Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

GrGen.NET

The Expressive, Convenient and Fast Graph Rewrite System

Edgar Jakumeit • Sebastian Buchwald • Moritz Kroll

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
`www.grgen.net`

The date of receipt and acceptance will be inserted by the editor

Abstract. GrGen.NET is a generative programming system for graph rewriting, transforming intuitive and expressive rewrite rule specifications into highly efficient .NET code. The user is supported by a convenient environment consisting of a graph viewer, an interactive shell with integrated debugging support, and an elegant domain specific language for the combination of rewrite rules. After rapid prototyping with these tools, the resulting graph transformation programs can be easily integrated into arbitrary .NET applications to serve as the algorithmic kernel. Expressiveness, convenience, and speed are exemplified by GrGen-solutions to the case studies AntWorld, Refactoring, and ConferenceScheduling – besides others.

Key words: General purpose graph transformation • Graph rewriting • Domain specific language • Generative programming tool • Search plan driven graph pattern matching

---

1 Introduction

In this chapter we will give an introduction into GrGen.NET [BG08, KG07, GBG+06] and its specification languages which will serve as a basis to the following chapters in which we will present our solutions to the GraBaTs 08 [RVG08] challenges Conference Scheduling, AntWorld, and Program Graphs. Afterwards, we will discuss our experience with GraBaTs, firstly regarding new features learned, secondly casting a recurring discussion into paper form, before we finally conclude.

The application domain neutral graph rewrite system GrGen.NET licensed under LGPL3 [LGP07] is combined from two groups of components, the first being the graph rewrite generator GrGen itself together with the execution environment libGr offering the basic functionality of the system, and the second being the interactive command line GrShell plus the graph viewer yComp, offering a rapid prototyping environment.

The Base System

The graph rewrite generator reads in files containing specifications of the *graph model* and of the *graph rewrite rules* to be used and emits highly efficient C#-programs / .NET-assemblies implementing these specifications, assisted by the runtime library, which in addition offers a *control language for rule application.*

The Graph Model Language

The graph model language allows the specification of typed and attributed multigraphs with multiple inheritance on node and edge types, offering directed as well as undirected edges. Types bring safety and performance with them, while multiple inheritance with the root types `Node` and `Edge` allows for concise matching patterns. Graph elements are attributed depending on their type with values of the basic attribute types `boolean`, `int`, `float`, `double`, `string`, `object`, or user-defined `enums`, plus `set<T>` and `map<S,T>` with `S` and `T` being basic attribute types (the latter have been added to better support users from the area of computer linguistics needing string dictionaries). In addition, the graph may be validated against connection assertions constraining the degrees and types of edges between nodes of certain types.

The Pattern Language

The graph rewrite language offers `tests` consisting only of a pattern to be matched, and `rules` consisting of a pattern part to be matched and a rewrite part to be applied.

The pattern part is built from node and edge declarations or references with an intuitive and easy to learn

syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type, a dot `.` is used for introducing an anonymous node of type `Node`. An edge `e` with source `x` and target `y` is declared by `x -e:t-> y`, whereas `-->` introduces an anonymous edge of type `Edge`. Nodes and edges are referenced outside their declaration by `n` and `-e->`, respectively. Undirected edges are declared by `x -e:t- y` or `--` and are referenced by `-e-`. Furthermore, there are

– Negative Application Conditions (NACs [EHK+97]), which are pattern graphs which must not be present in the host graph for the rule to be applicable; they are specified in `negative` blocks and may be arbitrarily nested.
– Type and attribute conditions, given within an `if`-clause, constraining the allowed types and attribute values of matched elements by numerous type and value expressions consisting of boolean, arithmetic, string, set, and map operations.
– Homomorphic matching in addition to the default isomorphic matching, allowing distinct pattern elements given within the `hom()`-statement to be bound to the same host graph element.
– Parameters allowing pattern elements to be preset from outside the pattern before the rest of the pattern is searched.
– Several more to be introduced later on.

The Rewrite Language

The relationship between the pattern and the rewrite part is given by a preservation morphism according to the theoretically well-founded SPO [EHK+97] approach, while the DPO [CMR+97] approach is supported as well (by prepending `dpo` before `rule` in the specification). Syntactically, it is specified by a `modify`- or `replace`-block nested within the pattern. With `replace`-mode, graph elements which are referenced within the replace-block are kept, graph elements declared in the replace-block are created, and graph elements declared in the pattern, not referenced in the replace-part are deleted. With `modify`-mode, all graph elements are kept, unless they are specified to be deleted within a `delete()`-statement. Additionally allowed rewrite constructs are

– Attribute recalculations: results of expressions can be assigned to the attributes of matched or new elements within an `eval`-statement.
– Retyping: matched elements can be retyped not necessarily respecting the type hierarchy – keeping common attributes of initial and final type. This comes in handy if incident edges are to be kept.
– Creation with dynamic types: new graph elements may be created using the actual type of some matched element (more exact than the specified static supertype).
– Element returns: non-deleted elements can be returned to users outside the rule.

The Rule Application Control Language

Graph rewrite sequences are a domain specific language used to combine single rule/test applications into graph rewriting (sub-)programs. They resemble a union of logical and regular expressions.

– A rule application $(y_1, \ldots y_k) = r(x_1, \ldots, x_m)$ attempts to extend the matches of its parameters x_1, $\ldots, x_m$ to an arbitrary match of its pattern so that the rule can be applied. If this is possible, the application *succeeds*, and defines the variables $y_1, \ldots y_k$; otherwise it *fails*. A test is handled the same way, but does not modify the graph.
– For rewrite sequences S_1, S_2, the logical operations conjunction S_1 && S_2 and disjunction $S_1 \,||\, S_2$ are evaluated lazily from left to right: S_2 is not evaluated if the success or failure of S_1 does already determine the result of the operation. Their strict counterparts & and | always evaluating the second operand exist as well, supplemented by the negation ! .
– Iteration is supported by the construct $S[n : m]$ with the rewrite sequence S and the non-negative integers n and m, which executes S until it fails, but at most m-times. $S[n : m]$ fails, if S was not executed at least n-times. Typical iterations can be abbreviated with S^* $(n = 0,\, m = \infty)$ and S^+ $(n = 1,\, m = \infty)$.
– Furthermore, variables may be declared, in order to store results of rule applications and pass them as arguments to following rule applications.

Graph models, graph rewrite rules and graph rewrite sequences are accessible from any .NET-language via an API (offered by libGr). An application built on top of this API is the GrShell shipped with GrGen.NET.

The Rapid Prototyping Environment

On top of the graph rewrite functionality, the GrGen.NET system offers tools for rapid development and graphical debugging. These are the GrShell, offering interactive or file-based execution of a graph rewrite script, including step-by-step execution of graph rewrite sequences, aided by the yFiles[WEK02]-based graph viewer yComp, which can visualise the current graph and each rule application, thus finally the entire rewrite process.

Now that we know the basics of the GrGen.NET system, we will have a deeper look at the GrGen solutions to the online and offline challenges posed at the GraBaTs 2008 tool contest.

2 Conference Scheduling

The purpose of the Conference Scheduling case study [?] – mapping paper presentations to time slots – was to evaluate the tools of the participating groups under

time pressure in form of a live contest. So, rapid prototyping was the key to win the contest. With GrGen.NET we not only managed to be the first to solve the basic assignment, but were also the only ones with a correct solution.

The important features of GrGen.NET which enabled us to solve the problem so quickly are: intuitive, well-designed languages, the capability to find all matches for a pattern, and the ability to execute a graph rewrite sequence for each found match. So in a first step we could look for a rough mapping of papers to consecutive time-slots (13,824 possibilities), while the additional constraints were checked in a second step leading to the 3,008 valid solutions, which were written to a file. Just mapping the papers to any time-slots (as done by some other groups) would have resulted in 10,077,696 possibilities to be thoroughly checked, so the rough mapping already saves much time.

The additional constraints of the basic assignment were easy to express:

```
test checkConstr
{
    // No simultaneous presentations by same person
    negative {
        ps:Person -:presents-> :Paper -:assocTimepos-> t:Slot;
        ps -:presents-> :Paper -:assocTimepos-> t;
    }

    // No presenter is chairing another session simultaneously
    negative {
        ps:Person -:presents-> p1:Paper -:assocTimepos-> t:Slot;
        p1 -:presentedIn-> session:Session;
        ps -:chair-> othersession:Session <-:presentedIn-
            p2:Paper -:assocTimepos-> t;
    }
}
```

The above listing declares `checkConstr` as a test, which consists of two negative application conditions (NACs) for the constraints. The test fails if one of the NACs matches. The first NAC matches, if one person `ps` presents two different papers during the same time-slot `t`; the second NAC matches, if one person `ps` presents a paper in one session during a time-slot `t` and chairs another session at the same time.

3 AntWorld

The AntWorld case study [?] is designed to compare the performance of various graph transformation tools. It consists of simple rules, which simulate the life of an ant colony. To achieve good performance many solutions submitted to GraBaTs'08 use a more complex approach than described by the case study itself. As a result, more than half of the submitted solutions contained bugs. This shows, that the real challenge of the AntWorld case study is to develop a fast *and correct* solution, i.e. to find a good balance between performance and simplicity.

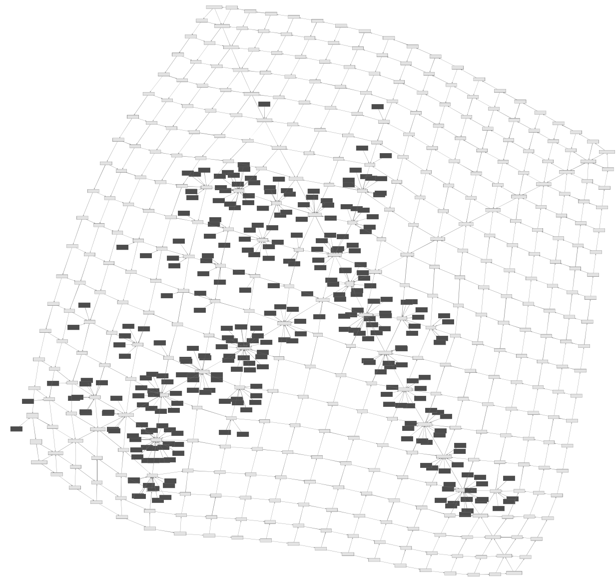

Fig. 1. An AntWorld before grid extension after 61 rounds

GrGen.NET faces this challenge by providing a clean and expressive rule specification language, as well as fully automatic optimization of transformation rules. It allows to write the required transformation rules straight from the specification, e.g. the excerpt "If the ant is in search mode and no outer neighbour has sufficient pheromones, the ant moves to any of its neighbour fields based on a fair random choice. (However, an ant without food shall not enter the ant hill.)" leads to the following transformation rule for such an ant:

```
rule SearchAimless(curAnt:Ant)
{
    // get position of current ant
    curAnt -oldPos:AntPosition-> old:GridNode;

    // get neighbour field
    old <-:GridEdge-> new:GridNode\AntHill;

    modify {
        // move ant to neighbour field
        delete(oldPos);
        curAnt -:AntPosition-> new;
    }
}
```

The tricky part of the rule is to find all possible neighbours, regardless of whether the neighbour is connected by an incoming or outgoing edge. For this purpose GrGen.NET offers arbitrary directed edges "<-->" matching a directed edge either way, which is one of many features that keep GrGen.NET rules simple and intuitive. As already mentioned above, this simplicity allows a faster and less error-prone development of solutions.

In spite the burden of offering a convenient language, the AntWorld case study reaffirms that GrGen.NET is among the fastest graph transformation tools (cf. [GK07, TBB$^+$08]). This is due to the generative approach used by GrGen.NET (something we have in common with

e.g. the tools PROGRES [SWZ99], FUJABA [NNZ00], GReAT [KASS03] or recently VMTS [LLMC05], contrasted by interpreted or constraint-solving based tools like VIATRA2 [VB07] or AGG [Tae04]), and the rule-specific search plans [Bat06, VVF06], which determine the order of matching graph elements, allowing a fast pruning of the search space taking dynamic host graph statistics into account. GrGen.NET computes the search plans fully automatically (and the computed search plans perform well, cf. [BKG08]), in contrast to e.g. the semi-automatic visual programming tool FUJABA [NNZ00], which for example requires the user to choose the starting point of the search, where a point not chosen wisely may cause a performance penalty of a complexity class; this may happen easily for users not knowing the implementation, as witnessed with the Varró-Benchmark [GK07, VSV05].

Using GrShell and yComp, it was very easy to play around with the rules and see what they actually do (how and when they get applied, what they match and what they modify), or, at the abstraction level of the challenge, see the ants crawling (figure 1 shows an example state reached while simulating the AntWorld).

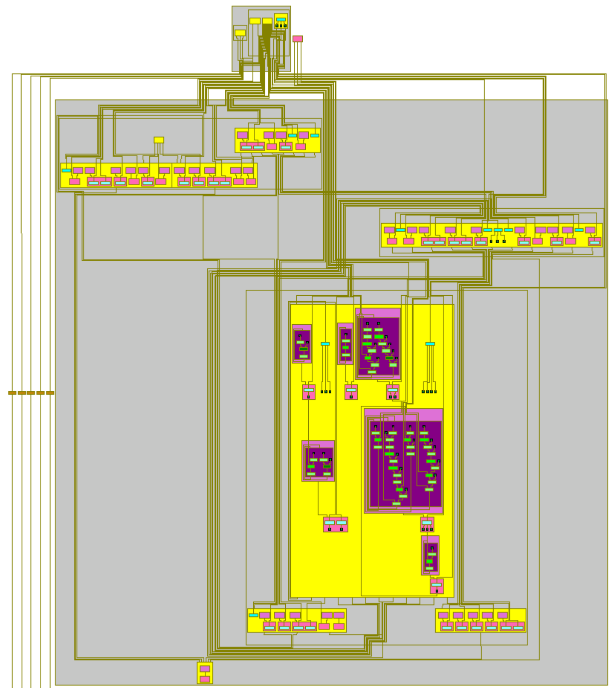

Fig. 2. Overview of the initial program graph

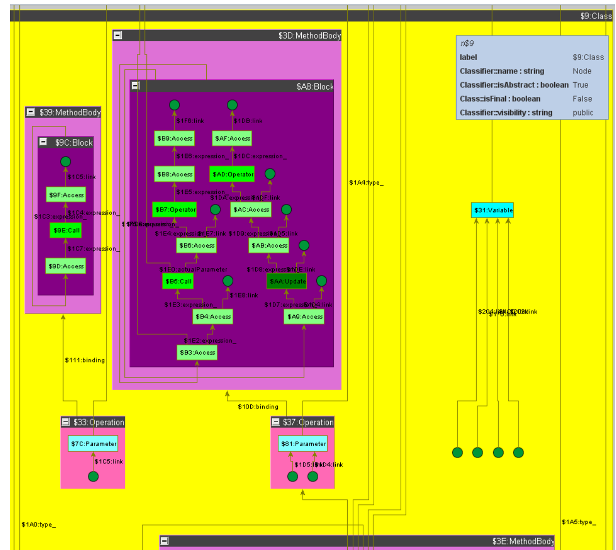

Fig. 3. Some details of the "Node" class

4 Program Graphs

The program graphs challenge [?] is designed to compare graph rewrite systems regarding their expressiveness, and in addition their ability to interact with the user. It requires the capability to import and export GXL [WKR02, HSESW05]. The task is to implement the refactorings encapsulate field and move method on a given graph representation of programs (especially a given host graph). In the following we will introduce several features of GrGen with relevance to this case.

Graph Relabeling

GrGen.NET is not only a graph rewriting system, it is also a graph relabeling system [LMS99]. With the syntax `y:t<x>` you may specify a graph element `y` to be a retyped (to the new type `t`) version of an original graph element `x` of a different - not necessarily - super type. This allows to keep the incident edges of a node untouched, while retyping it to its new type. This is a feature which was included in GrGen because of its usefulness in optimizations on graph-based compiler intermediate representations of programs in SSA form, which was the original task of the GrGen graph rewrite system. It is used in the encapsulate field refactoring of the program graphs challenge to retype the `Access` or `Update` nodes into `call` nodes, keeping all the users of the original node, thus saving us from iterating through this unknown set in order to replicate the incident edges which would be required, if we had to delete the original node and create a new `call` node.

Nested Graph Layout

Graphs allow fast visual understanding – but only as long as the amount of nodes and edges is kept low; the given sample graph (depicted in figures 2 and 3) with its more than 500 graph elements is beyond comprehension if presented in a flat layout. Luckily, a lot of graphs are hierarchically structured, with the program graphs or compiler intermediate graphs falling into this category. For them our graph viewer offers the layout Hierarchic, extending nodes to large areas containing entire subgraphs, and the layout Compilergraph, developed out

of the needs of day-to-day work with a compiler intermediate representation as an extension to Hierarchic, cutting certain edges, marking the start and endpoint of the cut with fat dots, allowing to jump to the other point by clicking on either one. Together with color coding (grey for packages, yellow for classes, magenta for methods, cyan for variables and green for expressions), this allows for an understandable, well-structured visualisation of the host graph, in contrast to the unusable haystack of a flat layout.

The layout as well as the colors are configured by commands in the graph rewrite script of our solution; further layouts are available, the same known from yFiles which yComp uses.

Emit and Exec

GXL file import is achieved by a specialized program `gxl2grs`, transforming the given (invalid) .gxl into an equivalent .grs. Exporting is done by normal GrGen rules, as given in the file `JavaProgramGraphsGxlDumper.gri` (within the `examples/JavaProgramGraphs`-directory of the GrGen.NET package).

Features used in the export process are:

- the `emit` statement, which allows to emit a sequence of string expressions to stdout or into an arbitrary file as specified by the user,
- the `exec` statement, which allows to execute a graph rewrite sequence directly after the basic rule was applied, with the graph elements of the pattern and rewrite part being available as arguments to the tests and rules in the rewrite sequence,
- the visited flags, which allow to mark elements, in order to highlight them for further processing or to prevent them from being processed twice during a graph walk; they could be introduced manually in the graph model, but this specialized treatment is more convenient to the user and allows for a more efficient implementation (as they are saved in some excess bits of the graph elements).

One of the rules used iteratively for exporting the graph is shown below, matching a package node not yet visited, marking it as visited and emitting the node with its internal name plus the name attribute.

```
rule dumpNodePackage
{
    n:Package;

    if { !visited(n); }

    modify {
        eval { visited(n) = true; }

        emit("<node id=\"", nameof(n), "\">",
            "<type xlink:href=\"Package\"/>",
            "<attr name=\"name\"> <string>",
                n.name, "</string> </attr>",
            "</node>");
    }
}
```

The same mechanism is used in the MOF-Suite for GrGen.NET, extending GrGen for model transformation [GDG08], capable of importing and exporting UML models conforming to MOF 2.0 given in XMI.

Recursive Patterns

The java program graphs defined in the challenge are to a good degree tree like, resembling an abstract syntax tree with additional intermediate edges from uses to definitions. Tree-like or other structures of recursive nature can get matched or even rewritten neatly by one application of a single rule. This is allowed by the concept of *subpatterns* resp. *subrules* (which in our case are subpatterns with nested `modify/replace`-parts), combined with *alternative parts*, as introduced in [HJG08] (based on Hyperedge Graph Replacement [DKH97] on the meta level to assemble rules to be applied on the object level). Similar approaches – limited to recursive subpatterns – can be found in the tools VIATRA2 [VHV08] and TEFKAT [LS05]. In [HJG08] sub*rules* were used to model the transcription of a DNA strand into an RNA strand, here sub*patterns* are used to model an iterated path condition in the implementation of the “Move Method Refactoring”, checking whether the name of the operation already exists in the scope of the target class. This check is accomplished by the following subpattern `methodNameExistsSuper`, which succeeds if in the given class `cls` there is an operation `op`, which either bears the name we are searching for (alternative case “here”), or there is a superclass `super`, in which the check succeeds (alternative case “super”).

```
pattern methodNameExistsSuper(opForName:Operation, cls:Class)
{
    cls <-:belongsTo- op:Operation;

    alternative {
        here {
            if { op.name==opForName.name; }
        }
        super {
            cls -:extends_-> super:Class;
            :methodNameExistsSuper(opForName, super);
        }
    }
}
```

(Furthermore, the subpatterns are used for dumping the graph to a text file for debugging, obeying the nesting structure; there the `emitpre` and `emitpost` statements are used, which allow to emit text before and after emitting the text for the nested subpatterns.)

5 Features learned

The GraBaTs – Graph-Based Tools Contest – is designed to stimulate learning own limitations and learning from

other tools, as well as serving as a discussion platform for tool creators and potential users. After a short explanation of the development goals of GrGen.NET we will see how feedback from the tool contest helped improving the GrGen tool regarding each one of them. In the following section Discussion 6 we want to cast a recurring discussion of GraBaTs about textual and graphical representations into paper form.

Development goals

Based on our own experience of day-to-day work with rewriting of a graph based compiler intermediate language – a real world task tackled with graph rewriting – we focused on

- expressive language constructs allowing for a concise and descriptive solution
- high execution speed at modest memory consumption, so that you can really use the result of your work
- convenient development, realized by step wise and graphical debugging, together with a well designed API and an extensive user manual.

The tool contest gave valuable input on each of these fields, based on it we are currently designing and implementing the following extensions to the GrGen.NET system. Most of them will be available in the next major release.

Convenience: GXL

The current solution with the gxl2grs converter and the user-specified export is working, but lacks convenience. Instead we will add an importer and an exporter for GXL 1.0 directly to the libGr, available from the API and the GrShell. This will ease tool comparisons and migration.

Execution speed: Storages

Storages – sets of nodes– gave the tool VMTS the thrust to win the performance case AntWorld (cf. [?]).
In GrGen.NET they will be represented as variables of type `set<Node>`, accessible for graph rewrite sequences like `s.add(el)`, adding an element to a set, `s.rem(el)` for removing an element from a set, and the iteration `for{x in s; r(x)}` executing the nested sequence for all elements in the set, with a new element bound to the iteration variable in each step. Looking up elements in the storages instead of searching for marked elements in the graph will allow for faster solutions. (But sets of nodes will not be made available as attribute types in the graph model because we regard this usage to be too error prone).

Expressiveness: Pattern Cardinality and PAC

The refactoring case indeed proved to be a stress test regarding expressiveness. While the recursive patterns / rules were a major leap ahead, they weren't as useful in solving this challenge as expected. While they do well in matching structures of arbitrary depth, as e.g. iterated paths, or of arbitrary breadth, e.g. a multinode, they have their weaknesses in matching recursive structures of arbitrary depth *and* arbitrary breadth, like an abstract syntax tree. To overcome this limitation they will be supplemented by an `iterated`-construct, matching its nested pattern as often as possible. With this feature, it will be possible to match a spanning tree as simple as this:

```
pattern SpanningTree(root:Node)
{
    iterated {
        root -- next:Node;
        :SpanningTree(next);
    }
}
```

A further problem we experienced in this case was the incapability to check an iterated path condition targeting an unknown but already matched node (matched in a subpattern containing the current pattern). The check always fails because the node is locked due to the isomorphy constraint. This problem will be solved by Positive Application Condition patterns – syntactically specified by an `independent`-block, requiring the nested pattern to get matched, but not constraining the contained elements to get matched isomorphically to the elements of the nesting pattern. So in the following test, there will be always a match of the iterated path heading to a node of type `A` in case the test pattern was found – the elements of the test pattern once again.

```
test someTest
{
    na:A <-- . <-- n:Node;
    independent {
        :IteratedPathToA(n);
    }
}
```

6 Discussion

As elaborated at the beginning of the previous section, our major development goals were:

- language expressiveness
- execution speed
- development convenience

We did *not* invest into the development of graphical languages and graphical editors. Regarding several other tools offering graphical languages and graphical editors (FUJABA [FNTZ00], VMTS [LLMC05], AGG [Tae04])

and the discussions on the “graphical vs. textual” issue during the tool contests, we think this is a good place to cast our arguments (largely in favour of textual languages) into paper form.

Benefits

The superiority of graphical representations and editors or textual representations and editors depends on the domain and is to a certain degree a matter of taste. For rule application control and attribute computations we doubt graphical representations and editors to be superior at all, a claim we think is backed by the unsuccessfulness of graphical programming languages compared to textual ones. For graph pattern matching and rewriting on the other hand a graphical representation would be a more intuitive and direct user interface – at least as long as the graphs stay small, they quickly loose the property of fast visual understandability with an increase in the number of elements. But the less direct textual representations offer some other benefits, and most important in our case, are less expensive.

Costs

Given limited resources, you must choose into which areas to invest. While graphical representations are the more intuitive ones for the user in the case of graph patterns, they come at a higher cost. As they are not offering a higher expressiveness, only a more convenient user interface, we decided to follow a first things first policy and focused on the aforementioned development goals. In our opinion, a textual language which offers the user the expressiveness and performance he needs to tackle a given problem is more important than a graphical language of less expressiveness backed by a slower engine developed given the same resources.

Tool support

The lower costs are to a certain degree based on tool support, with compiler construction tools like parser generators – which gave us for instance the ability to add the $-operator for indeterministic choice needed for the AntWorld-case within the shortest amount of time – being more mature than graphical representation toolkits. But while this gap will narrow in the future, we doubt it will vanish, since graphical representations are the more complex technology compared with textual representations.

Robustness

Technological complexity has another recurrence: human optimized textual representations allow the user to read, edit, and write the entities of design with the simplest of editors, in contrast to graphical representations which require the original viewer/editor to be available and running without major bugs. We think that the more complex technology should be built on top of the simple one: a graphical rule editor reading and writing the textual representation can be implemented later on as an add-on, even without contact to the original developers, so the development process itself gets more robust. (Software developers willing to implement a graphical rule editor on top of GrGen.NET are welcome.)

Harnessing generative power

Finally, domain specific languages mapping to the graph rewrite language or simple helper programs generating rules programmatically (exploiting similarities) can be implemented and debugged easier. Generation is just a matter of filling some schemes with variables and invoking a text emit. Further on, a compiler operating on a human-optimized format, itself meant to be used by humans, normally gives useful error messages, which, together with the fact that all information is immediately available in the file processed, allows locating the error quickly. (And our experience of implementing the GrGen.NET code generator as well as an optimizer generator for a compiler intermediate language emitting GrGen rules taught us that this is a point not to be neglected.) A graphical representation on the other hand must be accessed via the API of the editor or via the machine optimized import/export format, which will be cumbersome or impossible to read directly; and the tool normally will give you only the information that import failed, as it is only meant to be used for reading/writing its own rules, with errors handled by the developers in their debugger.

7 Conclusion

GrGen.NET is a special purpose software development system and a general purpose graph rewrite system. It is designed for software developers and meant to be used for the generation of the algorithmic core of applications processing graph structured data. Its user interface and implementation were chosen to perform reasonably well on any graph rewriting task – and we think it does so, looking at uses in e.g. compiler construction [SG07], model transformation [GDG08], computer linguistics [BG09] or computational biology [SGS09].

Regarding the GraBaTs challenges, GrGen.NET has shown its usability and superiority for rapid prototyping in the live contest Conference Scheduling case (as well as in the GraBaTs 09 live contest Conveyor Belt case [VGLR09]). Normally fast speed of development leads to low execution speed of the resulting code, which is not the case with GrGen.NET, as witnessed in the AntWorld challenge or proven by the Varró benchmark [VSV05]. This is due to the generative approach and

the host graph sensitive search plan based graph pattern matching, which allows the user to write down directly what he thinks, still getting a high-performance solution. With the `iterated` language construct introduced due to feedback from the Refactoring case GrGen.NET has taken a leading position in rewrite rule expressiveness, now being capable of handling recursive structures of arbitrary depth and breadth, which allows for further shifting of work from the imperative control programs to the declarative rewrite rules.

GrGen.NET (available at `www.grgen.net`) offers a convenient development environment consisting of a sophisticated graph viewer and an interactive shell supporting graphical and step-wise debugging; it provides intuitive and very expressive specification languages and a highly optimized code generator, yielding the highest combined speed of development and execution.

We thank our three reviewers, the two anonymous ones as well as our third one, Rubino Geiß.

References

Bat06. Batz, Gernot V.: *An Optimization Technique for Subgraph Matching Strategies.* Internal Report, 2006. – "`http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf`"

BG08. Blomer, Jakob ; Geiss, Rubino: *The GrGen.NET User Manual.* Internal Report. `http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf`. Version: 2008

BG09. Bédaride, Paul ; Gardent, Claire: Semantic Normalisation: a Framework and an Experiment. In: *Eighth International Conference on Computational Semantics*, 2009

BKG08. Batz, Gernot V. ; Kroll, Moritz ; Geiss, Rubino: *A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching.* Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Proceedings, 2008. – "`http://www.info.uni-karlsruhe.de/papers/agtive_2007_search_plan.pdf`"

CMR+97. Corradini ; Montanari ; Rossi ; Ehrig ; Heckel ; Löwe: *Algebraic Approaches to Graph Transformation - Part I: Basis Concepts and Double Pushout Approach.* Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, 1997

DKH97. Drewes ; Kreowski ; Habel: *Hyperedge Replacement Graph Grammars.* Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, 1997

EHK+97. Ehrig ; Heckel ; Korff ; Löwe ; Ribeiro ; Wagner: *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach.* Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, 1997

FNTZ00. Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars ; Zündorf, Albert: Story Diagrams: A New Graph Grammar Language based on the Unified Modelling Language and Java. In: *Theory and Application of Graph Transformations*, 2000, S. 157–167

GBG+06. Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam M.: *GrGen: A Fast SPO-Based Graph Rewriting Tool.* Proceedings 3rd Int. Conf. on Graph Transformation (ICGT '06), 2006. – "`http://www.info.uni-karlsruhe.de/papers/grgen_icgt2006.pdf`"

GDG08. Gelhausen, Tom ; Derre, Bugra ; Geiss, Rubino: Customizing GrGen.NET for Model Transformation. In: *GraMoT*, 2008, S. 17–24

GK07. Geiss, Rubino ; Kroll, Moritz: *On Improvements of the Varró Benchmark for Graph Transformation Tools.* Internal Report. `http://www.info.uni-karlsruhe.de/papers/TR_2007_7.pdf`. Version: 2007

HJG08. Hoffmann, Berthold ; Jakumeit, Edgar ; Geiss, Rubino: *Graph Rewrite Rules with Structural Recursion.* 2nd Intl. Workshop on Graph Computational Models (GCM 2008), 2008. – "`http://www.info.uni-karlsruhe.de/papers/GCM2008.pdf`"

HSESW05. Holt, Richard C. ; Schürr, Andy ; Elliott Sim, Susan ; Winter, Andreas: GXL: A graph-based standard exchange format for reengineering. In: *Science of Computer Programming* (2005)

KASS03. Karsai, Gabor ; Agrawal, Aditya ; Shi, Feng ; Sprinkle, Jonathan: On the use of graph transformation in the formal specification of model interpreters. In: *Journal of Universal Computer Science* 9 (2003), S. 1296–1321

KG07. Kroll, Moritz ; Geiss, Rubino: *Developing Graph Transformations with GrGen.NET.* Internal Report. `http://www.info.uni-karlsruhe.de/papers/agtive_2007_grgennet.pdf`. Version: 2007

LGP07. *GNU Lesser General Public License.* `http://www.gnu.org/licenses/lgpl-3.0.txt`. Version: 2007

LLMC05. Levendovszky, Tihamér ; Lengyel, László ; Mezei, Gergely ; Charaf, Hassan: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In: *Electronic Notes in Theoretical Computer Science*, 2005, S. 65–75

LMS99. Litovsky ; Métivier ; Sopena: *Graph Relabelling Systems and Distributed Algorithms.* Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3, 1999

LS05. Lawley, Michael ; Steel, Jim: Practical Declarative Model Transformation with Tefkat. In: *MoDELS Satellite Events*, 2005, S. 139–150

NNZ00. Nickel, Ulrich ; Niere, Jörg ; Zündorf, Albert: The FUJABA environment. In: *ICSE '00 Proceedings of the 22nd international conference on Software engineering*, 2000, S. 742–745

RVG08. Rensink, Arend ; Van Gorp, Pieter: *Graph-Based Tools: The Contest.* Proceedings 4th Int. Conf. on Graph Transformation (ICGT '08), 2008

SG07. Schösser, Andreas ; Geiss, Rubino: *Graph Rewriting for Hardware Dependent Program Optimizations.* Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Proceedings. `http://www.info.uni-karlsruhe.de/papers/agtive_2007_firm.pdf`. Version: 2007

SGS09. Schimmel, Jochen ; Gelhausen, Tom ; Schaefer, Christoph A.: Gene Expression with General Purpose Graph Rewriting Systems. In: *Proceedings of the 8th GT-VMT Workshop*, 2009

SWZ99. Schürr, A. ; Winter, A. J. ; Zündorf, A.: The PROGRES approach: language and environment. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2*, 1999, S. 487–550

Tae04. Taentzer, Gabriele: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: *Applications of Graph Transformations with Industrial Relevance*, 2004, S. 446–453

TBB+08. Taentzer, Gabriele ; Biermann, Enrico ; Bisztray, Dénes ; Bohnet, Bernd ; Boneva, Iovka ; Boronat, Artur ; Geiger, Leif ; Geiss, Rubino ; Horvath, Ákos ; Kniemeyer, Ole ; Mens, Tom ; Ness, Benjamin ; Plump, Detlef ; Vajk, Tamás: Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In: *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, 2008, S. 514–539

VB07. Varró, Dániel ; Balogh, András: The Model Transformation Language of the VIATRA2 Framework. In: *Science of Computer Programming* 68 (2007), Nr. 3, S. 214–234

VGLR09. Van Gorp, Pieter ; Levendovszky, Tihamér ; Rensink, Arend: *Live Challenge Problem.* `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/live_problem.pdf`. Version: 2009

VHV08. Varró, Gergely ; Horváth, Ákos ; Varró, Dániel: Recursive Graph Pattern Matching. In: *Applications of Graph Transformations with Industrial Relevance.* 2008, S. 456–470

VSV05. Varró, Gergely ; Schürr, Andy ; Varró, Daniel: Benchmarking for Graph Transformation. In: *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, S. 79–88

VVF06. Varró, Gergely ; Varró, Dániel ; Friedl, Katalin: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In: *GraMot 2005, International Workshop on Graph and Model Transformations*, 2006, S. 191–205

WEK02. Wiese ; Eiglsperger ; Kaufmann: *yFiles: Visualization and Automatic Layout of Graphs.* Graph Drawing, 9th International Symposium, 2002

WKR02. Winter ; Kullbach ; Riediger: *An Overview of the GXL Graph Exchange Language.* Software Visualization - International Seminar Dagstuhl Castle, 2002