	-	+	+
Graph traversal:								
Visited flags	+	-	×	×	+	-	+	-
DFS	- -	-	×	×	×	×	+	+
Other traversals	×	×	×	×	×	×	×	×

Table 4.2: Comparison of different features of XL, GrGen.NET 1.3.1, GrGen.NET including the proposed changes, and GrGen.NET with G# in terms of convenience (Con) and type safety (TS) (s/A = spec/API)

Element methods: Is it possible to declare methods in element types?

New: Not supported (✕).

With G#: *Convenience*: Node and edge types support element methods (+).
Type safety: Present (+).

Embedding: Is it possible to declare graph types in the "normal" source code?

New: Not supported (✕).

With G#: *Convenience*: Yes, explicitly in G# files (+).
Type safety: Present (+).

4.4.2 Rules

Modes: Available rule modes.

New: *Convenience*: Test/rule and modify/replace (o). Rule modifiers **dpo**, **induced**, and **exact** to avoid many negative application conditions [Buc08](+).
Type safety: Present (+).

With G#: Unchanged.

LHS and RHS: The left and right hand side of graph rewrite rules.

New: *Convenience*: Straightforward syntax (+). Subpatterns and alternative patterns enabling recursive patterns [Jak08] (+). Constructors can be used for both matching and instantiating (+).
Type safety: Present (+).

With G#: Unchanged.

Parameters: Parameters of rules "called" in some way. For XL this means, that the rule is wrapped by a method.

New: *Convenience*: Graph elements as well as all attribute types supported by GrGen.NET can be passed to a rule (+).
Type safety: Although the rule specification handles parameters type-safely (+), the API receives the parameters in a general object array (-).

With G#: *Convenience*: Unchanged.
Type safety: The rule specification, the method-like rule calls, and the exec statement handle parameters type-safely (+).

Returns: Return values of rules "called" in some way like above.

New: *Convenience*: Graph elements as well as all attribute types supported by GrGen.NET can be returned by a rule (+). Return values must be extracted from an array (-).
Type safety: Although the rule specification handles parameters type-safely (+), the API uses a general object array for the return values (-).

With G#: *Convenience*: Graph elements as well as all attribute types supported by GrGen.NET can be returned by a rule (+). Convenient assignment to variables with tuple notation (+).

Type safety: The rule specification, the method-like rule calls, and the exec statement handle return values type-safely (+).

Outer access to LHS: How can matched graph elements of the LHS of a rule be accessed by imperative code executed before or instead of a RHS without "returning" the elements explicitly?

New: Unchanged.

With G#: *Convenience*: Inside a **do**-part the LHS elements can just be accessed via their names (+ +). There is no simpler way than that.

Type safety: The names refer to elements with the correct type (+).

Outer access to RHS: How can elements of the RHS of a rule be accessed by imperative code executed after the rule without "returning" the elements explicitly? A typical example would be the creation of a graph element from an embedded rule and its use outside of the rule.

New: Not supported (×).

With G#: *Convenience*: Elements can be directly assigned to variables or fields of the outer scope with "`:=`" (+).

Type safety: Present (+).

Embedding: Is it possible to embed graph rewrite rules in the "normal" source code?

New: Not supported (×).

With G#: *Convenience*: Yes, with several new statements (+).

Type safety: Present (+).

4.4.3 Rule Execution

Graph queries: Are graph queries supported?

New: Not supported (×).

With G#: Not supported (×).

XGRS: Support for extended graph rewrite sequences.

New: *Convenience*: Supported by API (+). Parameters and return values have to be accessed via graph variables (-). Verbatim strings supported to reduce noise in source code (o).

Type safety: String is interpreted at run-time, so besides wrong names even syntactic errors are possible (-)

With G#: *Convenience*: `exec` statement provides direct XGRS support (+).
Type safety: XGRS compilation, thus full checking supported (+).

Boolean combination: Is it possible to combine several "rule applications"[5] directly with Boolean operators like in XGRSs?

New: *Convenience*: Dummy variables necessary because C# has no expression statements (-).

Type safety: General object array for parameters and return values (-).

With G#: *Convenience*: Support for expression statements (+). Simple XGRS-like Boolean combination even with return values due to tuple assignments (+).

Type safety: Present (+).

Boolean combination without returns: This criterion is similar to "Boolean combination" but here only "rule applications" not returning any values (except success and failure) are considered.

New: Unchanged.

With G#: *Convenience*: Support for expression statements (+).
Type safety: Present (+).

4.4.4 Graph Traversal

Visited flags: The existance of visited flags.

New: *Convenience*: Support for visited flags in rules and via API (+).
Type safety: Any integer could be used as a handle for a so called "visitor ID", but an invalid handle is recognized in most situations (-).

With G#: Unchanged.

DFS: Support for traversing a graph in depth-first order.

New: Not supported (×).

With G#: *Convenience*: Special syntax for DFS provided (+).
Type safety: Present (+).

Other traversals: Support for any other graph traversals, like breadth-first-search or some weighted searchs.

New: Not supported (×).

With G#: Not supported (×).

[5]Method calls for XL.

Chapter 5

Implementation

This chapter contains a description of how the Mono C# compiler works and how it can be extended to support the language introduced in chapter 4 and specified in more detail in appendix A. Due to lack of time only a part of the extensions have actually been implemented during this work, namely: **model**, **rule**, **test**, and **pattern** declarations and **match**, **matchatonce**, **matchactionatonce**, **modify**, and **replace** statements. Constructors and element methods belonging to the **model** part, and element retyping, alternative patterns, and use of subpatterns belonging to the graph rewrite parts have not been implemented, yet. In appendix B some examples working with the current implementation are shown.

5.1 The Mono C# Compiler

The job of a C# compiler is to translate any number of C# files into Common Intermediate Language (CIL, a stack-based object-oriented assembly language for a virtual machine) code stored in a .NET assembly. When the assembly is later executed, the Common Language Runtime (CLR) just-in-time compiles the CIL code into machine code executable by the CPU. As the translation of the C# code is only partial, the compiler only has to check the code for syntactic and semantic errors, correctly resolve all names to their definitions, perform some basic simplifications mandated by the language standard, and emit the CIL code. The Mono C# compiler does this in several phases:

1. **Parsing:** All source files are parsed and an intermediate representation (IR) is built. While the lexer is handwritten, the parser is generated by the LALR(1) parser generator *jay* [Sch08], *yacc* retargeted to C# and Java. The AST is manually built in the grammar production handlers.

2. **Loading Referenced Assemblies:** All referenced assemblies are loaded via reflection to make their types available for the next phases.

3. **Type Hierarchy:** The type hierarchy is created into a new assembly, first recursively resolving the interfaces and then classes and structs, but all types are kept empty, as they may reference types which have not been created, yet.

4. **Member Definition:** All types are filled with their fields and empty definitions for methods, properties, indexers, and events.

5. **Code Generation:** All types are emitted into the assembly. Whenever a method is about to be emitted, it is first resolved, i.e. all used names are resolved to their definitions, the types of all expressions are calculated, constant expressions are simplified, and semantic analysis takes place. Then the code generation emits the method and continues with the current type.

6. **Output:** The assembly is written to a file.

The phases starting from phase 2 make heavy use of the reflection features provided by the `System.Reflection.Emit` namespace of the Framework Class Library. This is probably a reason why the Mono C# compiler is quite slow. For example `Assembly.Load` does not only load the meta information of an assembly into memory, which would be all the compiler needs to know about, but also the CIL code of all method implementations.

5.2 Extending the Mono C# Compiler

The G# language adds several constructs to the C# language, which must be properly handled by a compiler: four new kinds of types (**model**, **rule**, **test**, and **pattern** declarations), several new statements (**match**, **matchatonce**, **matchaction**, **matchactionatonce**, **modify**, **replace**, and **fordepth**), and two new expressions (**exec** and tuple-calls). Additionally special semantics is given to a **foreach** loop iterating over a graph type. Due to these extensions some additional steps must be taken in order to compile G# files: After parsing all source files, the parsed graph models must be processed, because their types are needed for both the normal C# code, which may try to reference them, and the embedded graph patterns, which definitely do. But for the latter the way the Mono C# Compiler processes the source files causes a problem: an embedded graph pattern may reference entities from the C# context. To generate an action from this pattern via the GrGen.NET frontend, the types of these entities must be known, but they are not calculated until the containing method becomes resolved during the Code Generation phase. But in the Code Generation phase it is too late to generate and add the classes, especially as it would be very inefficient to call the GrGen.NET frontend once for every single action. The way way I have chosen to solve this problem is to resolve methods containing embedded graph patterns and to generate and add the classes for the according actions before the Code Generation phase. So the (partially[1]) extended compiler uses the following new sequence of phases:

1. **Parsing:** All source files are parsed and the AST is built.

2. **Model Generation:** The graph models are generated using the GrGen.NET frontend, which produces new C# files, for which phase 1 has to be repeated.

[1]More phases might be necessary, when the other constructs are implemented, too.

3. **Loading Referenced Assemblies:** All referenced assemblies are loaded.

4. **Type Hierarchy:** A type hierarchy of empty types is created into a new assembly.

5. **Member Definition:** All types are filled with their fields and empty definitions for methods, properties, indexers, and events.

6. **Action Generation:** All graph rewrite actions are generated using the GrGen.NET frontend which again produces new C# files, for which phases 1, 4, and 5 have to be repeated.

7. **Code Generation:** All types are emitted into the assembly.

8. **Output:** The assembly is written to a file.

This order has a performance problem: The rather slow GrGen.NET frontend has to be called two times, because the graph model used by an embedded graph pattern must be known prior to its generation. In contrast to the explicit action declarations, the embedded patterns get information about the model to be used only implicitly through the type of the passed graph. But to be able to resolve this type, the type must already exist. Perhaps it may be possible to use pseudo-types for the graph models and replace them later by the types of the generated models. This way the GrGen.NET frontend could be called just one time. Although it is unclear, if this really works, it is definitely worth investigating.

Apart from the new phases, both the lexer and the parser must be extended as described in the following subsections.

5.2.1 Lexer

The lexer is responsible for transforming the character stream of the source code into a token stream, which can then be further processed by the parser. A token represents one or more characters, for example an identifier, a keyword, an operator, or a punctuation mark.

For G# we have to add several keywords for the new kinds of types and for the new statements, and some tokens for pattern edge declarations. Some keywords solely used inside action definitions like "`delete`" or "`eval`" are only used as keywords, when the lexer is in an action context. There, also three new tokens are valid, needed for edge parsing to disallow spaces in arrowheads: "`?-`", "`-?`", and "`<-`". For some constructs, it is not necessary to actually parse their contents, because it will just be written to a file and given to the GrGen.NET frontend. Therefore the new method `Tokenizer.read_block_body` reads the whole content of a block[2] into a string.

[2]i.e. without the braces

5.2.2 Parser

The parser reads the tokens provided by the lexer and tries to find out which productions of the grammar of the according language were used to generate the source code. “Tries”, because the source code may contain syntax errors, in which case there is no valid derivation.

As mentioned above, the Mono C# Compiler uses an LALR(1) parser, i.e. a shift-reduce parser with one token lookahead. A shift-reduce parser either pushes input tokens onto a stack (“shifts” them) or replaces elements at the top of the stack by other elements (“reduces” them). Whether the parser shifts or reduces depends on the current state of the parser and the lookahead of one token. The decision must always be unambiguous except for some shift/reduce conflicts which cannot be avoided[3]. Therefore the grammar has to be formulated accordingly to make it possible to generate a parser for it.

The used yacc grammar allows to specify user-defined code to be executed whenever a production or a part of it has been determined. In the user-defined code the Mono C# compiler builds up its intermediate representation. G# extends the C# grammar by quite a number of new productions. First it adds the four new kinds of type declarations, **model**, **rule**, **test**, and **pattern**. Here the content of the declarations is just read into a string using `Tokenizer.read_block_body` explained in the previous subsection, as we use the GrGen.NET frontend to build C# code for models and rules. So when we parse a model, we need to store the model name and its content, and when we parse an action, we need to store the action name, the used model, the parameter and return types, and its content.

The new statements need much more work than these declarations, as illustrated using the **match** statement: The **match** statement consists of a reference to a graph, an optional number of times to execute, an optional action modifier (**dpo** and such), the pattern to be searched for, a **modify** or **replace** or **do**[4] part, and an optional **else** part with an embedded statement. For this the pseudo code shown in listing 5.1 is built up in the AST, replacing the meta variables in angle brackets and the meta statements in double angle brackets accordingly. Line 1 requests a current version of the action associated to the given pattern. This way dynamically generated search plans can still be used, although there is no user-defined name for this action. Because the static version of the action for this pattern will not exist until before code generation, a pseudo expression is used for `action.Instance` which is not resolved before the code for the type is actually emitted, but in the meantime serves as an *IAction* object to allow normal type analysis. The parameters pseudo-array in line 4 refers to the C# variables used inside the pattern, so here variables from the C# context are passed to the graph rewriting context over the *curparams* array. The assignments are done in every iteration here, because the parameters could be changed by the **do** part. In line 9 graph elements from the graph rewriting context are passed to the C# context by creating local variables with the according names and initializations. The **else** part is only executed, when no match has been found in the first iteration of the **for** loop (see line 16). The other (currently implemented) statements work very similar.

[3]Like the famous if-if-else case.

[4]Containing an embedded statement to be executed for each found match

Listing 5.1: Pseudo code representation of code generated for a match statement

```
IAction <curaction> = <graph>.GetNewestActionVersion(<action.Instance>);
object[] <curparams> = new object[<numparams>];
for(int <forvar> = 0; <forvar> < <maxnumiter>; ++<forvar>) {
    <<FOREACH i>>: <curparams>[<i>] = <parameters[i]>;
    IMatches <curmatches> = <curaction>.Match(<graph>, 1, <curparams>);
    if(<curmatches>.Count != 0) {
        IMatch <curmatch> = <curmatches>.GetMatch(0);
        <<IF USED WITH DO>>
            <<LOCAL VARS WITH GRAPH ELEMENTS FROM PATTERN>>
            <<STATEMENT FROM DO>>
        <<ELSE>>
            <curaction>.Modify(<graph>, <curmatch>);
        <<ENDIF>>
    } else {
        <<IF ELSE PART SPECIFIED>>
            if(<forvar> == 0) {
                <else part>
            }
        <<ENDIF>>
        break;
    }
}
```

Chapter 6

Results and Evaluation

To evaluate this diploma thesis, we consider three aspects in this chapter: the practical advantage of using G# on the basis of two examples, the many improvements presented in the previous chapters, and performance measurements of the different GrGen.NET versions.

6.1 Benefit of Using G#

In this section the practical advantage of using G# is shown by comparing the implementation of some parts of the libFirm[Lin02] compiler library written in C with analogue implementations written in G#. We will see that the new language constructs can simplify the implementation of graph based applications in several places and make them much easier to understand.

6.1.1 Short Introduction to libFirm

libFirm is a library for compilers offering a graph-based intermediate representation (IR) of programs, thoroughly following the static single assignment (SSA) approach from high-level IR down to code generation. It provides several analyses, many optimizations and a functioning IA-32 backend. The compiler developer builds up the IR graph using the API of libFirm as part of the language-specific frontend. The elements of the IR graph describe the control flow and the data dependency of the program as well as the operations themselves. For example the integer addition of two integer values (not necessarily constants) represented by the IR nodes a and b is an *Add* node with an ordered set of operands a and b modelled as reversed data flow (df) edges[1]. The outgoing edges are accessible via indices through *get_irn_n* and the incoming edges through *get_irn_n_out*[2]. libFirm provides numerous convenience functions for all kinds of nodes to give the *get_irn_n* calls with specific indices some semantics. So e.g. $get\_binop\_left(n)$ yields the left operand of a binary operation n.

[1]In fact, they represent the data dependencies of the operations.

[2]That's not a typo, the function name is highly irritating.

6.1.2 Example: The Weight of a Method Parameter

As a first example we take a look at a function, which calculates the impact of a constant used as a given method parameter for inlining. If the "weight" of the parameter is high, i.e. large parts of the method could be simplified if this parameter was a constant, it may be beneficial to inline this method or at least create a specialized copy of it. The function assumes that the given IR node is constant, and checks for each usage of this node, whether the usage is then also constant in which case the function is called recursively for the according usages.

The following listing shows an excerpt of this function from *analyze_irg_args.c* leaving out some switch cases which could not be simplified and combining two related if statements into one for fairer comparison:

Listing 6.1: The C-version of the calc_method_param_weight function from libFirm

```c
static unsigned calc_method_param_weight(ir_node *arg) {
    int      i, j, k;
    ir_node  *succ, *op;
    unsigned weight = null_weight;

    /* We mark the nodes to avoid endless recursion */
    set_irn_link(arg, VISITED);

    for (i = get_irn_n_outs(arg) - 1; i >= 0; i--) {
        succ = get_irn_out(arg, i);

        /* We were here.*/
        if (get_irn_link(succ) == VISITED)
            continue;

        /* We should not walk over the memory edge.*/
        if (get_irn_mode(succ) == mode_M)
            continue;

        switch (get_irn_opcode(succ)) {
        case ...:   /* some uninteresting cases left out... */

        case iro_Tuple:
            /* unoptimized tuple */
            for (j = get_Tuple_n_preds(succ) - 1; j >= 0; --j) {
                ir_node *pred = get_Tuple_pred(succ, j);
                if (pred == arg) {
                    /* look for Proj(j) */
                    for (k = get_irn_n_outs(succ) - 1; k >= 0; --k) {
                        ir_node *succ_succ = get_irn_out(succ, k);
                        if (is_Proj(succ_succ) && get_Proj_proj(succ_succ) == j) {
                            /* found */
                            weight += calc_method_param_weight(succ_succ);
                        }
                    }
                }
37              }
38              break;
39
40          default: ... // left out
41          }
42      }
43      set_irn_link(arg, NULL);
44      return weight;
45  }
```

Lines 9–18 iterate to the next usage of *arg*, which has not been visited in the current recursion path, yet, and which is not reached by a memory edge. The tuples analysed in lines 25–37 always have as many operands as they have *Proj* nodes, each projecting out one of the tuple operands. Here the code searches for all *Proj* nodes corresponding to *arg* as tuple operand and calls *calc_method_param_weight* recursively on these *Proj* nodes.

An implementation in G# formulates the first part clearer and the second part much clearer with the help of graph patterns:

Listing 6.2: The G#version of the calc_method_param_weight function from libFirm

```
1   uint CalcMethodParamWeight(IR_node arg, int visID)
2   {
3       uint weight = (uint) ArgsWeight.Null;
4
5       // We mark the nodes to avoid endless recursion
6       graph.SetVisited(arg, visID, true);
7
8       // Iterate over all unvisited users of arg => succ
9       matchatonce(graph)
10      {
11          arg <-:df\mem- succ:IR_node;
12          if { !visited(succ, visID); }
13
14          do {
15              switch((NodeTypes) succ.Type.TypeID)
16              {
17                  case ...: // some uninteresting cases left out ...
18
19                  case NodeTypes.Tuple:
20                      /* unoptimized tuple */
21                      matchatonce(graph, *)
22                      {
23                          succ_succ:Proj -:df-> succ -dfarg:df-> arg;
24                          if { succ_succ.proj == dfarg.pos; }
25
26                          do {
27                              weight += CalcMethodParamWeight(succ_succ, visID);
28                          }
29                      }
30                      break;
31
32                  default: ... //left out
33              }
34          }
35      }
36      graph.SetVisited(arg, visID, false);
37      return weight;
38  }
```

The iteration over the usages is done by lines 9–14 clearly not allowing traversing memory edges ("`<-:df\mem-`") and requiring an element which has not been visited, yet. Searching for the corresponding *Proj* node results in the very simple pattern matching in lines 21–26.

6.1.3 Example: Finding a Rotl Pattern

The "iropt.c" source file of libFirm contains over 6.000 lines of transformations simplifying and normalizing the intermediate representation of a program. For example it transforms $-(a - b)$ to $b - a$, a & $(a\,\hat{}\,b)$ to a & $\tilde{}b$, and $(X - a) == (X - b)$ to $a == b$. Writing those transformations is very tedious and repetitive: many conditions have to be met for the transformation to be applicable, leading to many if statements followed by a return statement. Many operations are commutative and/or associative requiring more very similar alternative conditions. Working with constants always requires several function calls. All these points make the transformations hard to read and understand, as the following listing of a transformation shows, which searches for some patterns of the *Rotl* (Rotate left) instruction starting with a given *Or* node:

Listing 6.3: The C-version of the transform_node_Or_Rotl function from libFirm

```
/**
 * Optimize an Or(shl(x, c), shr(x, bits - c)) into a Rotl
 */
static ir_node *transform_node_Or_Rotl(ir_node *or) {
    ir_mode *mode = get_irn_mode(or);
    ir_node *shl, *shr, *block;
    ir_node *irn, *x, *c1, *c2, *v, *sub, *n, *rotval;
    tarval *tv1, *tv2;

    if (! mode_is_int(mode))
        return or;

    shl = get_binop_left(or);
    shr = get_binop_right(or);

    if (is_Shr(shl)) {
        if (!is_Shl(shr))
            return or;

        irn = shl;
        shl = shr;
        shr = irn;
    } else if (!is_Shl(shl)) {
        return or;
    } else if (!is_Shr(shr)) {
        return or;
    }
    x = get_Shl_left(shl);
    if (x != get_Shr_left(shr))
        return or;

    c1 = get_Shl_right(shl);
    c2 = get_Shr_right(shr);
    if (is_Const(c1) && is_Const(c2)) {
        tv1 = get_Const_tarval(c1);
        if (! tarval_is_long(tv1))
            return or;

        tv2 = get_Const_tarval(c2);
        if (! tarval_is_long(tv2))
            return or;

        if (get_tarval_long(tv1) + get_tarval_long(tv2)
                != (int) get_mode_size_bits(mode))
            return or;

        /* yet, condition met */
        block = get_nodes_block(or);

        n = new_r_Rotl(current_ir_graph, block, x, c1, mode);

        DBG_OPT_ALGSIM1(or, shl, shr, n, FS_OPT_OR_SHFT_TO_ROTL);
        return n;
    }

    if (is_Sub(c1)) {
        v      = c2;
        sub    = c1;
        rotval = sub; /* a Rot right is not supported, so use a rot left */
    } else if (is_Sub(c2)) {
        v      = c1;
        sub    = c2;
        rotval = v;
    } else return or;

    if (get_Sub_right(sub) != v)
        return or;

    c1 = get_Sub_left(sub);
    if (!is_Const(c1))
        return or;

    tv1 = get_Const_tarval(c1);
```

```
74      if (! tarval_is_long(tv1))
75          return or;
76
77      if (get_tarval_long(tv1) != (int) get_mode_size_bits(mode))
78          return or;
79
80      /* yet, condition met */
81      block = get_nodes_block(or);
82
83      n = new_r_Rotl(current_ir_graph, block, x, rotval, mode);
84
85      DBG_OPT_ALGSIM0(or, n, FS_OPT_OR_SHFT_TO_ROTL);
86      return n;
87  }  /* transform_node_Or_Rotl */
```

As the *Or* operation is commutative, lines 13–27 are required to get its *Shl* and *Shr* operands. Lines 28–30 make sure both shift operations use the same left operand. Lines 35–45 check whether the constant right operands of the shift operations add up to the size of the according mode (e.g. 32 bits for an *int*32_*t*). Lines 56–64 determine which shift operation has a *Sub* as an operand.

Using the graph patterns of G# and a more convenient representation of constants[3] the above example can be written in a very clear and concise way:

Listing 6.4: The G#-version of the transform_node_Or_Rotl function from libFirm

```
pattern PatRotSub(sub:Sub, var bits:int, v:IR_node)
{
    sub -:df_left->  c:ConstLong;
    sub -:df_right-> v;
    if { c.Value == bits; }
}

/// <summary>
/// Optimize an Or(shl(x, c), shr(x, bits - c)) into a Rotl
/// </summary>
public IIR_node TransformNodeOrRot(IIR_node or)
{
    IMode mode = or.GetMode(irg);
    if(!mode.IsInt) return or;

    match(graph)
    {
        or -:df_operand-> shl:Shl -:df_left-> x:IR_node;
        or -:df_operand-> shr:Shr -:df_left-> x;

        alternative {
            const_const {
                hom(c1, c2);
```

[3]As LIBFIRM is written in C it has no access to object-oriented features and cannot use operator overloading.

```
24                 shl -:df_right-> c1:ConstLong;
25                 shr -:df_right-> c2:ConstLong;
26                 if { c1.Value + c2.Value == mode.SizeBits; }
27 
28                 do {
29                     return irg.NewRotl(or.GetBlock(irg), x, c1, mode);
30                 }
31             }
32             sub_value {
33                 shl -:df_right-> sub:Sub;
34                 shr -:df_right-> v:IR_node;
35                 :PatRotSub(sub, mode.SizeBits, v);
36 
37                 do {
38                     // A Rot right is not supported, so use a rot left with sub = bits - v
39                     return irg.NewRotl(or.GetBlock(irg), x, sub, mode);
40                 }
41             }
42             value_sub {
43                 shl -:df_right-> v:IR_node;
44                 shr -:df_right-> sub:Sub;
45                 :PatRotSub(sub, mode.SizeBits, v);
46 
47                 do {
48                     // A Rot left
49                     return irg.NewRotl(or.GetBlock(irg), x, v, mode);
50                 }
51             }
52         }
53     }
54     return or;
55 }
```

By using the general *df_operand* edges, both operand orders and the left operand of the shifts are checked in just two lines (lines 18–19) instead of sixteen lines (lines 13–30 of listing 6.3 ignoring empty lines). The *ConstLong* nodes make it much easier to ensure, that the constant right operands of the shift operations add up to the correct number of bits (line 26). The three cases of this pattern can be clearly recognized in the named **alternative** cases. With the help of the *PatRotSub* pattern the difference between the alternatives *sub_value* and *value_sub* is clear at first sight. If the conditions specified by the patterns are not met, the program will just find no match and return to the caller in line 54. That means two returns for the negative case instead of thirteen in the libFirm version.

6.1.4 Evaluation

To compare the productivity of G# and C, table 6.1 not only shows the number of lines used by the implementations of both examples, but also the number of tokens. Although "number of lines" is easier to understand, it depends very much on coding style and does not account for the length of a line. The “number of tokens”, on the other hand, ignores comments, whitespaces, newlines, and the length of words. It is only vulnerable to unnecessary braces due to coding style. Therefore, it is a good means of measuring the structural verbosity of the implementations.

So, looking at table 6.1 we see, that the G# notation is much more concise. The G# version of the “Rotl” example requires 34% less tokens than the C version *and* is easier to understand.

Example	C Implementation		G# Implementation		C → G#	
	Lines	Tokens	Lines	Tokens	Lines	Tokens
Param Weight	45	210	38	149	-16%	-29%
Rotl	87	438	55	287	-37%	-34%

Table 6.1: Comparison of two examples between C and G# implementations

6.2 Usability Improvements

The usability of a programming system heavily depends on two factors: The convenience of solving even complex problems and type safety. A very expressive and convenient system is of no use, if the time saved during the first development stage has to be spent with tedious debugging of hidden type errors; especially, if the type errors induce very subtle problems.

As shown in table 6.2 (copied from the previous chapter), the convenience and type safety of GrGen.NET without G# has already been significantly improved in the categories “Model” and “Rules”. Especially the assimilation of the attributes into the element types was a big step forward. But with G# there is little left to be desired as rules can be handled much easier and safer with embedded rules, method-like rule calls, and XGRSs. Also depth-first-order graph traversals can now be specified easily and concisely. Only the handles of visited flags are still not type-safe, because simple integers are used instead of heavy-weight yet type-safe objects.

Comparing XL with GrGen.NET/G# we see, that GrGen.NET can now very well compete with XL in expressiveness and convenience and is even superior in the categories “Model” and “Rules”. Especially the fully featured graph models and the tuple notation for multiple return values are important advantages over XL.

	XL		**GrGen.NET**					
			1.3.1		New		with G#	
	Con	TS	Con	TS	Con	TS	Con	TS
Model:								
Graph	o	+	+	+	++	+	++	+
Multiple graph models (s/A)	o	+	× /o	× /-	× /+	× /+	+	+
Graph structure assertions	×	×	+	+	+	+	+	+
Node types (spec/API)	o	+	+	+/-	+	+	+	+
Edge types (spec/API)	- -	-	+	+/-	+	+	+	+
Attribute access (spec/API)	++	+	++/- -	+/-	++	+	++	+
Constructors (node/edge)	+/×	+/×	×	×	+	+	+	+
Element methods (node/edge)	+/×	+/×	×	×	×	×	+	+
Embedding	+	+	×	×	×	×	+	+
Rules:								
Modes	+	+	o	+	+	+	+	+
LHS and RHS	o	+	o	+	+	+	+	+
Parameters (spec/API)	++	+	o	+/-	+	+/-	++	+
Returns (spec/API)	-	+	o/-	+/-	+	+/-	++	+
Outer access to LHS	++	+	-	-	-	-	++	+
Outer access to RHS	o	+	×	×	×	×	+	+
Embedding	+	+	×	×	×	×	+	+
Rule execution:								
Graph queries	+	+	×	×	×	×	×	×
XGRS	×	×	-	-	-	-	+	+
Boolean combination	-	+	×	×	-	-	+	+
- without returns	+	+	-	-	-	-	+	+
Graph traversal:								
Visited flags	+	-	×	×	+	-	+	-
DFS	- -	-	×	×	×	×	+	+
Other traversals	×	×	×	×	×	×	×	×

Table 6.2: Comparison of different problems solved by XL, GrGen.NET 1.3.1, GrGen.NET including the proposed changes, and GrGen.NET with G# in terms of convenience (Con) and type safety (TS) (identical to table 4.2)

6.3 Performance of GrGen.NET

In chapter 4 several improvements to GrGen.NET were introduced: The refactoring of the type architecture (see 4.1.2) and the most important part of the new type-safe element representation (see 4.1.1) was implemented in v1.4. The rest of the latter change, support for non-graph-element parameters and return values (see 4.2.2), and the visited flags (see 4.2.3) have been implemented in v2.0. The type-safe graphs (see 4.1.3) and the element constructors (see 4.2.1) have not been implemented, yet. Additionally, Sebastian Buchwald implemented undirected edges and DPO-matching in v1.4 [Buc08] and Edgar Jakumeit rewrote the matcher generator [Jak07] in v1.4 and added subpatterns, alternative patterns, and arbitrary nested negative application conditions (NACs) in v2.0 [Jak08]. All these changes added great features and improved the convenience for the user, but some of them have a deep impact on the graph rewriting engine. So it is a valid question whether the convenience comes at the cost of performance.

6.3.1 Memory Usage

The memory usage per graph element of the different versions of GrGen.NET is shown in table 6.3 divided into values for 32-bit and 64-bit .NET environments. While "Node" and "Edge" elements do not have any attributes, "Attributed Node" and "Attributed Edge" elements contain an integer (32 bit), a string reference (size of a pointer) and three booleans (each 8 bit), which sums up to 11 bytes for 32-bit and 15 bytes for 64-bit. In a 32-bit environment the memory alignment is 4 byte and in a 64-bit environment 8 byte, thus the objects are accordingly padded.

The graph element sizes have been measured on the basis of the creation of four million accordingly typed elements. The creation loop was surrounded by calls to the common language runtime (CLR) method `GC.GetTotalMemory(true)`, which forces a garbage collection and then returns the "number of bytes currently thought to be allocated" [Mic08]. Because of the large number of allocated elements, this measurement method yields reliable results.

As shown in table 6.3 the only change in memory usage happened between v1.3.1 and v1.4: Nodes and edges are generally one pointer size and 8 bytes smaller, i.e. 12 bytes for 32-bit and 16 bytes for 64-bit. Due to the type-safe graph elements the reference to the external attribute object has been removed and due to changes by Edgar Jakumeit[4] three other fields were combined into one flags field. Graph elements using attributes additionally require two pointers less memory, because the overhead of the extra attribute object does not exist anymore[5].

6.3.2 Running Time

Table 6.4 presents the running times of several benchmarks executed by version 1.2, 1.3.1, 1.4, and 2.0 of GrGen.NET. The "v2.0 *" column avoids most interpreted extended graph

[4]Within the scope of the preparation of nested NACs.

[5]For elements without attributes such objects were never allocated, of course.

Element kind	Bytes per element v1.2	v1.3.1	v1.4	v2.0
32-Bit Process:				
Node	44	44	32	32
Attributed Node	64	64	44	44
Edge	60	60	48	48
Attributed Edge	80	80	60	60
64-Bit Process:				
Node	80	80	64	64
Attributed Node	112	112	80	80
Edge	112	112	96	96
Attributed Edge	144	144	112	112

Table 6.3: Memory usage per graph element of several GrGen.NET versions

rewrite sequences (XGRS) by using compiled XGRSs in form of rules only containing an **exec** statement on the RHS. Although the XGRS compiler was mainly implemented for the G# **exec** statement, it also proves worthy for normal rules, as the last column shows. The given running times are the means of 40 consecutive measurements without the upper and lower 0.2-quantiles[6]. The "mean relative change" is the mean of the relative changes from the base version of this work, version 1.3.1, to the according version. If all benchmarks would take twice the time as in version 1.3.1, the mean relative change would be 2. Version 1.2, the first stable release of GrGen.NET, is just listed for completeness and to show the overall trend.

Due to the many changes from v1.3.1 to v2.0 it is unclear, which changes caused v2.0 to need 11.5% more running time than v1.3.1 on average (without using compiled XGRSs). One reason might be that most graph elements cannot be directly reused during a rewrite step, but are stored in a cache for later reuse (see section 4.1.1). Although this is far less expensive than allocating a new object, it is less efficient, than just exchanging the type and attribute information. The Dragon and Sierpinski benchmarks are very memory consuming, so the smaller graph elements are probably the reason for the better running time here.

A benchmark with very interesting running times is "Busybeaver3". In contrast to the "Busybeaver" variant, it makes extensive use of graph variables and element attributes. The GrGen.NET 2.0 version with compiled XGRSs only needs about 23% of the running time required by the version with interpreted XGRSs. The most important difference with respect to running time is, that in the compiled version the graph variables, which are stored in a hash map of the graph object, are replaced by local variables. For the over 4.7 million applied graph transformations each using at least one graph variable, this was obviously the dominating factor, when you see the much smaller running time reduction to about 69% between the interpreted and compiled "Busybeaver" variant, where over

[6]Measured on an Intel Core 2 Quad at 2.4 GHz with 2 GB RAM running Windows Vista 64 SP1 with the Microsoft .NET Framework 2.0.50727.1434 in 64-bit mode

7 million transformations only take advantage of the non-interpreted code.

For the "Mutex" benchmark of size 10,000 and the "Ludo2" benchmark, the compiled XGRS versions need more time than the interpreted ones. The reason is that now both the XGRS interpreter as well as the compiled XGRS code has to be just-in-time compiled, because the interpreter is still used to call the rules containing the compiled XGRSs. But for "longer" taking benchmarks (approx. $>$ 500 ms) this additional overhead becomes irrelevant with respect to the performance gain.

So all in all, with the help of compiled XGRSs it is possible to get the improved convenience and type safety for free.

		Time in ms for GrGen.NET				
Benchmark	Size	v1.2	v1.3.1	v1.4	v2.0	v2.0 *
Mutex	10,000	291	301	396	382	391
	100,000	614	647	754	756	713
	1,000,000	4,205	4,393	4,961	4,973	4,655
Dragon	8	2,781	2,911	2,903	2,872	2,512
	9	10,981	11,065	10,869	10,615	9,673
Sierpinski	12	2,593	2,483	3,093	3,048	2,987
	13	8,807	8,724	8,589	8,213	8,136
Busybeaver	5 No. 7	3,084	3,102	3,982	3,966	2,720
Busybeaver3	5 No. 7	7,674	9,278	9,686	8,544	1,933
Ludo2 (random seed=)	98754321	183	185	256	235	260
Total time		41,213	43,089	45,489	43,604	33,980
Mean relative change		0.969	1	1.153	1.115	0.982

Table 6.4: Performance comparison between several GrGen.NET versions on the basis of the running time of several benchmarks in a 64-bit environment

Chapter 7

Conclusion and Prospects

7.1 Conclusion

The main purpose of this work was to significantly improve the development of graph transformation based GrGen.NET applications regarding convenience, type safety, and productivity. In order to achieve this goal, firstly, the types provided by the API were rigorously refactored leading to type-safe and much more convenient handling of graphs, graph element types, graph elements, and their attributes. Secondly, new features have been introduced to GrGen.NET simplifiying element initialization, value passing between graph rewrite rules, and implementations of graph traversals a lot. Finally, the introduction of the embedded domain-specific language (EDSL) G# makes using graphs and graph rewriting in applications much more comfortable, and fills up the remaining issues with type safety.

Because of the timing constraints of this work, I searched for a simple solution to implement a compiler for G# and started by extending an existing C# source-to-source compiler to keep the required effort low. Sadly this didn't work out, so I switched to extending the heavy-weight Mono C# Compiler. The resulting partial implementation of the G# compiler shows that it is possible to generate a compiler for a C#-based EDSL this way. Without the detour of the first approach, the compiler implementation even would have been finished during this work. But perhaps it would have been better to develop the compiler using the MetaBorg approach [BV04], despite the need to create a whole C# SDF grammar and type attribution. Due to the domain-specific languages used by this approach, the result would have been much easier to understand, maintain, and extend.

Nonetheless, the already implemented improvements have an huge positive impact on the usability and type safety of GrGen.NET while retaining its high performance.

7.2 Prospects

As the compiler has not been finished during this work, there is still some work to do. But apart from this, there are some points which could use some further attention:

- Inheritance on graph models would allow rules specified e.g. for an XML graph model to be also usable with an SVG model.
- To make the parameters and return values of actions (from the API point of view) type-safe, they could be handled using normal method parameters and out/ref parameters instead of an object array.
- Special match types could be generated providing more convenient access to LHS elements via their names.
- More graph traversals could be supported by G#, like breadth-first-search or some weighted searchs.
- Perhaps some sort of XL-like graph query in combination with LINQ query operators could be interesting for G#.
- It might be desirable to bring the exec syntax a bit more to the normal expression level to be able to write "`(curAnt) = graph.Food2Ant(curAnt)* | [graph.EvaporateWorld]);`" instead of "`exec(graph, (curAnt) = Food2Ant(curAnt)* | [EvaporateWorld]);`". It is unclear though, whether it is possible to integrate something like this into G# because of the already existing operators and their precedences.

Acknowledgements

I would like to especially thank the following persons: Rubino Geiß for supervising me despite his lack of time; Edgar Jakumeit for his many constructive comments and the great discussions; Christoph Mallon for his great help with the C# Parser project and the many interesting and less interesting discussions ;D ; Michael Beck for supervising me in times I needed it the most; Sebastian Buchwald for his comments on some language constructs; Tom Gelhausen for taking the time to talk about this work (especially the **fordepth** statement) in an early phase of it; Jakob Blomer for providing the benchmark script.

Also I would like to thank Prof. Goos for keeping the IPD going even long after being emeritus and for the great support for GrGen.NET, and Prof. Snelting for supporting our project and allowing us to represent our university at the GraBaTs 2008 in Leicester, although his field of research does not directly deal with graph rewriting.

Finally, many thanks to my wife, Ursula, for her support especially in form of cakes for the IPD, to all who participated in the after work finger exercises, and to Chuck the Plant!

Appendix A

Language Definition

Embedded GrGen.NET (G#) extends the programming language C# by several domain-specific language constructs to increase the convenience and type safety of graph rewrite applications. In this chapter the syntax and the semantics of these constructs are defined.

A.1 C# Grammar Extensions

The new G# constructs extend the C# grammar [Ecm06] productions as follows:

```
type-declaration:
        ...
    |   gs-model-declaration
    |   gs-rule-declaration
    |   gs-test-declaration
    |   gs-pattern-declaration

embedded-statement:
        ...
    |   gs-match-statement
    |   gs-matchatonce-statement
    |   gs-matchaction-statement
    |   gs-matchactionatonce-statement
    |   gs-modify-match-statement
    |   gs-replace-match-statement
    |   gs-modify-statement
    |   gs-fordepth-statement

primary-no-array-creation-expression:
        ...
    |   gs-tuple-call-expression
    |   gs-exec-expression
```

A.1.1 Definitions

In this section some definitions are introduced which are referred to in the rest of this chapter.

```
gs-graph-expression:
    expression
```

The type of this expression must be a class which implements *IGraph* and *IGraphModel*.

```
gs-model-type:
    identifier
```

The name of a graph model.

```
gs-range-expression:
    "["
    (
        "*"
    |
        "+"
    |
        min=expression [ ":" (max=expression | "*") ]
    )
    "]"
```

Defines a range of `min` to `max`. For “`[*]`” `min` is zero, while it is one for “`[+]`”. For both `max` is infinity encoded as $Int32.MaxInt$. “`[<min>:*]`” sets `max` to infinity as well. For “`[<min>]`”, `max` is set to `min`.

```
gs-action-modifier:
    "dpo" | "induced" | "exact"
```

gs-action-modifier specifies how a pattern is to be matched. “**dpo**” means, that all edges of deleted nodes must be specified. “**induced**” means, that all edges between the nodes of the pattern must be specified for a match to be found. “**exact**” rules match only, when there are no unspecified edges connecting to the nodes of the pattern. See [Buc08] for more details.

```
gs-param:
    (
        name=identifier ":" type=identifier
    |
        "var" name=identifier ":" type=identifier
    |
        "-" name=identifier ":" type=identifier "->"
    |
        "-" name=identifier ":" type=identifier "-"
    |
        "<-" name=identifier ":" type=identifier "->"
    |
        "?-" name=identifier ":" type=identifier "-?"
    )

gs-params:
    gs-param ( "," gs-param )*
```

Declares a parameter of an according kind with the identifier `name` and the type `type`.

```
gs-returnparam:
    type=identifier

gs-returnparams:
    gs-returnparam ( "," gs-returnparam )*
```

Declares a return parameter with the type `type`.

```
gs-call-return-argument:
    name=identifier [ ":" type=identifier ]
```

Specifies and optionally declares a return argument. The optional part must be specified if and only if `name` is not already declared in the current scope. In this case, a new local-variable with the identifier `name` and the type `type` is added to the current scope.

A.1.2 Model Declarations

```
gs-model-declaration:
    opt-attributes
    "public" "model" name=identifier
    "{" grgen-model-declbody "}"
```

A **gs-model-declaration** declares a model with the name `name`. A model type implements both *IGraph* and *IGraphModel* and provides convenience methods to create elements for every type of the model. The **public** modifier is here for upward compatibility, when other access modifiers might be implemented.

A.1.3 Rule Declarations

```
gs-rule-declaration:
    opt-attributes
    "public" [ gs-action-modifier ] "rule" "<" model=gs-model-type ">" identifier
    [ "(" params=gs-params ")" ]
    [ ":" "(" returns=gs-returns ")" ]
    "{" gs-pattern-body (gs-modify-body | gs-rewrite-body) "}"
```

Declares a GrGen rule according to the GrGen syntax for the graph model specified by `model`. The **public** modifier is here for upward compatibility, too.

A.1.4 Test Declarations

```
gs-test-declaration:
    "public" [ gs-action-modifier ] "test" "<" model=gs-model-type ">" identifier
    [ "(" params=gs-params ")" ]
    [ ":" "(" returns=gs-returns ")" ]
    "{" gs-pattern-body "}"
```

Declares a GrGen test according to the GrGen syntax for the graph model specified by `model`. Again the **public** modifier is given for upward compatibility.

A.1.5 Pattern Declarations

```
gs-pattern-declaration:
    "public" [ gs-action-modifier ] "pattern" "<" model=gs-model-type ">" identifier
    [ "(" params=gs-params ")" ]
    "{" gs-pattern-body (gs-modify-body | gs-replace-body) "}"
```

Declares a GrGen pattern according to the GrGen syntax for the graph model specified by `model`. The **public** modifier is here for upward compatibility, too.

A.1.6 The Match Statement

```
1 gs-match-statement:
2     "match" "(" gs-graph-expression [ "," (max=expression | "*") ] [ "," gs-action-modifier ] ")"
3     "{"
4     gs-pattern-body
5     gs-matched-statement
6     "}"
7     [ "else" gs-matched-statement-noreplace ]
```

The **gs-match-statement** searches for a match up to `max`-times, where the type of `max` must be an integer. If `max` is not specified, it defaults to one. If a "*" was specified instead, `max` is set to infinity. Whenever a match has been found, it executes line 5 once, where the number of already found matches can be accessed by the special variable **nummatches**, the current match by the special variable **curmatch**, and the pattern elements by their names as local variables of the according interface types in the scope of the **match** block. If "**dpo**" was specified as `gs-actions-modifier`, `gs-matched-statement` must be either a `gs-modify-statement` or a `gs-replace-statement`. If no matches were found and an else part was specified (line 7), this is executed instead.

The pattern is specified directly in GrGen syntax (line 4). The given graph acts as the host-graph for the matching process. It is not guaranteed, that a different match is found in a next iteration step, if the previous match has not been altered in such a way, that the pattern does not match anymore.

A.1.7 The Match-At-Once Statement

```
1 gs-matchatonce-statement:
2     "matchatonce" "(" gs-graph-expression [ "," gs-range-expression ]
3                       [ "," gs-action-modifier ] ")"
4     "{"
5     gs-pattern-body
6     gs-matched-statement
7     "}"
8     [ "else" gs-matched-statement-noreplace ]
```

The **gs-matchatonce-statement** searches for at least `min` and up to `max` matches as a snap-shot, i.e. saves all matches before continuing. `min` and `max` are the minimum and maximum values of the `gs-range-expression`, respectively. For each found match, it then executes line 6 once, where all found matches can be accessed by the special *IMatches* variable **allmatches**, their number by a special integer variable **nummatches**, the current match by the special *IMatch* variable **curmatch**, and the pattern elements by their names as normal local variables of according types in the scope of the **matchatonce** block. If "**dpo**" was specified as `gs-action-modifier`, `gs-matched-statement` must be either a `gs-modify-statement` or a `gs-replace-statement`. If the number of found matches is less than `min` and an else part was specified (line 8), this is executed instead. If `min` is zero, the else part will never be executed.

The pattern is specified directly in GrGen syntax (line 5). The given graph acts as the host-graph for the matching process. If line 6 changes another still unprocessed match, the match may not be valid anymore, when it becomes processed. Elements may have been deleted, attribute or negative application conditions may be violated. It is the user's responsiblity to handle these problems.

A.1.8 The Match-Action Statement

```
1 gs-matchaction-statement:
2     "matchaction" "(" gs-graph-expression "," gs-action-call [ gs-range-expression ] ")"
3     gs-matched-statement
4     [ "else" gs-matched-statement-noreplace ]
5
6 gs-action-call:
7     [ "(" gs-call-return-argument ( "," gs-call-return-argument )* ")" "=" ]
8     [ "?" ] ActionName
9     [ "(" [ expression ( "," expression )* ] ")" ]
```

A **gs-matchaction-statement** executes a given action up to `max`-times, where `max` is the maximum value of the `gs-range-expression` whose minimum value must be one. If no `gs-range-expression` is specified, it defaults to "`[1]`". Whenever a match has been found, it executes line 3 once, where the number of already found matches can be accessed through a special variable **nummatches**, and the current match by the special variable **curmatch**.

Any variables declared as a `gs-call-return-argument` are only valid in the "then"-part. If no matches were found and an else part was specified (line 4), this is executed instead.

The **gs-action-call** references a GrGen test or a "testified" rule with optional arguments and return parameters. A testified rule is a rule which is prefixed by a "`?`" meaning that the right hand side of the rule (including any return statements) has to be ignored. So no return parameters may be specified for a testified rule.

The given graph acts as the host-graph for the matching process. It is not guaranteed, that a different match is found in a next iteration step, if the previous match has not been altered in such a way, that the pattern does not match anymore.

A.1.9 The Match-Action-At-Once Statement

```
1 gs-matchactionatonce-statement:
2     "matchactionatonce" "(" gs-graph-expression "," gs-action-call [ gs-range-expression ] ")"
3     gs-matched-statement
4     [ "else" gs-matched-statement-noreplace ]
```

A **gs-matchactionatonce-statement** searches for the pattern of a given action for at least `min` and up to `max` matches as a snap-shot, i.e. saves all matches before continuing. `min` and `max` are the minimum and maximum values of the `gs-range-expression`, respectively. If no `gs-range-expression` is specified, it defaults to "`[+]`". For each found match, it then executes line 3 once, where all found matches can be accessed by the special *IMatches* variable **allmatches**, their number by a special integer variable **nummatches**, and the current match by the special *IMatch* variable **curmatch**.

Any variables declared as a `gs-call-return-argument` are only valid in the "then"-part. If no matches were found and an else part was specified (line 4), this is executed instead.

The **gs-action-call** references a GrGen test or a "testified" rule with optional arguments and return parameters just like in section A.1.8.

The given graph acts as the host-graph for the matching process. If line 6 changes another still unprocessed match, the match may not be valid anymore, when it becomes processed. Elements may have been deleted, attribute or negative application conditions may be violated. It is the user's responsiblity to handle these problems.

A.1.10 The Matched Statement

```
1  gs-matched-statement:
2           gs-modify-match-statement
3       |   gs-replace-match-statement
4       |   "do" embedded-statement
5
6  gs-matched-statement-noreplace:
7           gs-modify-match-statement
8       |   "do" embedded-statement
9
10 gs-modify-match-statement:
    "modify" "{" GrGenModifyBodyWithoutReturn "}"

gs-replace-match-statement:
    "replace" "{" GrGenReplaceBodyWithoutReturn "}"
```

The **gs-modify-match-statement** applies modifications to the current match of the nearest enclosing `gs-match-statement`, `gs-matchatonce-statement`, `gs-matchaction-statement`, or `gs-matchactionatonce-statement`. The **gs-replace-match-statement** analogously replaces such a current match. It is a compile-time error, if there is no nearest enclosing match statement.

The modifications and the replacement are specified according to the GrGen syntax of a **modify** or **replace** part without **return** and **exec** statements. Before each execution of a **gs-modify-match-statement** and **gs-replace-match-statement** it is assured that all referenced graph elements exist. Otherwise, an *InvalidOperationException* is thrown.

The "**do**" version executes the given `embedded-statement`, which may also include multiple modify and replace statements.

A.1.11 The Modify Statement

```
gs-modify-statement:
    "modify" "(" gs-graph-expression ")" "{" GrGenModifyBodyWithoutReturn "}"
```

The **gs-modify-match-statement** applies modifications to the given graph. The modifications are specified according to the GrGen syntax of a **modify** part without **return** and **exec** statements. Before each execution of a **gs-modify-match-statement** it is assured that all referenced graph elements exist. Otherwise, an *InvalidOperationException* is thrown.

A.1.12 The ForDepth Statement

```
gs-fordepth-statement:
    "fordepth" "(" type identifier "in" gs-graph-expression
        "at" startnode=expression ")"
    [
        "along" ( gr-embedded-pattern | gr-action-call | "do" embedded-statement )
    |
        "child" ( gr-embedded-pattern | gr-action-call | "do" embedded-statement )
        "next" ( gr-embedded-pattern | gr-action-call | "do" embedded-statement )
    |
        "next" ( gr-embedded-pattern | gr-action-call | "do" embedded-statement )
        "child" ( gr-embedded-pattern | gr-action-call | "do" embedded-statement )
    ]
    (
        "pre" ( gs-modify-match-statement | gr-action-call | "do" embedded-statement )
        [ "post" ( gs-modify-match-statement | gr-action-call | "do" embedded-statement ) ]
    |
        "post" ( gs-modify-match-statement | gr-action-call | "do" embedded-statement )
    )
```

The **gs-fordepth-statement** runs a depth-first-search (DFS) over the given graph starting at the node `startnode`. The DFS can be run in two modes: In the "**along**" mode all children of the current node are determined by a pattern, an action call, or an `embedded-statement` "**yield return**"-ing them. In the "**child**"/"**next**" mode the first child of the current node is analogously determined by the "**child**" line and each following child is determined by the "**next**" line. The current node is available through the `identifier` variable of type `type` introduced by the **fordepth**-statement. For the "**next**" line the previous child is available as **prevchild**. In every case the special variable **nextnode** has to be assigned the next node to be processed. If neither "**along**" nor "**child**"/"**next**" parts are specified, it defaults to "**along**" mode with the pattern "`identifier ?-:AEdge-? nextnode:type`", which matches all nodes of type `type` adjacent to the current node.

While walking the graph, first the current node (starting with `startnode`) is marked as visited, then the "**pre**" part is executed, then all unvisited children are visited by the search, and last the "**post**" part is executed. The visitor ID used for marking the nodes is available via the special variable **curvisitorid**. In the "**pre**" and "**post**" parts the `break-statement` causes the whole DFS to terminate and the `continue-statement` skips any further processing of the current node and their children. Unprocessed children will not be marked as visited. If the current node is removed in the "**pre**" part, a `continue-statement` is issued. If the current node is removed while walking the children, a `continue-statement` is issued after the DFS returns from the current child. New children of the current node will be processed when they are added in the "**pre**" part or while walking the children.

A.1.13 The Tuple Call Expression

```
gs-tuple-call-expression:
    "(" gs-tuple-var ( "," gs-tuple-var )* ")" "?=" expression

gs-tuple-var:
    [ type=identifier ] name=identifier
```

The **gs-tuple-call-expression** executes a rule given by `expression` and only assigns the return values to the according `gs-tuple-vars`, if the rule succeeds. In this case, this expression evaluates to **true**, otherwise to **false** .

A.1.14 The Exec Expression

```
gs-exec-expression:
    "exec" "(" gs-graph-expression, gs-xgrs ")"
```

The **gs-exec-expression** executes the given `gs-xgrs` on the given graph and evaluates to a boolean indicating success (**true**) or failure (**false**) of the rewrite sequence.

A.2 C# Semantic Extensions

A.2.1 Foreach on Graphs

```
foreach-statement:
    "foreach" "(" type identifier "in" expression ")" embedded-statement
```

According to 15.8.4 of the C# language specification [Ecm06] an expression with a type implementing multiple `System.Collections.Generic.IEnumerable<T>` interfaces is not allowed to appear as an expression of a **foreach-statement**. But to allow an easy way of iterating over elements of a graph compatible to a given type, the semantics of the **foreach-statement** is extended. For expressions fulfilling the requirements of `gs-graph-expression` (see section A.1.1) the given type determines over which elements the foreach loop should iterate. The read-only iteration variable is declared by `type` and `identifier`. If the given type is an interface and inherits from *IGraphElement*, the loop iterates over all compatible elements. If the given type is a class and inherits from *IGraphElement*, the loop iterates over all elements with exactly the given type. Otherwise, a compile-time error is produced.

Appendix B

Examples

This chapter contains three (simple) examples written in G# which already work with the so far extended Mono C# Compiler.

Listing B.1: A G# version of the GrGen.NET "Sierpinski3" example

```
using System;
using de.unika.ipd.grGen.libGr;
using de.unika.ipd.grGen.lgsp;

using de.unika.ipd.grGen.Model_Sierpinski;

namespace testSierpinski
{
    public model Sierpinski
    {
        node class A;
        node class B;
        node class C;

        node class AB extends A,B;
        node class BC extends B,C;
        node class CA extends C,A;
    }

    public class TestSierpinski
    {
        public static void PrintGraphInfo(Sierpinski graph)
        {
            Console.WriteLine("\nNum nodes: " + graph.NumNodes
                            + "\nNum edges: " + graph.NumEdges);
        }

        public static void Main(String[] args)
        {
            int n;
            if(args.Length != 1 || !int.TryParse(args[0], out n))
            {
```

```
            Console.WriteLine("Usage: testSierpinski <N>");
            return;
        }

        Sierpinski graph = new Sierpinski();

        // init rule
        modify(graph)
        {
                     a:A;

                c-->a;  a-->b;

            c:C;    c<--b;    b:B;
        }

        PrintGraphInfo(graph);

        for(int i = 0; i < n; i++)
        {
            // gen rule
            matchatonce(graph, [*])
            {
                                   a:A;



                       c-->a;                a-->b;



                c:C;              c<--b;              b:B;

                replace {
                                        a;

                               ca-->a;      a-->ab;

                           ca:CA;    ca<--ab;    ab:AB;

                      c-->ca;   ca-->bc;   bc-->ab;   ab-->b;

                    c;     c<--bc;    bc:BC;     bc<--b;     b;
                }
            }
        }

        PrintGraphInfo(graph);

        // WORKAROUND: Mono 1.9 compiler crashes for
        //             "using(VCGDumper ...) graph.Dump(dumper);"
```

```
            VCGDumper dumper = new VCGDumper("testSierpinski.vcg");
            graph.Dump(dumper);
            dumper.Dispose();
        }
    }
}
```

Listing B.2: A G# version of the GrGen.NET “Mutex” example

```
using System;
using de.unika.ipd.grGen.libGr;
using de.unika.ipd.grGen.lgsp;

using de.unika.ipd.grGen.Model_Mutex;

namespace testMutex
{
    public model Mutex
    {
        node class Process;
        node class Resource;

        edge class next
            connect Process [0:1] -> Process [0:1];

        edge class blocked
            connect Resource [*] -> Process [*];

        edge class held_by
            connect Resource [1] -> Process [*];

        edge class token
            connect Resource [1] -> Process [*];

        edge class release
            connect Resource [1] -> Process [*];

        edge class request
            connect Process [*] -> Resource [*];
    }

    public class TestMutex
    {
        public static void PrintGraphInfo(Mutex graph)
        {
            Console.WriteLine(
                  "\nNum nodes: "          + graph.NumNodes
                + "\nNum edges: "          + graph.NumEdges
                + "\nNum Process nodes: "  + graph.GetNumExactNodes(Process.TypeInstance)
                + "\nNum Resource nodes: " + graph.GetNumExactNodes(Resource.TypeInstance)
                + "\nNum next edges: "     + graph.GetNumExactEdges(next.TypeInstance)
                + "\nNum request edges: "  + graph.GetNumExactEdges(request.TypeInstance)
                + "\nNum token edges: "    + graph.GetNumExactEdges(token.TypeInstance)
                + "\nNum release edges: "  + graph.GetNumExactEdges(release.TypeInstance));
        }

        public static void Main(String[] args)
        {
            int n;
            if(args.Length != 1 || !int.TryParse(args[0], out n))
            {
                Console.WriteLine("Usage: testmutex <N>");
                return;
            }

            Mutex graph = new Mutex();

            // initRule
            modify(graph)
            {
                p1:Process -:next-> :Process -:next-> p1;
            }

            // newRule
            match(graph, n-2)
            {
                p1:Process -:next-> p2:Process;
                replace {
                    p1 -:next-> :Process -:next-> p2;
                }
            }

            // mountRule
            match(graph, 1)
            {
                p:Process;
                replace {
                    p <-:token- :Resource;
                }
            }

            // requestRule
            match(graph, *)
            {
                p:Process;
                r:Resource;
                negative {
                    r -hb:held_by-> p;
                }
                negative {
                    p -req:request-> m:Resource;
                }

                replace {
                    p -req:request-> r;
                }
            }

            PrintGraphInfo(graph);

            // WORKAROUND: Mono 1.9 compiler crashes for
            //             "using(VCGDumper ...) graph.Dump(dumper);"
            VCGDumper dumper = new VCGDumper("testMutex.vcg");
            graph.Dump(dumper);
            dumper.Dispose();

            for(int i = 0; i < n; i++)
            {
                // takeRule
                match(graph, 1)
                {
                    r : Resource;
                    r -t:token-> p:Process -req:request-> r;

                    replace {
                        r -hb:held_by-> p;
                    }
                }

                // releaseRule
                match(graph, 1)
                {
                    r : Resource;
                    r -hb:held_by-> p:Process;
                    negative {
                        p -req:request-> m:Resource;
                    }

                    replace {
                        r -rel:release-> p;
                    }
                }

                // giveRule
                match(graph, 1)
                {
                    r : Resource;
                    r -rel:release-> p1:Process -n:next-> p2:Process;

                    replace {
                        p1 -n-> p2 <-t:token- r;
                    }
                }
            }
```

```

            PrintGraphInfo(graph);

            // WORKAROUND: Mono 1.9 compiler crashes for
            //             "using(VCGDumper ...) graph.Dump(dumper);"
            VCGDumper dumper2 = new VCGDumper("testMutex-2.vcg");
            graph.Dump(dumper2);
            dumper2.Dispose();
        }
    }
}
```

Listing B.3: An example just showing several already working features of G#

```
using System;
using de.unika.ipd.grGen.lgsp;
using de.unika.ipd.grGen.libGr;

using de.unika.ipd.grGen.Model_Turing;

namespace grgensharptest
{
    public model Turing
    {
        node class BandPosition { value : int; }
        node class State { name : string = "State"; }
        node class WriteValue { value : int; }

        edge class right
            connect BandPosition [0:1] -> BandPosition [0:1];

        edge class readZero;
        edge class readOne;

        edge class moveLeft;
        edge class moveRight;
    }

    public test<Turing> getValueForReadZero(s:State, bp:BandPosition) : (WriteValue) {
        s -rv:readZero-> wv:WriteValue;
        if{bp.value == 0;}
        return(wv);
    }

    class testClass
    {
        static void PrintElemInfo(Turing graph)
        {
            Console.WriteLine("Num nodes: " + graph.NumNodes);
            Console.WriteLine("Num edges: " + graph.NumEdges);
        }

        static void Main(String[] args)
        {
            Turing graph = new Turing();
            BandPosition bp = graph.CreateNodeBandPosition();
            for(int i = 0; i < args.Length; i++)
            {
                State state = graph.CreateNodeState();
                state.name = "s" + i;
                graph.CreateEdgeEdge(bp, state);
            }

            PrintElemInfo(graph);

            modify(graph)
            {
                bp --> sA : State -:readZero-> wv:WriteValue;
                eval {
                    wv.value = 1;
                }
            }
            Console.WriteLine(sA.name);

            PrintElemInfo(graph);

            WriteValue w = null;
            matchactionatonce(graph, (w)=getValueForReadZero(sA, bp))
            do
            {
                Console.WriteLine(w.value);
            }
            if(w != null)
                Console.WriteLine("w != null");

            PrintElemInfo(graph);

            match(graph, 4)
            {
                bp --> state2 : State;
                do
                {
                    Console.WriteLine(state2.name);
                    modify {
                        delete(state2);
                        x : WriteValue;
                        eval {
                            x.value = 8;
                        }
                    }
                    Console.WriteLine(x.value);
                }
            }
```

```
            else do
            {
                Console.WriteLine("Nothing found...");
            }

            PrintElemInfo(graph);

            match(graph, 1)
            {
                hom(bp, bp2);
                bp2:BandPosition;
                replace {
                    bp2 -:right-> bp;
                }
            }

            PrintElemInfo(graph);
        }
    }
}
```

Bibliography

[Bat06] BATZ, Gernot V.: An Optimization Technique for Subgraph Matching Strategies / Universität Karlsruhe, IPD Goos.
http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf.
Version: April 2006. – Forschungsbericht

[BH05] BOX, Don ; HEJLSBERG, Anders: *The LINQ Project.*
http://msdn.microsoft.com/en-us/library/aa479865.aspx.
Version: September 2005

[BKG08] BATZ, Gernot V. ; KROLL, Moritz ; GEISS, Rubino: A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching.
In: SCHÜRR, A. (Hrsg.) ; NAGL, M. (Hrsg.) ; ZÜNDORF, A. (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)* Bd. NN, Springer, 2008. –
http://www.springerlink.com/content/105633/

[Buc08] BUCHWALD, Sebastian: *Erweiterung von GrGen.NET um DPO-Semantik und ungerichtete Kanten.*
http://www.info.uni-karlsruhe.de/papers/sa_buchwald.pdf.
Version: June 2008. – Studienarbeit

[BV04] BRAVENBOER, Martin ; VISSER, Eelco: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions.
In: *SIGPLAN Not.* 39 (2004), Nr. 10, S. 365–383.
DOI: http://doi.acm.org/10.1145/1035292.1029007. – ISSN 0362–1340

[BV07] BRAVENBOER, Martin ; VISSER, Eelco: Designing Syntax Embeddings and Assimilations for Language Libraries.
In: *Proceedings of the 4th International Workshop on Language Engineering (ATEM 2007)*, 2007

[DE08] DEBREUIL, Robin ; ERCHOFF, Denis: *C# Parser.*
http://www.codeplex.com/csparser.
Version: May 2008

[Den07] Denninger, Oliver: *Erweiterung des Kantenkonzepts deklarativer GES von Einfachkanten über Hyperkanten zu "Superkanten".* Version: March 2007. – Universität Karlsruhe, IPD Tichy, Diplomarbeit

[Ecm06] Ecma: *Standard ECMA-334: C# Language Specification.* http://www.ecma-international.org/publications/standards/Ecma-334.htm. Version: June 2006

[ERT99] Ermel, C. ; Rudolf, M. ; Taentzer, G.: The AGG Approach: Language and Environment. In: *[Roz99]* Bd. 2. 1999, S. 551–603

[GBG$^+$06] Geiss, Rubino ; Batz, Veit ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, A. (Hrsg.) ; Ehrig, H. (Hrsg.) ; Montanari, U. (Hrsg.) ; Ribeiro, L. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Graph Transformations - ICGT 2006*, Springer, 2006 (Lecture Notes in Computer Science), 383 – 397. – Natal, Brasilia

[Gei08] Geiss, R.: *GrGen.* http://www.grgen.net. Version: May 2008

[Jak07] Jakumeit, Edgar: *Vorarbeiten für die Erweiterung des Graphersetzungssystems GrGen um dynamisch zusammengesetzte Muster.* http://www.info.uni-karlsruhe.de/papers/sa_jakumeit.pdf. Version: September 2007. – Studienarbeit

[Jak08] Jakumeit, Edgar: *Mit GrGen.NET zu den Sternen – Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken.* http://www.info.uni-karlsruhe.de/papers/da_jakumeit.pdf Version: July 2008. – Diplomarbeit

[KKBS05] Kurth, Winfried ; Kniemeyer, Ole ; Buck-Sorlin, Gerhard: Relational Growth Grammars - A Graph Rewriting Approach to Dynamical Systems with a Dynamical Structure. In: *Unconventional Programming Paradigms*, Springer, 2005 (Lecture Notes in Computer Science), S. 56 – 72

[Kro07] Kroll, Moritz: *GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen.* http://www.info.uni-karlsruhe.de/papers/sa_kroll.pdf. Version: May 2007. – Studienarbeit

[Leh08] Lehrstuhl Grafische Systeme, BTU Cottbus: *grogra.de - XL*. http://www-gs.informatik.tu-cottbus.de/grogra.de/grammars/xl.html. Version: May 2008

[Lin02] Lindenmaier, Götz: libFIRM – A Library for Compiler Optimization Research Implementing FIRM. http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps. Version: September 2002. – Universität Karlsruhe, IPD Goos, Forschungsbericht Nr. 2002-5.

[Lut08] Lutz Roeder: *Reflector for .NET*. http://www.aisto.com/roeder/dotnet/. Version: June 2008

[Mic08] Microsoft: *GS.GetTotalMemory Method (System)*. http://msdn.microsoft.com/de-de/library/system.gc.gettotalmemory(en-us).aspx. Version: August 2008

[Mon08a] Mono Developer Team: *Mono*. http://www.mono-project.com/CSharp_Compiler. Version: March 2008

[Mon08b] Mono Developer Team: *Mono*. http://www.mono-project.com. Version: March 2008

[Roz99] Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999

[Sau02] Sauer, Hermann: *Relationale Datenbanken*. Pearson Education Deutschland, 2002

[Sch08] Schreiner, Axel: *jay*. http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html. Version: August 2008

[SDF08] SDF Developer Team: *SDF*. http://www.program-transformation.org/Sdf/WebHome. Version: May 2008

[SNZ08] Schürr, A. (Hrsg.) ; Nagl, M. (Hrsg.) ; Zündorf, A. (Hrsg.): *Applications of Graph Transformation with Industrial Relevance, Proceedings of the Thrird International AGTIVE 2007 Symposium, Schlosshotel am Bergpark Wilhelmshöhe,*

Kassel, Germany.
Bd. *5088*. Heidelberg : Springer Verlag, 2008 (Lecture Notes in Computer Science (LNCS))

[Str08] STRATEGO/XT DEVELOPER TEAM: *Stratego/XT*.
http://www.program-transformation.org/Stratego/WebHome.
Version: May 2008

[Zü08] ZÜNDORF, Albert: *AntWorld*.
http://www.se.eecs.uni-kassel.de/~fujabawiki/index.php/AntWorld.
Version: May 2008. – GraBats 2008 workshop