GRAPE – A Graph Rewriting and Persistence Engine

Jens H. Weber(✉)

LEADlab, Department of Computer Science,
University of Victoria, BC Victoria, Canada
jens@acm.org

Abstract. Graph-based data structures are fundamental to many applications in Computer Science and Software Engineering. Operations on graphs can be formalized as graph transformations or graph rewriting rules and a rich theoretical underpinning has been developed in the research community that supports reasoning about the properties of graph transformation systems. Various tools exist for developing graph transformations, including visual editors as well as textual languages that can be integrated with general purpose programming languages. This paper introduces *Grape* (Graph Rewriting and Persistence Engine), a hybrid, embedded Domain Specific Language (DSL) for Clojure. *Grape* is a lightweight approach to computing with persistent graphs within Clojure. It combines the ease of use of a textual DSL with a graphical visualization that is inlined with the program code when needed to aid comprehension and documentation of graph rewriting rules. Moreover, *Grape* supports persistence, programmed transactions and backtracking.

Keywords: Graph transformations · Tool support · DSL · Persistence · Clojure

1 Introduction

Graph-based data structures play an important role in many applications of Computer Science and Software Engineering. Operations on graphs can be formalized with graph transformation rules (also referred to as graph rewriting rules). A rich theoretical background exists on the formal properties of graph transformation systems (GTS) and various tools have been developed in support of their development [1, 2]. Current tool support ranges from visual development environments (with underlying transformation engines) to textual languages that may be integrated with general purpose programming languages. Visual development environments for GTS provide the benefit of a more intuitive, graphical way of specifying operations on graphs. However, these tools are often expensive to build and maintain. Moreover, visual development tools may pose usability challenges, as developers need to learn how to use them properly [3]. Other challenges pertain to the integration of visual programming in the overall software development lifecycle, such as the integration with other parts of a software program, configuration management and merging of different versions, etc.

Textual graph transformation languages provide a more lightweight approach to developing graph-based computations and avoid many of these challenges. However,

J. de Lara and D. Plump (Eds.): ICGT 2017, LNCS 10373, pp. 209–220, 2017.
DOI: 10.1007/978-3-319-61470-0_13

textual graph rewriting rules may not be as easy to understand as their visual counterparts. Hybrid approaches in which graph transformations are specified textually but documented visually have been suggested as a compromise. However, textual programs and visual documentations are often not well integrated, which impede incremental and dynamic development of graph-based programs.

Another concern with existing graph transformation tools pertains to their scalability and the persistence of large graphs. Most current tools process graph models in main memory, which imposes practical limits to the scalability of their applications. Moreover, most current tools do not provide support for complex transactions of programmed graph rewrite operations, in the sense of the typical ACID properties (Atomicity, Consistency, Integrity and Durability).

This paper introduces *Grape* (Graph Rewriting and Persistence Engine) as a lightweight, hybrid GTS development tool that seeks to address the above concerns. *Grape* provides a lightweight, hybrid GTS extension to the Clojure programming language. Graph transformations are programmed textually, but visualized graphically, inline with the program code, within the LightTable general purpose text editor[1]. *Grape* utilizes the highly scalable Neo4 J graph database for graph persistence and transaction support. *Grape* programs provide full support for transactions, including backtracking. It has been made available under an open source license on Github.

The rest of this paper is structured as follows. The next section provides a short introduction to graph rewriting and graph transformation systems. Section 3 provides an overview of the work related to tool support for developing graph transformation systems in software applications. Section 4 provides an overview of the architecture of *Grape*, while Sect. 5 introduces the *Grape* DSL within Clojure and demonstrates the use of LightTable as a lightweight hybrid development environment. Finally, Sect. 6 provides concluding remarks and an outlook on future work.

2 Graph Rewriting and Graph Transformation Systems

2.1 Directed, Attributed, and Labeled (DAL) Graphs

A directed graph is a data structure that consists of a set of nodes N and a set of edges E, such that each edge $e \in E$ has a source $s(e)$ and target $t(e)$ in N. Labeled graphs allow nodes and edges to be labeled, i.e., a labeling function $l(o)$ associates each graph object $o \in N \cup E$ with a set of labels. Attributed graphs further allow the association of graph objects with attribute properties, i.e., an attributation function $a(o)$ associates each graph object with a set of key/value pairs.

2.2 Graph Transformation Rules

A graph transformation rule $L \rightarrow R$ consists of two graphs, commonly referred to as *left-hand side* (LHS) and *right-hand side* (RHS), respectively. The LHS specifies a

[1] http://lighttable.com.

subgraph pattern to find in a given graph (the *host graph*) and the RHS defines how a found subgraph is to be rewritten as a result of the transformation. In other words, LHS acts as a precondition of the rule, while RHS specifies its post-condition. Graph transformation rules are applied in a three-stepped process:

1. **Match:** The hostgraph is searched for a subgraph that matches the rule's LHS.
2. **Delete:** Graph objects in the rule's LHS that are not in the rule's RHS are deleted from the hostgraph.
3. **Add:** Graph elements in the rule's RHS that are not included in its LHS are added to the host graph.

When a rule matches multiple subgraphs in the host graph, a match is chosen non-deterministically.

The above process of applying graph transformation rules may result in a structure that is not a graph. This is the case when a node is deleted that is a source or a target of an edge in the host graph that is not part of the match found for the rule's LHS. The graph transformation community has developed different approaches on how to prevent this situation. One approach is to permit the deletion of such a node and to delete all edges that may be left "dangling" in the host graph. This approach is based on the "*single pushout*" (SPO) theory for graph transformations [1]. Another approach is to prohibit the application of a transformation rule in cases where its execution would result in dangling edges. This more restrictive condition is commonly referred to as the *gluing condition* in the *double-pushout* (DPO) theory for graph transformations [1].

Different approaches exist as well with respect to the type of morphism that is used to find a match for a rule's LHS in the host graph. Isomorphic matching requires that each object in a rule's LHS matches a distinct graph object in the host graph, while homomorphic matching allows different objects in a rule's LHS to match to the same graph object in the host graph.

Graph transformation rules may also have a set of negative application conditions (NACs) that may be used to prevent rule application in certain contexts. NACs are an important concept for many practical applications of GTS [4]. NACs can be specified as graph patterns (and conditions on attributes) that extend a rule's LHS. The application of graph transformation rules with NACs becomes a four-step process:

1. **Match:** The hostgraph is searched for a subgraph that matches the rule's LHS.
2. **Check:** Attempt to extend the matched subgraph with a match for any of the rule's NACs. If this is possible, prevent rule application in this context.
3. **Delete:** Graph objects in the rule's LHS that are not in the rule's RHS are deleted from the hostgraph. (Validate gluing condition for DPO rewriting approach.)
4. **Add:** Graph elements in the rule's RHS that are not included in its LHS are added to the host graph.

A graph transformation system (GTS) is defined as a set of graph transformation rules. A graph grammar is a GTS with a defined start graph. Graph grammars are commonly used for defining and parsing graph-based languages. In this paper, we are less concerned with the definition of graph-based languages and rather focus on engineering applications of graph rewriting. Such applications typically require imperative control structures to govern the execution of graph transformation rules. A programmed GTS is a GTS that has been associated with an imperative control program.

3 Related Work: Tool Support for Graph Rewriting

3.1 Visual Tools

PROGRES is an integrated development environment for visual developing of programmed GTS [5]. PROGRES provides a powerful specification language and uses a graph database for scalability and persistence with support for complex transactions and backtracking. However, PROGRES lacks integration with general purpose programming languages and its development has been discontinued.

FUJABA supports the development of programmed GTS for Java [6]. Control structures are specified using “story diagrams”, a combination of activity diagrams and graph rewriting rules. Graph transformations are carried out in main memory. Graphs can be serialized for file-based storage. FUJABA generates Java code, which can be integrated with general purpose Java programs. FUJABA does not support transactions and backtracking.

AGG is a visual development environment for GTS [7, 8]. Transformation rules are executed based on an interpreter and the graph is held in main memory. The definition of control structures is supported. An API allows the integration with the Java general purpose programming language.

GROOVE is another visual GTS development environment that is particularly suitable for formal verification and state space exploration [9]. The graph is held in main memory and transformations are executed by an interpreter. Control structures are provided in form of a dedicated scripting language. Backtracking and transactions are not supported. Integration with general purpose programming languages is possible through an internal (undocumented) API.

Henshin is a visual graph transformation tool based on the Eclipse modelling framework (EMF) [10]. Graph transformations are carried out by an interpreter that can be interfaced with general purpose programming languages (Java) through an API. Graphs are kept in main memory. No transactions or backtracking is supported.

3.2 Textual Tools

Viatra is an Eclipse plugin that provides a textual language for specifying graph transformations [11, 12]. Control structures are specified using abstract state machines.

GrGen provides a textual language to define graph transformations on object graphs held in main memory (C# or Java) or kept in a relational database [13]. GrGen generates C# code or .net assemblies.

SDMlib is an internal DSL for graph transformations with Java [14]. Graphs are kept in main memory and can be persisted in a file. Graphs and graph transformations can be visually documented. Transactions and backtracking is not supported.

FunnyQT is an internal graph transformation DSL for Clojure [15]. Homomorphic as well as isomorphic graph pattern matching is supported. FunnyQT provides a framework for in-place graph rewriting with arbitrary Clojure actions. The semantics of rewriting rules is not grounded in a particular theory (such as DPO or SPO). Graphs are held in main memory and transactions are not supported.

4 The Grape Architecture

Grape provides an internal domain-specific language (DSL) for programming GTS with Clojure (cf. Fig. 1). *Grape* uses the Neo4J graph database for storing the host-graph, i.e., the graph is not held in main memory. Graph transformation rules defined in the *Grape* DSL are translated to Cypher, Neo4J's native query language. Cypher provides powerful constructs for graph pattern matching, which are leveraged by *Grape*. *Grape* also provides a visualizer for graph transformation rules based on GraphViz [16]. *Grape* does not depend on any particular development tool or IDE, but provides a convenient integration with LightTable, which allows developers to visualize their graph transformation rules "in line" with their textual definition. Independently of the editor used, the *Grape* visualizer also provides functions to generate visual representations of GTS in the file system. Neo4J also provides an extensible graph browser that can be used to visualize graphs.

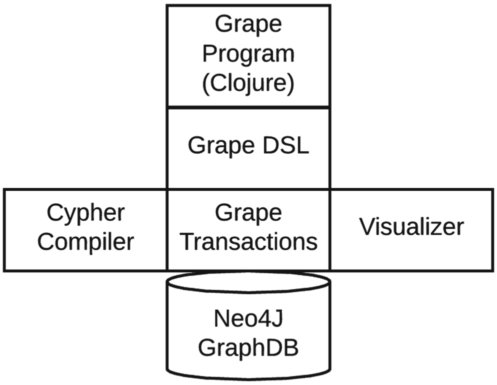


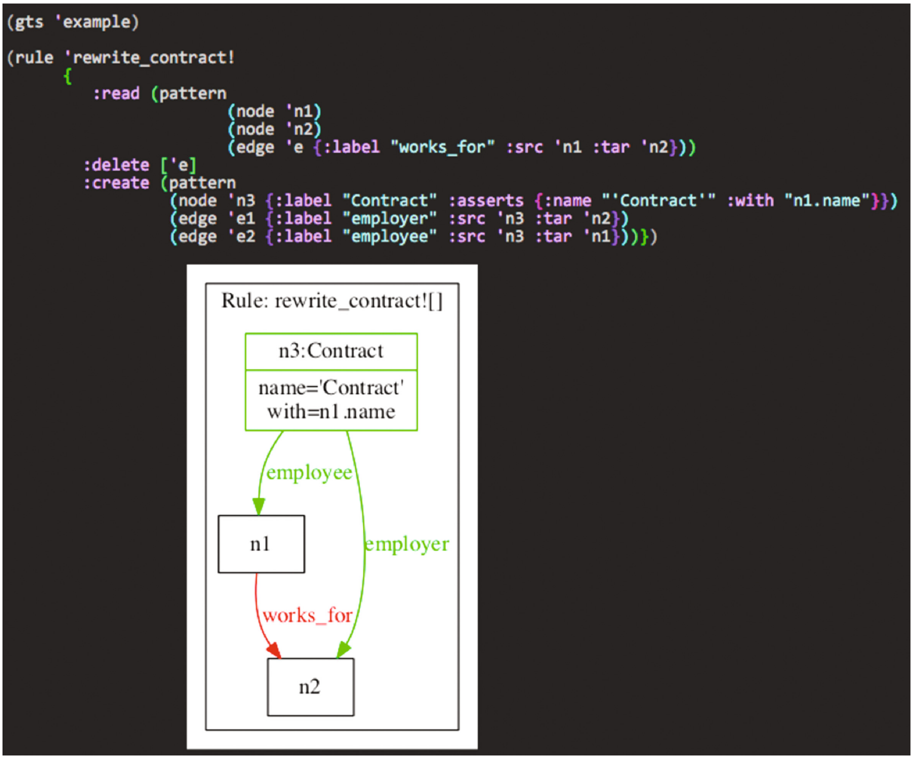
Fig. 1. The *Grape* architecture

Grape provides support for complex transactions of programmed graph transformations with full support for backtracking. This functionality is based on Neo4J's flat transaction model and implemented in *Grape's* transaction module. (Neo4J native "flat" transaction model needs to be extended to support nested transactions, as required for the desired backtracking behaviour.)

5 A Taste of Grape – Introduction to Programming with Grape

5.1 Simple Rule Definition and Execution

The *Grape* DSL uses native Clojure syntax. Graphs are schema-less (untyped), following the schema-less design philosophy of no-SQL databases such as Neo4 J. Figure 2 provides a first example of the *Grape* language. It defines a new GTS (using the `gts` form) and a simple transformation rule that matches two graph nodes that are connected with a `works_for` edge in order to replace that edge with a new `Contract` node with `employee` and `employer` edges.



```
(gts 'example)

(rule 'rewrite_contract!
      {
         :read (pattern
                       (node 'n1)
                       (node 'n2)
                       (edge 'e {:label "works_for" :src 'n1 :tar 'n2}))
        :delete ['e]
        :create (pattern
                 (node 'n3 {:label "Contract" :asserts {:name "'Contract'" :with "n1.name"}})
                 (edge 'e1 {:label "employer" :src 'n3 :tar 'n2})
                 (edge 'e2 {:label "employee" :src 'n3 :tar 'n1}))})
```

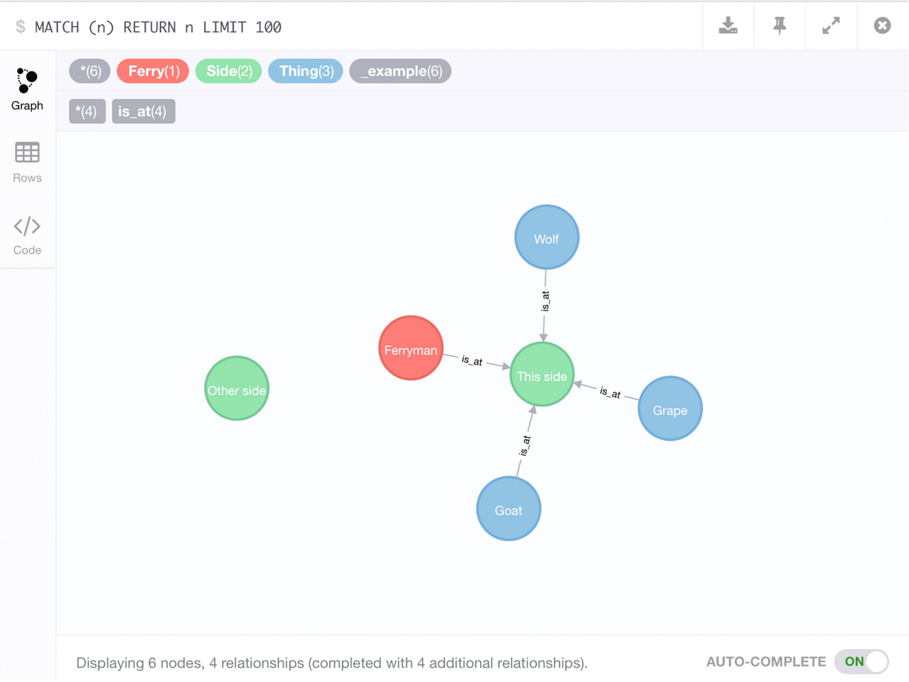
Fig. 2. Definition of a GTS and a simple transformation rule

As the example in Fig. 2 shows, *Grape* rules have three main parts: *read*, *delete* and *create*. (Of course, some of these parts may be missing. For example, a rule that simply creates a graph structure will not have a *read* or *delete* part.) Nodes and edges are identified by id symbols, e.g., ‘`n1`. Labels are defined using the `:label` key and assertions on attributes are defined using the `:asserts` key. In the above example, two attributes are defined for the new `Contract` node: the first attribute (`name`) is assigned the literal value “Contract”, while the second attribute (`with`) is assigned the value of the `name` attribute of node `n1`.

The visualization of the rule in Fig. 2 is automatically generated by *Grape* and inlined after the textual definition (if LightTable is used as the editor). The visualization uses the popular representation of graph transformation rules where LHS and RHS are merged into the same graph, with red colours showing deleted graph elements and green colour showing new ones. *Grape* programmers who do not use LightTable can still make use of the rule visualization by generating visual documentation in the project’s file system.

Once a rule has been defined (as above), it can be invoked simply by calling an equally named function, i.e., by calling `(rewrite_contract!)` in the above example. Calling this function will return true if the `rule` could be applied and `false` otherwise. Its invocation will non-deterministically select a possible match and attempt its transformation. The usual Clojure control structures can be used with this function. For example, if all `works_for` occurrences are to be rewritten, a programmer may simply use `(while (rewrite-contract!))`.

Grape does not implement its own tool for visualizing the state of the hostgraph, since Neo4J provides a powerful graph browser as part of its community edition. Figure 3 shows a hostgraph visualized with the Neo4J browser for the Ferryman example discussed at the end of this section.



```
$ MATCH (n) RETURN n LIMIT 100
```

Displaying 6 nodes, 4 relationships (completed with 4 additional relationships).

Fig. 3. Hostgraph visualization using Neo4J’s graph browser

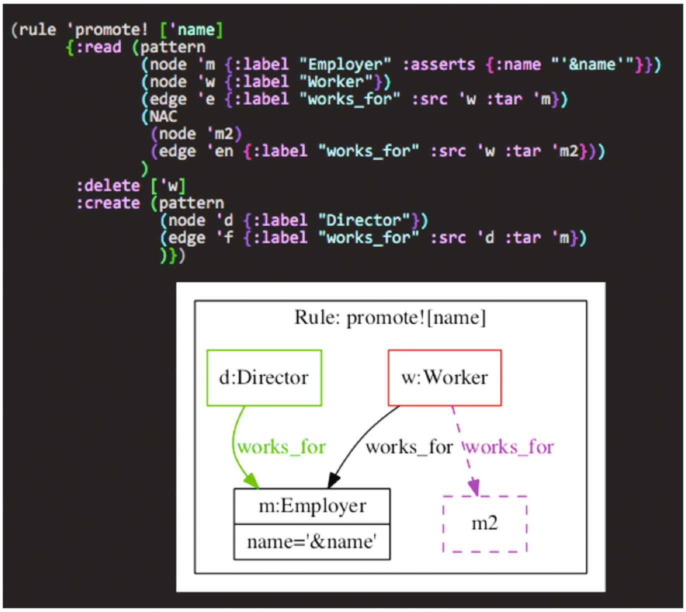
5.2 Customizing Matching and Rewrite Semantics

Grape performs isomorphic graph matching by default. This means that in the above example, self-employment relationships (where n1 and n2 match the same node in the host graph) would not be matched. If homomorphic matching is desired, a `:homo` key can be added to the definition of the *read* pattern.

Furthermore, *Grape* rules implement SPO rewrite semantics by default, i.e., any "dangling" edges that may arise from deleting nodes are automatically deleted as well. If the more restrictive DPO semantics is desired, an `:dpo` key can be added to a rule, which results in checking the *gluing condition* prior to executing the transformation.

5.3 Parameters and NACs

Grape transformation rules can by parameterized and contain an arbitrary number of NACs. Figure 4 provides an example for a rule "promote!" with a formal parameter and one NAC. It searches for a `Worker` who `works_for` an `Employer` with a given name (parameter) and replaces the `Worker` node with a `Director` node, if that worker does not also work for another Employer (i.e., if there is not a `work_for` edge from node `w` to another node in the host graph). Graph patterns defined in NACs are visualized with dashed lines and using a different colour for each defined NAC, if multiple NACs are defined. Invoking a parameterized rule uses the normal Clojure parameter passing, e.g., `(promote! “John”)` in the above example.



```
(rule 'promote! ['name]
      {:read (pattern
              (node 'm {:label "Employer" :asserts {:name "'&name'"}})
              (node 'w {:label "Worker"})
              (edge 'e {:label "works_for" :src 'w :tar 'm})
              (NAC
               (node 'm2)
               (edge 'en {:label "works_for" :src 'w :tar 'm2}))
              )
       :delete ['w]
       :create (pattern
                (node 'd {:label "Director"})
                (edge 'f {:label "works_for" :src 'd :tar 'm})
                )})
```

Fig. 4. A parameterized rule with a single NAC

5.4 Transactions

Grape supports transactions of complex graph rewriting operations consisting of multiple transformation rules, with full backtracking support. Transactions are created using the `transact` form. The left-hand side of Fig. 5 shows a simple transaction consisting of a sequence of programmed transformation rules.

```
(transact
  (apl `rule1)
  (apl `rule2)
  ..
  (apl `ruleN))
```

```
(transact
  (apl `rule1 p)
  (bind `out `n)
  (apl `rule2 (consult `out))
  ..)
```

Fig. 5. Defining simple transactions (left) and defining parameterized transaction operations with parameter passing (right).

Note that *programmed* transformation rules require the `apl` form to define each rule application. Writing `(rule1)` instead of `(apl ‘rule1)` would cause the Clojure REPL to execute “rule1” at the time of defining the transaction. Of course, we could have used a macro instead of a regular Clojure function for defining the transact form to prevent this behaviour. However, we intentionally decided to avoid macros in the development of *Grape* to keep the code simple and functional.

Grape also allows parameter passing between different transformation rules in a transaction. This is realized using two forms: `bind` and `consult`. The first form binds a graph element (node or edge) matched by the previously executed graph transformation rule to a symbol, while the second form (`consult`) dereferences the bound graph object for the purpose of passing it to a subsequent transformation rule. The right-hand side of Fig. 5 shows an example. Here, the graph element `n` matched in `rule1` is bound to symbol `out` and then passed to `rule2` as a parameter.

```
(defn tx [p]
  (transact
    (apl `rule1 p)
     ..
    ))
```

```
(attempt (tx! “hello”))
```

Fig. 6. Defining and attempting to execute a parameterized transaction.

The `transact` form returns a value that can be passed to the `attempt` form for execution. Of course, parameterized transactions can be defined as regular Clojure functions, using the `defn` form. Figure 6 shows an example of defining a transaction with an example parameter `p` (left) and attempting to execute it with an example argument “hello” (right). The result of executing an `attempt` form is *true* or *false*, depending on whether the transaction succeeded.

5.5 Control Structures

As *Grape* is an embedded DSL the full breadth of control structures of the host language (Clojure) can be utilized for programming with graph transformation rules. (An example was given at the end of Sect. 5.1.) Moreover, since Clojure is a JVM-based language, other JVM languages can also be used.

A limitation of using the GPL host language's control structures is that they provide no support for backtracking in operations that compose multiple graph transformations. *Grape* provides a set of native DSL control structures that can be used to program complex graph operations where backtracking is desired. In particular, *Grape* provides control structures for loops, non-deterministic choice, and negative graph tests.

Figure 7 shows an example program for solving the well-known Ferry Crossing puzzle [17]. In that problem, a ferryman has the task to safely transport three items across the river. He can only take one item at a time. Two unsafe states exist: (1) the wolf will eat the goat and (2) the goat will eat the cabbage, if left unsupervised.

```
(until 'all_on_the_other_side?
         (transact (choice (apl 'ferry_one_over!)
                           (apl 'cross_empty!))
                   (avoid (apl 'wolf-can-eat-goat?)
                          (apl 'goat-can-eat-grape?)))))
```

Fig. 7. Example using *Grape* control structures

The `until` form is used to define a loop with a break condition given as its first argument, followed by a set of *Grape* transactions that are to be executed in each iteration. The break condition '`all_on_the_other_side?` is a *graph test*. It is defined as a regular *Grape* graph transformation rule that only has a *read* part. The `choice` form takes a list of *Grape* rule applications or transactions and non-deterministically selects one of them for execution. Finally, the `avoid` form takes a list of graph tests and checks whether any of them have a match in the host graph. In that case, the current state of the graph exploration is considered a failure and the program will backtrack. Note that the `avoid` form is not strictly necessary for expressiveness, since *Grape* rules support the definition of NACs. However, we believe that its existence may increase the readability and conciseness of programs.

The program in Fig. 6 is an example for implementing a graph exploration search algorithm in *Grape*. Essentially, the program implements a forward rule-chaining algorithm. Rule selection by the `choice` operator is non-deterministic, which means that the above search is not guaranteed to find a solution (and to terminate). Of course, the graph transformation rules can be defined to limit the search space. A common approach is to define a cost for each ferry crossing and allocate a budget. This can be done using graph attribute assignments and application conditions. In the future, we are interested in extending *Grape* with a heuristics guided choice operator that uses a utility function to aid the prioritization of alternative rule applications.

6 Conclusions and Future Work

The choice of a graph transformation tool ultimately depends on the application it is used for. Several graph transformation tools have been developed and made available. We found that for our applications, graph persistence, scalability, transaction handling and the ability to easily integrate with general purpose programming languages were important requirements. Moreover, we found that the user interfaces of fully visual graph transformation tools often create usability challenges and do not integrate seamlessly with modern, distributed software development practices, e.g., versioning, merging, test-driven development, etc.

In this paper, we have introduced *Grape* as a lightweight, hybrid graph transformation engine embedded in Clojure. *Grape* is highly scalable, as graphs are not kept in main memory but in a graph database (Neo4J scales to graphs consisting of tens of billions of nodes). *Grape* is considered a “hybrid” tool, since rules are authored textually but visualized graphically “in line” with the textual code (if LightTable is used as the editor). Since *Grape* programs are authored with an embedded DSL in Clojure, they can easily be interfaced with the rest of a software program. Moreover, *Grape* provides full support for complex transactions, including backtracking.

Grape has been made available for public use on Github[2]. So far, we have used *Grape* in the development of one small-sized application in the medical domain. The source code for this example application is also available on Github[3]. While in this first application we did not make use of some of *Grape*’s advanced concepts (such as complex transactions and backtracking), it illustrates nicely how easy it is to integrate a *Grape* GTS with the rest of a typical Web-based software system.

There are many avenues for future work on improving *Grape*. We already mentioned at the end of the last section the plan to add a heuristics guided choice operator, to direct the selection of alternative graph transformation rules in complex transactions. Moreover, *Grape* currently operates on untyped graphs. This provides a great degree of flexibility but also increases the likelihood of specification errors. We will be adding the option of working with typed graphs in the future. Another worthwhile extension to *Grape* would be the addition of path expressions. Neo4J provides support for powerful graph expressions in its query language Cypher. We expect to be able to use this feature as a basis for implementing path expressions in *Grape*.

Finally, we are intending to extend *Grape* with respect to supporting bidirectional transformations between graph structures. Triple Graph Grammars (TGG) have been proposed and successfully used for bidirectional graph model synchronization problems. Their integration in *Grape* is planned for a future release [18].

[2] https://github.com/jenshweber/grape.

[3] https://github.com/sdiemert/app-project.

References

1. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation - Volume 1: Foundations. World Scientific Publishing Company (1999)
2. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation - Volume 2: Applications, Languages, Tools. World Scientific Publishing Company (1999)
3. Rouly, J.M., Orbeck, J.D., Syriani, E.: Usability and suitability survey of features in visual ides for non-programmers. In: Proceeding of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools, New York, NY, USA, pp. 31–42 (2014)
4. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Informaticae **26**(3), 287–313 (1996)
5. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In: Schäfer, W., Botella, P. (eds.) ESEC 1995. LNCS, vol. 989, pp. 219–234. Springer, Heidelberg (1995). doi:10.1007/3-540-60406-5_17
6. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proceeding of the 22nd Intl Conference on Software Engineering, New York, NY, USA, pp. 742–745 (2000)
7. Taentzer, G.: AGG: a graph transformation environment for modeling and validation of software. In: Applications of Graph Transformations with Industrial Relevance, pp. 446–453 (2003)
8. Runge, O., Ermel, C., Taentzer, G.: AGG 2.0 – new features for specifying and analyzing algebraic graph transformations. In: Applications of Graph Transformations with Industrial Relevance, pp. 81–88 (2011)
9. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Int. J. Softw. Tools Technol. Transf. **14**(1), 15–40 (2012)
10. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Model Driven Engineering Languages and Systems, pp. 121–135 (2010)
11. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Proceedings of the 2006 ACM Symposium on Applied Computing, New York, NY, USA, pp. 1280–1287 (2006)
12. Bergmann, G., et al.: Viatra 3: a reactive model transformation platform. In: Theory and Practice of Model Transformations, pp. 101–110 (2015)
13. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: a fast SPO-based graph rewriting tool. In: Graph Transformations, pp. 383–397 (2006)
14. Priemer, D., George, T., Hahn, M., Raesch, L., Zündorf, A.: Using graph transformation for puzzle game level generation and validation. In: Graph Transformation, pp. 223–235 (2016)
15. Horn, T.: Graph pattern matching as an embedded clojure DSL. In: Graph Transformation, pp. 189–204 (2015)
16. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz— open source graph drawing tools. In: Graph Drawing, pp. 483–484 (2001)
17. Gasarch, W.: Review of algorithmic puzzles by Anany Levitin and Maria Levitin. SIGACT News **44**(4), 47–48 (2013)
18. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Graph Transformations, pp. 411–425 (2008)