A GrGen.NET solution of the Hello World Case for the Transformation Tool Contest 2011

Sebastian Buchwald Edgar Jakumeit

May 16, 2011

1 Introduction

We introduce the graph transformation tool GrGen.NET (www.grgen.net) by solving the Hello World Case [1] of the Transformation Tool Contest 2011 which consists of a collection of small transformation tasks; for each task a section is given explaining our implementation.

2 What is GrGen.NET?

GrGen.NET is an application domain neutral graph rewrite system developed at the IPD Goos of Universität Karlsruhe (TH), Germany [2]. The feature highlights of GrGen.NET regarding practical relevance are:

Fully Featured Meta Model: GrGen.NET uses attributed and typed multigraphs with multiple inheritance on node/edge types. Attributes may be typed with one of several basic types, user defined enums, or generic set, map, and array types.

Expressive Rules, Fast Execution: The expressive and easy to learn rule specification language allows for a straightforward formulation of even complex problems, with an optimized implementation yielding high execution speed at modest memory consumption; outstanding features are iterated and recursive pattern matching incl. rewriting.

Programmed Rule Application: GrGen.NET supports a high-level rule application control language, Graph Rewrite Sequences (GRS), offering logical, sequential and iterative control plus variables and storages for the communication of processing locations between rules.

Graphical Debugging: GrShell, GrGen.NET's command line shell, offers interactive execution of rules, visualising together with yComp the current graph and the rewrite process. This way you can see what the graph looks like at a given step of a complex transformation and develop the next step accordingly. Or you can debug your rules.

3 Hello World!

The first task is to create a `Greeting` node with appropriate text. To solve the task we use the GrGen rule shown in Figure 1 that creates the required graph when being executed.

```
rule createHelloWorld {
    replace {
        greeting:helloworld_Greeting;

        eval {
            greeting._text = "Hello World";
        }
    }
}
```

Figure 1: HelloWorld.grg

Rules in GrGen consist of a pattern part specifying the graph pattern to match and a nested rewrite part specifying the changes to be made. The pattern part is built up of node and edge declarations or references with an intuitive syntax: Nodes are declared by `n:t`, where `n` is an optional node identifier, and `t` its type. 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. Attribute conditions can be given within `if`-clauses.

The rewrite part is specified by a `replace` or `modify` block nested within the rule. 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. Attribute recalculations can be given within an `eval`-statement. These and the language elements we introduce later on are described in more detail in the extensive GrGen.NET user manual [2].

The rule shown in Figure 1 consists of an empty pattern part and a replace part that creates a new node `greeting` of type `helloworld_Greeting` and `eval`uates the corresponding `_text` attribute.

The creation rule for the extended metamodel given in Figure 2 is similar in structure, just more voluminous. A `Greeting` node is created and linked with a `greetingMessage` edge to a `GreetingMessage` node. Furthermore it is linked with a `person` edge to a `Person` node. Then the attributes of the `message` and the `person` are initialized to the requested values (and the containment indices for XMI are set).

```
rule createHelloWorldExt {
    replace {
        greeting:Greeting;
        greeting -e:greetingMessage-> message:GreetingMessage;
        greeting -f:person-> person:Person;

        eval {
            message.text = "Hello";
            person.name = "TTC␣Participants";

            e.index = 0;
            f.index = 0;
        }
    }
}
```

Figure 2: HelloWorldExt.grg

The files containing the rules seen so far start with a `using` statement

```
using helloworld_ecore;
```

which imports the GrGen metamodel which was generated from the given Ecore metamodel. GrGen.NET does not support importing Ecore metamodels directly (in contrast to GXL and native GRS files). Instead we supply an import filter generating an equivalent GrGen-specific graph model (.gm file) from a given Ecore file by mapping classes to GrGen node classes, their attributes to corresponding GrGen attributes, and their references to GrGen edge classes. Inheritance is transferred one-to-one, and enumerations are mapped to GrGen enums. Class names are prefixed by the names of the packages they are contained in to prevent name clashes; the same holds for references which are prefixed by their node class name, and node/edge attributes which are prefixed by an underscore. This name mangling can be seen in Figure 1, in the following listings it was removed due to space constraints and for the sake of readability, up to the migration chapter, from there on it is needed again to prevent type ambiguity. Normally you want to import an instance graph XMI adhering to the Ecore model thus adhering to the just generated equivalent GrGen graph model; if given, that one is imported by the filter, too, serving as the host graph for the following transformations (this can be seen in Figure 10, in this section we are only interested in the model). The import process described is brought to life by the `import` command of the GrShell.

The GrShell offers the host environment for applying the rules given in the `.grg` files. The command line shell may be operated in interactive mode or in batch mode, for the tasks of this case we use it in batch mode by executing graph rewrite scripts, i.e. `.grs` files. The high level workflow of using GrGen is shown in Figure 3

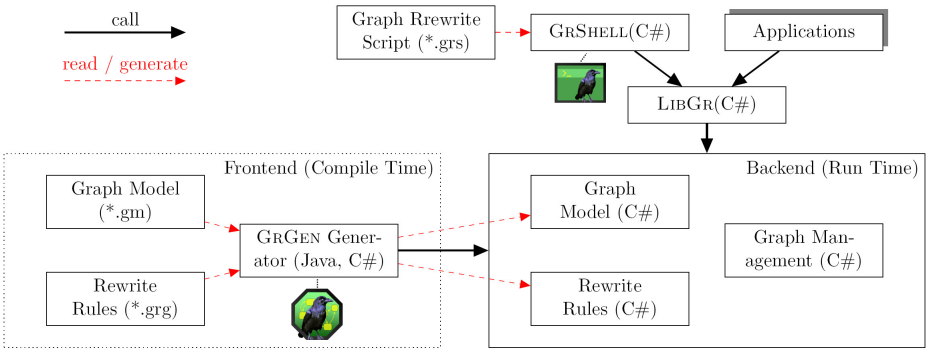

Figure 3: The structure of the GrGen.NET system

For the first HelloWorld task the Shell executes the script given in Figure 4.

```
import helloworld.ecore HelloWorld.grg
xgrs createHelloWorld
show graph ycomp
quit
```

Figure 4: HelloWorld.grs

The first line imports the Ecore metamodel `helloworld.ecore` and loads the transformation rules declared in the rule file `HelloWorld.grg`. The import process automatically generates the GrGen metamodel file `helloworld_ecore.gm` that is used in Figure 1 and Figure 2. We then execute an **e**xtended **g**raph **r**ewrite **s**equences that consists of one application of the `createHelloWorld` rule. The graph rewrite sequences offer multiple operators for controling rule execution and parameter passing between rules, several of them will be introduced later on, here we only execute a single parameterless rule once. The third line is used to `show` the resulting graph in the yComp tool, and finally the shell execution is `quit`.

Let us continue with the model-to-text transformation, Figure 5 shows the corresponding rule. The rule matches a node of type `Greeting` and the corresponding `person` and `message`. If a match is found, it creates a new node of type `StringResult` and assigns the concatenation of the `text` of the `message` and the `name` of the `person` to the `result` attribute. The `StringResult` is then `emit`ted into an XMI file with the rule in Figure 6, which gets `#include`d into the main rule file. The process is controled by the shell script Figure 7. The 5th line is used to `redirect` the output of the `emit` statements from stdout to the specified file.

```
#include "Emitter.gri"

rule messageToText {
    greeting:Greeting;
    greeting -:greetingMessage-> message:GreetingMessage;
    greeting -:person-> person:Person;

    replace {
        result:StringResult;

        eval {
            result.result = message.text + "␣" + person.name + "!";
        }
    }
}
```

Figure 5: HelloWorldExt.grg

```
rule emitString {
    string:StringResult;

    replace {
        emit("<?xml␣version=\"1.0\"␣encoding=\"ASCII\"?>\n");
        // some further lines emitting XMI text not displayed
        emit("␣␣␣␣result=\"" + string.result + "\"/>\n");
    }
}
```

Figure 6: Emitter.gri

```
import helloworldext.ecore result.ecore HelloWorldExt.grg
xgrs createHelloWorldExt
show graph ycomp
xgrs messageToText
redirect emit "Model-to-text-grgen.xmi"
xgrs emitString
quit
```

Figure 7: HelloWorldExt.grs

4 Count matches with certain properties

The next task is to count the number of occurrences of certain graph structures. For each subtask the result needs to be wrapped in a node, this node is created by an application of the rule in Figure 8. The rule creates a node

```
rule createIntResult : (IntResult) {

    modify {
        res:IntResult;

        eval {
            res.result = 0;
        }

        return (res);
    }
}
```

Figure 8: Count.grg

of type `IntResult`, initializes its `result` attribute to `0` and then `return`s the node out to the caller; it must be of type `IntResult` due to the output parameter declaration in the rule header with syntax `: (IntResult)`. (Alternatively we could create the node in the shell with the `new` command which is the prefered way for creating non-trivial initial host graph.) The node with the count has to be written to an XMI file; this is accomplished with a text emitting rule `emitInt` nearly identical to the already introduced one `emitString` available in the file `Emitter.grg`, cf. the SHARE image for details.

The first subtask consists of counting the number of nodes on the host graph. This is achieved by using the GrGen rule shown in Figure 9. The

```
rule countNode(res:IntResult) {
    n:Node;

    modify {
        eval {
            res.result = res.result + 1;
        }
    }
}
```

Figure 9: Count.grg

rule increments the `result` attribute of the `IntResult` parameter by one. To get the count of all the nodes we execute the rule for all matches in the host graph — this can be requested by the graph rewrite sequences calling

the rules by enclosing the rule name in all-brackets, as can be seen on line 4 of Figure 10. Having a closer look at this line we see that the subtask is handled by the successive application of 3 rules. The then-right operator `;>` executes the left sequence and then the right sequence, returning as result of execution the result of the execution of the right sequence. The potential results of sequence execution are *success* equaling `true` and *failure* equaling `false`; a rule which matches counts as success. The `IntResult` returned from the first rule is assigned to a variable `res`. This variable is read before executing the second rule, its value is handed in as input argument to the second rule, in fact to all applications of the second rule. The third rule emits (and deletes) the `IntResult` (it is not handed in, instead it gets matched in the `emitInt` rule). (Alternatively we could count the number number of nodes of a certain type `T` with `show num nodes T` in the GrShell.)

```
import graph1.ecore result.ecore Graph1.xmi Count.grg
include layout.grsi
redirect emit "Count-nodes-grgen.xmi"
xgrs (res)=createIntResult ;> [countNode(res)] ;> emitInt
redirect emit "Count-looping-edges-grgen.xmi"
debug xgrs (res)=createIntResult ;> [countLoopingEdge(res)] ;> emitInt
# the other subtasks follow the two-line scheme given above
quit
```

Figure 10: Count.grs

The count looping edges subtask is interesting because in the shell script calling it the keyword `debug` was prepended before the `xgrs` command (cf. line 6 of Figure 10). This causes sequence execution to start in debug mode, i.e. yComp is started visualizing the host graph and the rule matches of interest, and the sequence is executed stepwise under user control. The situation right after the one match available for `countLoopingEdge` was found is displayed in Figure 12, the graph elements are annotated with the names of the pattern elements which matched them, cf. Figure 11.

```
rule countLoopingEdge(res:IntResult) {
    e:Edge -:src-> n:Node;
    e      -:trg-> n;

    modify {
        eval {
            res.result = res.result + 1;
        }
    }
}
```

Figure 11: Count.grg

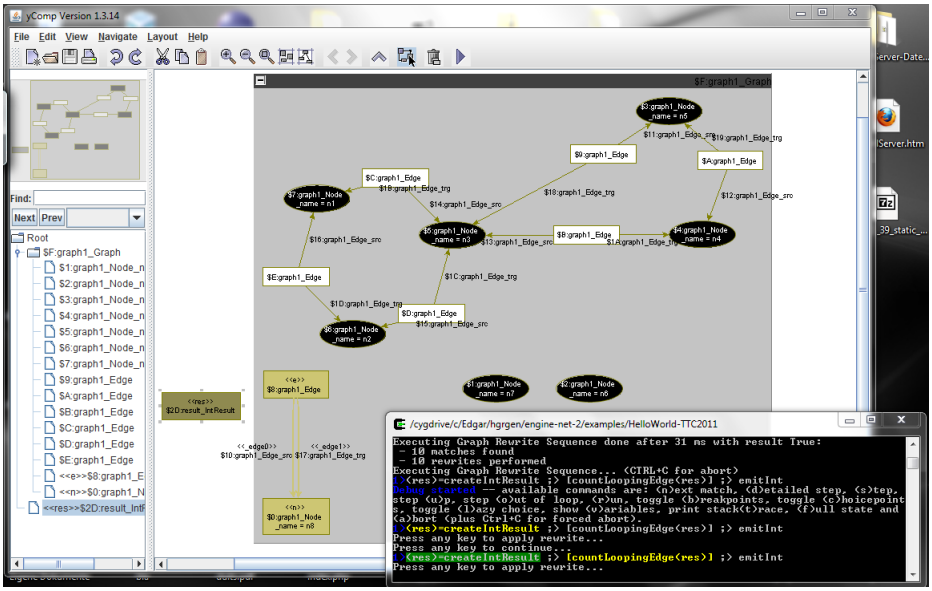

Figure 12: Debugging the sequence calling countLoopingEdge

A major feature of yComp is its high configurability. In line 2 of Figure 10 we `include` a further shell script given in Figure 13, which is used to achieve the nice layout displayed in Figure 12.

```
debug set layout Circular
dump add node Graph group by hidden outgoing edges
dump add node Graph group by hidden outgoing nodes
dump set node Graph color lightgrey
dump set node Node shape circle
dump set node Node color black
dump set node Node textcolor white
dump add node Node infotag name
dump set node Edge color white
```

Figure 13: layout.grsi

You can use one of several available layout algorithms – with hierarchic and organic being the most useful ones, here we use circular. You can configure for every available node or edge type in which color with what node shape or edge style it should be shown, with what attribute values or fixed text as element labels or tags it is to be displayed, or if it should be shown at all. We used it to distinguish the `Node` nodes from the `Edge` nodes. Furthermore you can configure graph nesting by registering edges at certain nodes to define a containment hierarchy, causing the nodes to become displayed as subgraphs containing the elements to which they are linked by the given edges. This can be seen on lines 2 and 3 of Figure 13

which cause all `Node` nodes and all `Edge` nodes to be contained in the graph node.

Figure 14 shows the rule for the optional subtask of matching all dangling edges. It matches an `Edge` node and then uses an `alternative` to match

```
rule countDanglingEdge(res:IntResult) {
    e:Edge;

    alternative {
        missingTrg {
            e -:src->;
            negative {
                e -:trg->;
            }

            modify { }
        }
        missingSrc {
            e -:trg->;
            negative {
                e -:src->;
            }

            modify { }
        }
    }

    modify {
        eval {
            res.result = res.result + 1;
        }
    }
}
```

Figure 14: Count.grg

either the `missingSrc` pattern or the `missingTrg` pattern. The `missingTrg` pattern matches the edge to the source node and uses a `negative` application condition (NAC) to ensure that the graph edge has no target node. A `negative` pattern causes the matching of the enclosing pattern to fail if it is found in the graph. Likewise, the `missingSrc` only matches an graph edge with a target node, but without a source node.

The rule `countIsolatedNode` in Figure 15 is not matching as soon as one of the negatives is found. The `countCycle` in Figure 16 is a direct encoding of the specification, so there's no need to go into depth here.

```
rule countIsolatedNode(res:IntResult) {
    n:Node;
    negative {
        n <-:src-;
    }
    negative {
        n <-:trg-;
    }

    modify {
        eval {
            res.result = res.result + 1;
        }
    }
}
```

Figure 15: Count.grg

```
rule countCycle(res:IntResult) {
    n1:Node <-:src- :Edge -:trg-> n2;
    n2:Node <-:src- :Edge -:trg-> n3;
    n3:Node <-:src- :Edge -:trg-> n1;

    modify {
        eval {
            res.result = res.result + 1;
        }
    }
}
```

Figure 16: Count.grg

5 Reverse Edges

To solve this task we need to reverse all edges. The GrGen rule shown in Figure 17 employs retyping also known as relabeling on the GrGen edges to accomplish this task.

```
rule reverseEdge {
    alternative {
        Src {
            -src:src->;

            modify {
                -:trg<src>->;
            }
        }
        Trg {
            -trg:trg->;

            modify {
                -:src<trg>->;
            }
        }
    }

    modify { }
}
```

Figure 17: Reverse.grg

Retyping is specified with the syntax `y:t<x>`: this defines `y` to be a retyped version of the original node `x`, retyped to the new type `t`; for edges the syntax is `-y:t<x>->`. After applying the rule with `xgrs [reverseEdge]` all source nodes are target nodes and all target nodes are source nodes. This approach naturally reverses even dangling edges.

6 Simple Migration

To solve this task we need to migrate the graph from the graph metamodel used in the previous tasks to another graph metamodel which is characterized by introducing a superclass `GraphComponent` for `Node` nodes and `Edge` nodes. Since to the target metamodel has a similar structure, we simply use retyping as introduced in the previous section to migrate the graph, with the rules given in Figure 18 controled by the sequence given in Figure 19. The name mangling from Ecore import is kept from now an as removing it would render the types ambiguous.

```
using graph1__ecore, graph2__ecore;

rule migrateGraph {
    n:graph1_Graph;
    modify {
        :graph2_Graph<n>;
    }
}
rule migrateNode {
    n:graph1_Node;
    modify {
        migrated:graph2_Node<n>;
        eval { migrated._text = n._name; }
    }
}
rule migrateEdge {
    e:graph1_Edge;
    modify {
        migrated:graph2_Edge<e>;
        eval { migrated._text = ""; }
    }
}
rule migrateEdgeSrc {
    -e:graph1_Edge_src->;
    modify {
        -migrated:graph2_Edge_src<e>->;
        eval { migrated.index = e.index; }
    }
}
rule migrateEdgeTrg {
    -e:graph1_Edge_trg->;
    modify {
        -migrated:graph2_Edge_trg<e>->;
        eval { migrated.index = e.index; }
    }
}
rule migrateGraphEdges {
    -e:graph1_Graph_edges->;
    modify {
        -migrated:graph2_Graph_gcs<e>->;
        eval { migrated.index = e.index; }
    }
}
rule migrateGraphNodes {
    -e:graph1_Graph_nodes->;
    modify {
        -migrated:graph2_Graph_gcs<e>->;
        eval { migrated.index = e.index; }
    }
}
```

Figure 18: SimpleToEvolved.grg

```
xgrs migrateGraph* & migrateNode* & migrateEdge* & migrateEdgeSrc*\
       & migrateEdgeTrg* & migrateGraphEdges* & migrateGraphNodes*
```

Figure 19: SimpleToEvolved.grs

In Figure 19 we iterate each rule until it does not match any more. This is denoted by the postfix star `*` causing the preceding sequence to be iterated as long as it succeeds. The result of a star iteration is always success (in contrast to the plus `+` postfix which requires the preceding sequence to match at least once in order to succeed), so the complete sequence linked by strict conjunction operators `&` succeeds (`true`), too. Disjuntion `|` is available as well, so are the lazy versions `&&` and `||` of the operators not executing the right sequence in case the result of the left sequence already determines the outcome.

The solution for the second (optional) target metamodel is similar to the first, so we only highlight the key difference: the second metamodel realizes edges by edges and not nodes anymore. Thus model migration requires a non-isomorphic transformation step for edges, here we use the rule shown in Figure 20.

```
rule migrateEdge {
    e:graph1_Edge;
    e <-graphEdge:graph1_Graph_edges-;
    e -:graph1_Edge_src-> src:Node;
    e -:graph1_Edge_trg-> trg:Node;

    modify {
        delete(e);

        src -migrated:graph3_Node_linksTo-> trg;

        eval {
            migrated.index = graphEdge.index;
        }
    }
}
```

Figure 20: SimpleToMoreEvolved.grg

Striving for perfection, we order the `linksTo` references by migrating the `graphEdge.index` and employing an additional fix-up rule that ensures that the index falls into the interval $0 \ldots |\texttt{linksTo}| - 1$. Thus the (outgoing) edges are ordered the same way as they are ordered in the original graph.

7 Delete Node with Specific Name and its Incident Edges

Deleting a node with a given name is a trivial task, as can be seen in rule given in Figure 21 using `modify` mode explicitely `delete`ing the matched node `if` it bears the name searched for.

```
rule deleteN1 {
    n:graph1_Node;

    if {n._name == "n1";}

    modify {
        delete(n);
    }
}
```

Figure 21: Delete.grg

The (optional) subtask of also deleting all incident edges is more interesting. Figure 22 shows the corresponding GrGen rule that matches the node with the name `n1` and all incident edges in an `iterated` fashion. The `iterated` construct munches eagerly the contained pattern as long as it is available in the graph and not yet matched; it succeeds even if the contained pattern is not available in the graph, in contrast to the similar `multiple` construct which requires the pattern to be available at least once causing matching of the enclosing pattern to fail otherwise. Since the `replace` parts are empty, all matched elements are deleted.

```
rule deleteN1AndAllIncidentEdges {
    n:graph1_Node;

    if {n._name == "n1";}

    iterated {
        n <-- e:graph1_Edge;

        replace {
        }
    }

    replace {
    }
}
```

Figure 22: Delete.grg

8 Insert Transitive Edges

The last (optional) task is to compute $R \cup R^2$ from a graph representation of a relation R. The obvious way to solve this task is to provide a rule `insertTransitiveEdge` that inserts a transitive edge and to apply it to all matches by `xgrs [insertTransitiveEdge]`. However, this may insert the same edge multiple times. Thus, instead we insert GrGen edges of type `Edge` (anonymously with `-->`), then we remove multiply inserted GrGen edges and finally transform the remaining GrGen edges into edges conforming to the metamodel: `[addTransitiveEdge] && removeDuplicatedEdges* && transformEdge*`

Figure 23 shows the rule that inserts the temporary transitive GrGen edges and Figure 24 the rule that removes temporary edges inserted too often as well as the rule that replaces the temporary edges by the edge nodes including the links. The last one is a bit complicated due to the wish for proper index handling.

The `hom(n1,n2,n3)` statement is a newly introduced construct which allows a homomorphic matching for the contained nodes, i.e. they *can* be matched to the same host graph node.

```
rule addTransitiveEdge {
    n1:graph1_Node;
    n2:graph1_Node;
    n3:graph1_Node;
    hom(n1,n2,n3);

    n1 <-:graph1_Edge_src- :graph1_Edge -:graph1_Edge_trg-> n2;
    n2 <-:graph1_Edge_src- :graph1_Edge -:graph1_Edge_trg-> n3;

    negative {
        n1 <-:graph1_Edge_src- :graph1_Edge -:graph1_Edge_trg-> n3;
    }

    modify {
        n1 --> n3;
    }
}
```

Figure 23: Transitive.grg

9 Conclusion

In this paper we presented a GrGen.NET solution to the Hello World! challenge of the Transformation Tool Contest 2011. We were able to solve all tasks of the challenge, including the optional tasks, introducing the reader alongside to a respectable amount of the functionality of GrGen.NET.

```
rule removeDuplicatedEdges {
    n1:graph1_Node;
    n2:graph1_Node;
    hom(n1,n2);

    n1 -e:Edge-> n2;
    n1 --> n2;

    modify {
        delete(e);
    }
}

rule transformEdge {
        n1:graph1_Node;
        n2:graph1_Node;
        hom(n1,n2);

        n1 -oldEdge:Edge-> n2;

        // Search for edge with highest index
        graph:graph1_Graph -:graph1_Graph_nodes-> n1;
        graph -graphEdge:graph1_Graph_edges->;

        negative {
                graph -otherGraphEdge:graph1_Graph_edges->;

                if { otherGraphEdge.index > graphEdge.index; }
        }

        modify {
                delete(oldEdge);

                n1 <-src:graph1_Edge_src- e:graph1_Edge
                            -trg:graph1_Edge_trg-> n2;
                graph -newGraphEdge:graph1_Graph_edges-> e;

                eval {
                        src.index = 0;
                        trg.index = 0;
                        newGraphEdge.index = graphEdge.index + 1;
                }
        }
}
```

Figure 24: Transitive.grg

References

[1] Mazanek, S.: Hello World! An Instructive Case for TTC. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2011/cases/ttc2011_submission_3.pdf (2011)

[2] Blomer, J., Geiß, R., Jakumeit, E.: The GrGen.NET User Manual. http://www.grgen.net (2011)