A Foundation for Functional Graph Programs: The *Graph Transformation Control Algebra* (GTA)

Jens H. Weber
Department of Computer Science
University of Victoria
Victoria, Canada, BC
jens@acm.org

Applications of graph transformation (GT) systems often require control structures that can be used to direct GT processes. Most existing GT tools follow a stateful computational model, where a single graph is repeatedly modified *in-place* when GT rules are applied. The implementation of control structures in such tools is not trivial. Common challenges include dealing with the non-determinism inherent to rule application and transactional constraints when executing compositions of GTs, in particular atomicity and isolation. The complexity of associated transaction mechanisms and rule application search algorithms (e.g., backtracking) complicates the definition of a formal foundation for these control structures. Compared to these stateful approaches, *functional* graph rewriting presents a simpler (stateless) computational model, which simplifies the definition of a formal basis for (functional) GT control structures. In this paper, we propose the *Graph Transformation control Algebra* (GTA) as such a foundation. The GTA has been used as the formal basis for implementing the control structures in the (functional) GT tool *GrapeVine*.

1 Introduction

There are diverse practical applications of graph transformations (GT) in software engineering, computer science and beyond [8]. They often require or benefit from *control structures* that can be used to restrict and direct the application of GTs. Heckel and Taentzer categorize four approaches for controlling the application of GTs, including non-terminals, dedicated control expressions, integrity constraints, and procedural abstraction [7]. Real-world applications often employ a combination of these approaches. Most existing GT tools follow a stateful model of computation where rule applications and control expressions mutate the program state. Multiple rule applications may need to be grouped into *atomic* units that are executed entirely or not at all and in isolation of other rule applications. Finding solutions with non-deterministic rule applications may require search mechanisms like backtracking and heuristic optimization. Implementing such sophisticated mechanisms is not trivial and their semantics is often not formalized, which impedes formal reasoning about the behaviour of non-deterministic GT programs.

It is well known that stateless computation provides for a simpler semantic model when compared to stateful computation. We recently presented a functional GT tool (called *GrapeVine*) that implements a stateless computational model with deterministic operators for rule application [21]. This paper presents the formal foundation for the control structures provided in this tool. We use an algebraic approach to define a set of operations referred to as the *Graph Transformation control Algebra* (GTA). In addition to providing a precise semantics for the control structures provided by *GrapeVine*, we hope that the GTA will enable future research on analysis and optimization of functional, deterministic GT programs, analogous to the use of the relational algebra for database query analysis and optimization.

The rest of this paper is structured as follows. We start with a brief introduction to graphs, graph transformations and graph constraints. Sec. 3 provides a description of related work on control structures for GT programs. We introduce the *Graph Transformation control Algebra* (GTA) and our notion of programmed graph transformation systems in Sec. 4. Sec. 5 provides a description of an implementation of the control structures in the tool *GrapeVine*, which are based on the GTA. Finally, we offer conclusions and discuss current work in Sec. 6.

2 Preliminaries

For the purpose of this paper, we limit ourselves to a definition of basic forms of graphs and graph transformations (GTs). While many GT tools (including *GrapeVine*) provide more advanced concepts (e.g., labels, attributes, typed graphs), the definitions below can easily be extended accordingly. The reader is referred to [3] for a more complete introduction to graph transformations.

Definition 1 (Graph)**.** *A graph is a tuple $G : (N,E,s,t)$ where N is a finite set of* nodes*, E is a finite set of* edges*, and $s,t : E \to N$ are total* source *and* target *functions, respectively.*

Definition 2 (Rule)**.** *A (GT)* rule *is defined as a pair of graphs (L,R) that share a common (possibly empty) interface graph $I = L \cap R$ containing all nodes and edges shared between in L and R. L is called the rule's* left-hand side *and R is called its* right-hand side*.*

Definition 3 (Transformation)**.** *An application of a rule $r : (L,R)$ to a given host graph G requires a* match *of the left-hand side in G, i.e., the existence of a graph morphism $L \xrightarrow{m} G$. The application deletes all elements from G that are matched to elements that only appear on the rule's left-hand side $(L-I)$, creates elements for any elements that appear only on the rule's right-hand side $(R-I)$, and embeds these new elements in the preserved context $(m(I))$. A more formal definition of a rule application based on category theory (double-pushout approach) is provided by Corradini et al. [2]. To avoid problems that may occur when deleting graph elements, a match has to satisfy the so-called* gluing condition*, which consists of two parts: (1) the embedding context (attached edges) of any deleted nodes must also be identified and deleted by the rule (*dangling condition*), and (2) deleted element must only have one pre-image in L (*identification condition*).*

The transformation of a given graph G into a graph G' by applying rule r at a valid match m is denoted as $G \overset{r,m}{\rightsquigarrow} G'$. Note that in contrast to the definition in category theory (which identifies graphs up to isomorphism), we assume a transformation to produce a unique graph.

As mentioned before, we opted against adding more advanced concepts to our formalization of rules and transformations, e.g., (negative) application conditions and rule parameters. While *GrapeVine* provides these concepts, they are not the focus of this paper. We do, however, introduce the notion of *graph constraints*,[11] as constraints are one of the mechanisms used to control the application of GT rules [7].

Definition 4 (Atomic Constraint)**.** *An atomic (graph) constraint $I \xrightarrow{c} T$ is defined as a graph monomorphism between two graphs I and T. A graph G satisfies c (denoted as $G \vDash c$) if for every monomorphism $I \xrightarrow{h} G$ there is a monomorphism $T \xrightarrow{f} G$ such that $h = f \circ c$.*

Definition 5 (Constraint)**.** *A (graph) constraint is inductively defined as either an atomic graph constraint $c : I \to T$, a negation of a graph constraint (i.e., if κ is a graph constraint, so is $\neg\kappa$), or the disjunction of two graph constraints (i.e., if κ_1 and κ_2 are graph constraints, so is $\kappa_1 \vee \kappa_2$).*

Now that we have introduced constraints, we extend our definition of graphs and transformations to constrained graphs and transformations on constraint graphs.

Definition 6 (Constrained Graph)**.** *A constrained graph is a tuple (G,K) where G is a graph and K is a finite set of graph constraints satisfied by G, i.e, $\forall \kappa \in K : G \vDash \kappa$.*

Definition 7 (Transformation of a constrained graph)**.** *A transformation of a constrained graph* $(G,K) \overset{r,m}{\rightsquigarrow} (G',K)$ *exists, if there exists a corresponding transformation for the unconstraint graph* $G \overset{r,m}{\rightsquigarrow} G'$*, where the resulting graph satisfies all constraints, i.e.,* $\forall \kappa \in K : G' \models \kappa$*.*

From here on, we only consider constrained graphs and transformations on constrained graphs. The reader should therefore assume that we refer to a "constrained graph" whenever we use the term *graph* without explicitly mentioning that we are talking about an unconstrained graph.

A transformation $G \overset{r,m}{\rightsquigarrow} G'$ is also called a *direct linear derivation*. In general, there are many possible matches for a rule in a given graph, i.e., multiple direct linear derivations are possible. We may just write $G \overset{r}{\rightsquigarrow} G'$ to denote *any* direct linear derivation, if we do not care about the match.

3 Related Work

Most GT tools offer some constructs to control the application of GT rules. The inherent complexity related to stateful computation and non-deterministic choice between possible derivations complicates the implementation of execution engines for GT programs. To the best of our knowledge, *PROGRES* [17], *Grape* [19], and *GP* [12] are the only GT tools that implement backtracking mechanisms, capable of reconstructing graphs at non-deterministic choice points. The PROGRES project has been discontinued since its platform is no longer available. *Grape* is still available on a modern platform but it is no longer under further development, in part due to the complexity of maintaining the implemented backtracking system. Work on *GP* appears to continue with the latest version of the language called *GP2*. The execution of *GP(2)* programs requires the York Abstract Machine (YAM) which handles a sophisticated data structure to keep track of non-determinism, choice points and environment frames.

PROGRES and *GP(2)* provide a formalized semantics for their control structures. The semantics definition for *PROGRES* is complex and spans over three hundred pages. The formalization of *GP(2)* is comparably simple. The original language (*GP*) consist of only four types of commands: non-deterministic rule application, sequential composition, branching, and iteration [12]. Its successor (*GP2*) provides additional constructs, including a second kind of branching statement and an explicit operator for non-deterministic choice between sub-programs [13]. Moreover, GP2 changed the semantics of control structures in order to allow for efficient implementation of branching and looping. In particular, failures in conditions of branching statements or loop bodies no longer enforce backtracking. While *GP(2)* programs theoretically associate all reachable output graphs to a given input graphs, the program computation may not succeed. This is because GP(2)'s non-deterministic rule application operators are not guaranteed to find these reachable output graphs and programs may diverge [12].

GROOVE is a tool that focuses on using verification and state-space exploration of GT systems [14]. Given a GT specification, the *GROOVE* simulator is capable of recursively generating (a possibly infinite) graph transition system (GTS). Rule applications can be controlled by specifying priorities. Moreover, *GROOVE* provides a simple textual control language, e.g., with operators for conditionals, loops and random choice. The tool offers different state space exploration strategies, including full exploration (depth-first or breadth-first) and partial exploration (linear, random linear, and condition). Partial exploration strategoes are useful in cases where the state space is too large. The tool has a mechanism to detect graph state collisions during exploration (up to graph isomorphism) [15]. That mechanism uses a hashing function to compute graph certificates for predicting isomorphic graphs efficiently and with high accuracy. The implementation of the GTA in our tool (*GrapeVine*) uses the same approach to detect graphs that are likely isomorphic.

AGG is another GT tool with a formally defined execution semantics [18, 16]. However, AGG avoids backtracking altogether and uses a random selection when making a non-deterministic choices during rule application and matching. *FUJABA* has control structures based on a combination of UML activity diagrams and GT rules [10]. The semantics has not been defined formally and there is no back-tracking. *GrGen* provides a textual language for controlling GT rule applications but also without a formal semantics and backtracking [5].

GReAT is a GT tool with dedicated support for model transformations [1]. It is notably different from other tools in that it support arbitrarily many input and output graphs. Its control structures use a combination of control-flow diagrams, structuring constructs and OCL-based model constraints.

4 Programmed Graph Transformation Systems with GTA

This section is divided in two parts. We begin by defining our notion of a programmed graph transformation system (GTS) and then discuss the rational behind the choices we made.

4.1 Syntax and Semantics

Definition 8 ((programmed) Graph Transformation System (GTS))**.** *We define a (programmed)* Graph Transformation System *(GTS) as a tuple* (R,C,P)*, where R is a set of* rules*, C is a set of* constraints*, and P is a set of graph* p*rograms.*

In this paper, we are mainly concerned with defining a formal basis for the programs P of a GTS. We define the *graph transformation control algebra* (GTA) for that purpose. GTA operators work on a common data type, a sequence of graph sets, referred to as a ***grap**h set **e**numeration* (or "*grape*" for short) in the rest of this paper. We define this data type below. Let $\mathbb{G}$ denote the domain of graphs.

Definition 9 (Graph set enumeration (grape))**.** *A* graph set enumeration *(or* grape *for short) is a finite, non-empty sequence* $\ddot{G} : \langle \bar{G}_1, \bar{G}_2, .., \bar{G}_n \rangle$ *where each* $\bar{G}_i$ *is a finite set of (possibly isomorphic) graphs, i.e.,* $\bar{G}_i \subset \mathbb{G}$*. Let* $\dddot{\mathbb{G}}$ *denote the domain (data type) of* grapes*. Furthermore, we define the constant* ☆ *to denote the grape with with a single element, containing a single, empty graph, i.e.,* $☆ = \langle\{((\varnothing,\varnothing,\varnothing,\varnothing),\varnothing)\}\rangle$.

The GTA has ten operators representing functions on *grapes*. We first introduce their syntax and then proceed to defining their semantics.

Definition 10 (Graph Program (syntax))**.** *Given a GTS* (R,C,P)*,* $c \in C$ *and* $r \in R$*, each program* $p \in P$ *is a GTA expression, which is defined as one of the following:*

- $\triangle(c)$ *and* ⨻(c) *are GTA expressions, called* constrain *and* unconstrain*, respectively ;*
- $\twoheadrightarrow(r)$ *is a GTA expressions, called* derive*;*
- $\circledcirc(n,\lesssim)$ *with* $n \in \mathbb{N}$ *and a total order* $\lesssim$ *on graphs is a GTA expression, called* select*;*
- ⸱⫶⸱(e_1,e_2) *and* $\div(e_1,e_2)$ *are GTA expressions, if* e_1 *and* e_2 *are GTA expressions; They are called* sequence *and* alternative*, respectively;*
- ↻ (e) *is a GTA expression called* loop *if e is a GTA expression;*
- ⥽ (c,e) *is a GTA expression called* search *if e is a GTA expression;*
-)(*and* $\not\equiv$ *are GTA expressions called* cut *and* distinct*, respectively.*

Definition 11 (Graph Program (semantics))**.** *A graph program p is interpreted as a function mapping* grapes *to* grapes*, i.e.,* $[\![p]\!] : \dddot{\mathbb{G}} \to \dddot{\mathbb{G}}$*. The program semantics is based on the interpretation of the individual GTA operators as functions* $\dddot{\mathbb{G}} \to \dddot{\mathbb{G}}$*. Most of them are interpreted as total functions, with the exception of* loop *and* search *(* ↻ , ⤼ *) which are not guaranteed to terminate.*

- **Constrain and Unconstrain (△ and ⨻)**
  The purpose of these functions is to declare and undeclare graph constraints, respectively. $\triangle(c)$ *declares constraint c on all the graphs in the last element of a given* grape *that satisfy c. All other graphs are removed from that element. Formally:* $[\![\triangle(c)]\!](\langle..,\bar{G}_n\rangle) = \langle..,\bar{G}'_n\rangle$ *with* $\bar{G}'_n = \{(G, K + \{c\}) \mid (G,K) \in \bar{G}_n \wedge G \vDash c\}$

  ⨻(c) *removes constraint c from the graphs in the last element of a grape:* $[\![$⨻$(c)]\!](\langle..,\bar{G}_n\rangle) = \langle..,\bar{G}'_n\rangle$ *with* $\bar{G}'_n = \{(G, K - \{c\}) \mid (G,K) \in \bar{G}_n\}$

  Please note that whenever we use the notation of $\langle..,\rangle$ *on two sides of a definition (like above), we require that ".." stands for the same sequence of elements. We use the notation "..a" and "..b" to refer to different sequences of elements.*

- **Derive (↠)**
  Function derive *computes all direct linear derivation of each graph in the last element of a given* grape *and extends the given input* grape *with an element that contains* all *resulting graphs, i.e.,* $[\![\twoheadrightarrow (r)]\!](\langle..,\bar{G}_n\rangle) = \langle..,\bar{G}_n,\bar{G}_{n+1}\rangle$ *where* $\bar{G}_{n+1} = \{G' \mid \exists G \in \bar{G}_n : G \overset{r}{\rightsquigarrow} G'\}$

- **Select (⊚)**
  Function Select *(*$\circledcirc(k,\lesssim)$*) reduces the last element of a given* grape *to at most k elements. The selection is determined by a total order on graphs* $\lesssim$*. Formally,* $[\![\circledcirc(k,\lesssim)]\!](\langle..,\bar{G}_n\rangle) = \langle..,\bar{G}'_n\rangle$*, with* $\bar{G}'_n \subseteq \bar{G}_n \wedge |\bar{G}'_n| \le k \wedge (|\bar{G}'_n| < k \Rightarrow \bar{G}'_n = \bar{G}_n) \wedge \nexists G \in \bar{G}_n - \bar{G}'_n, G' \in \bar{G}'_n : G' \lesssim G$

- **Sequence (·|·)**
  $\cdot|\cdot(a,b)$ *composes two GTA expressions sequentially by using functional composition, i.e.,* $[\![\cdot|\cdot(a,b)]\!] = [\![b]\!] \circ [\![a]\!]$.

- **Alternative (÷)**
  $\div(a,b)$ *composes two GTA expressions (a and b) as alternatives by extending a given* grape *with a new element that is the union of the last elements of the* grapes *produced by interpreting the two expressions, i.e.,* $[\![\div(a,b)]\!](\ddot{G} : \langle..x\rangle) = \langle..x, \bar{O}_1 \cup \bar{O}_2\rangle$ *with* $[\![a]\!](\ddot{G}) = \langle..y,\bar{O}_1\rangle$ *and* $[\![b]\!](\ddot{G}) = \langle..z,\bar{O}_2\rangle$*.*

- **Distinct** $\not\equiv$
  Graph exploration may produce identical graphs (up to isomorphism). The distinct *function ensures that the last element of the* grape *contains only unique graphs (up to isomorphism) when considering all graphs in the* grape*. If the last element of the* grape *contains several identical graphs, the smallest one is preserved (according to a given total order* $\lesssim$*). Formally,* $\not\equiv (\lesssim)$ *is defined as*
  $[\![\not\equiv (\lesssim)]\!](\langle\bar{G}_1,\ldots,\bar{G}_{n-1},\bar{G}_n\rangle) = (\langle\bar{G}_1,\ldots,\bar{G}_{n-1},\bar{G}'_n\rangle)$ *where* $\bar{G}'_n = \bar{G}_n \setminus \{D \mid (\exists i < n : \exists J \in \bar{G}_i : J \cong D) \vee (\exists D' \in \bar{G}_n : D' \lesssim D \wedge D' \cong D)\}$
  Note that we disregard any constraints attached to constrained graphs when comparing them for isomorphism, i.e., $(G_1,K_1) \cong (G_2,K_2) \Leftrightarrow G_1 \cong G_2$ *for any two constrained graphs* (G_1,K_1) *and* (G_2,K_2)*.*

- **Cut (**)(**)**
  The cut *operator "cuts off" all but the last element of a given grape. This is useful to restrict the*

distinct's operators ability to "look back" in the derivation history to find identical graphs. cut *is defined as* $[\![\,)(\,]\!](\langle .., \bar{G}_n \rangle) = \langle \bar{G}_n \rangle$.

- **Loop ($\circlearrowleft$)**
  $\circlearrowleft(e)$ *is interpreted as a function that recursively interprets GTA expression e on the most recently computed* grape *while the last element is not empty, i.e,*

$$[\![\circlearrowleft(e)]\!](\dddot{G}) = \begin{cases} \dddot{G}, & \mathit{if} [\![e]\!](\dddot{G}) = \langle .., \varnothing \rangle \\ [\![\circlearrowleft(e)]\!] \circ [\![e]\!](\dddot{G}) & \mathit{otherwise} \end{cases}$$

- **Search ($\looparrowright$)**
  $\looparrowright(c,o)$ *is interpreted as a (recursive) function that repeatedly interprets a GTA expression o on the most recently computed* grape *while none of the graphs in the last element of the current* grape *satisfy constraint c and the last element is not empty, i.e,*

$$[\![\looparrowright(c,o)]\!](\dddot{G} : \langle .., \bar{G}_n \rangle) = \begin{cases} \dddot{G}, & \mathit{if} \bar{G}_n = \varnothing \vee \exists G \in \bar{G}_n : G \vDash c \\ [\![\looparrowright(c,o)]\!] \circ [\![o]\!](\dddot{G}) & \mathit{otherwise} \end{cases}$$

4.2 Properties of the GTA and Rationale

Habel and Plump showed that any graph programming language capable of (nondeterministic) rule application, sequential composition, and iteration is computationally complete [6]. They also showed that this set of operators is minimal. While Habel and Plump's rule application operator ($\Rightarrow_R$) combines rule selection (from a set of alternatives R) and rule matching/derivation, the GTA presented in this paper has separate operations for choosing alternatives ($\div$) and matching/derivation ($\twoheadrightarrow$). However, it is easy to see that the behaviour of the $\Rightarrow_R$ operator can be achieved by combining the two GTA operators. The fact that these GTA operators are deterministic means that they can produce all possible outputs of the non-deterministic operator. Since the GTA also has operators for sequential composition ($\cdot|\cdot$) and iteration ($\circlearrowleft$), the set of operators $\{\twoheadrightarrow, \div, \cdot|\cdot, \circlearrowleft\}$ is computationally complete (and minimal).

To reduce the "state-space explosion" problem of breadth-first exploration, the GTA has operators for reducing the search space. In particular, the *Distinct* operator ($\not\equiv$) was introduced to detect and remove equivalent graph states (up to isomorphism). This operator is the main reason to consider *grapes* rather than simple graph sets as the common data type for computation. The *Select* operator ($\circledcirc$) provides a way to limit the search to an upper number of ranked graphs to explore (based on a given total order). The order relation used by the *Select* operator may be defined by the user according to the specifics of the graph program application. For example, it may be based on the notion of a graph-edit distance (GED) that measures the distance of the computed graphs from an objective target graph [4]. While the general problem of computing GEDs is NP-hard, algorithms for efficient approximations exist. For similar reasons (i.e., to restrict the search space), we included graph constraints in addition to GT rules. *Constrain* ($\triangle$) and *Unconstrain* ($\xcancel{\triangle}$) expressions provide programs with ways to (temporarily) restrict the exploration of possible computation.

5 Implementing the GTA within *GrapeVine*

An implementation of GTA-based graph programs requires a functional GT tool in which graphs are immutable first-class objects rather than treating *the* graph as "singleton" global variable for stateful manipulation. We have recently proposed *GrapeVine* as a tool to meet this criterion [21]. We start this section with a short introduction to this tool, before discussing graph programs in *GrapeVine*.

5.1 *GrapeVine*: A Short Overview

GrapeVine is implemented on top of the Neo4J graph database and derives much of its scalability from that architecture. The tool is a fundamentally new revision of the earlier tools *Grape* [19] and *Grape Press* [20]. Compared to these earlier tools, the main novelty in *GrapeVine* is its functional computational model, where graphs are considered immutable objects and programs can be deterministic.

GrapeVine uses a textual language for defining GT rules, constraints and programs. That language is provided as an internal DSL (domain-specific language) to the general-purpose programming language Clojure. *GrapeVine* has been integrated with *Gorilla*[1]. Gorilla is a browser-based computational notebook for Clojure, similar to other computational notebook technologies, like the well-known Jupyter notebooks. Using this platform, *GrapeVine* supports visualization of graphs, rules, and constraints (among other things) [20]. *GrapeVine* can also be used independently of the computational notebook UI; *GrapeVine* programs are regular Clojure programs, which run on the Java Virtual Machine (JVM) and can thus integrate with any other JVM language, such as Java, Kotlin, Scala, Groovy, etc.

5.2 *GrapeVine* Control Structures

When designing the control structures for *GrapeVine* we used the constructs provided by the functional host language (Clojure) as much as possible and avoided the creation of unnecessary "syntactic sugar". For example, rule applications (derivations) can simply be called by the name of the rule, i.e., declaring a rule creates rule application functions with that name. Similarly, declaring a constraint automatically creates a pair of functions to check that constraint (positively and negatively).

Tab. 1 provides a correspondence between the syntax used in *GrapeVine* graph programs and their interpretation in the form of GTA expressions. As mentioned above, a rule application is simply a call to a function with the rule's name. Similarly, constraint checks are done by calling a constraint by name (with an added minus sign for negation). Constraints are added to graphs or removed from graphs using the `schema/schema-drop` functions. *GrapeVine* uses the regular Clojure threading macro (`->`) for sequential composition and introduce a new macro (——) for composing alternatives. In addition to a simple "While possible" loop, *GrapeVine* has two variations of an "Until" loop. The behaviour of these variants is similar but the `->?+` operator ("Distinct Until") ensured that the computed graphs are unique.

Finally, *GrapeVine* provides functions `cut`, `newgrape`, `dist`, and `select`, which directly correspond to the GTA operations)(, ☆, ≢, and ◎.

Note that some *GrapeVine* control structures are variadic, i.e., they allow an arbitrary number of parameters. This is indicated by the ellipses in middle column of Tab. 1. Variadic parameter lists are interpreted in the obvious way for adding/dropping multiple graph constraints and creating sequences/alternatives over multiple operations. In the case of loops, longer parameter lists are interpreted as sequential composition blocks of operations that are performed at each iteration.

5.3 *GrapeVine* Control Structures: A Simple Example

Let us look at a simple example of a graph program in *GrapeVine* to get an impression. We use the example of the well-known ferryman problem [22]. In that problem, a ferryman needs to (safely) trans-

[1]http://gorilla-repl.org

Description	Concrete Syntax (*GrapeVine*)	Interpretation (GTA)
Application of rule *r*	`r`	$\twoheadrightarrow(r)$

Addition of constraint *c*
(negated)
	`schema c ..`
	`schema c- ..`
		$\triangle(c)$
		$\triangle(\neg c)$

Removal of constraint *c*
(negated)
	`schema-drop c ..`
	`schema-drop c- ..`
		$\xcancel{\triangle}(c)$
		$\xcancel{\triangle}(\neg c)$

Check of constraint *c* (negated)	`c` `c-`	$\cdot|\cdot(\triangle(c),\xcancel{\triangle}(c))$ $\cdot|\cdot(\triangle(\neg c),\xcancel{\triangle}(\neg c))$
Sequence	`-> e_1 e_2 ..`	$\cdot|\cdot(e_1,\cdot|\cdot(e_2,..))$
Alternative	`|| e_1 e_2 ..`	$\div(e_1,\div(e_2,..))$
Loop (while possible)	`->* e ..`	$\circlearrowleft(\cdot|\cdot(e,..))$
Until	`->?* c e ..`	$\looparrowright(c,\cdot|\cdot(e,..))$
Distinct Until	`->?+ c e ..`	$\looparrowright(c,\cdot|\cdot(\cdot|\cdot(e,..),\not\equiv))\circ)($
Cut	`cut`	$)($
A *grape* with a single element containing an empty graph	`newgrape`	☆
Distinct	`dist`	$\not\equiv$
Select	`select k v`	$\circledcirc(k,v)$

Table 1: Concrete syntax of *GrapeVine* control structures and their GTA interpretations

port three things (a goat, a wolf, and a cabbage) from one side of a river to the other side. He can only transport one thing at a time. The wolf will eat the goat and the goat will eat the cabbage, respectively, if left unattended with their food.

Fig.1 shows three rules and three constraints modelled within *GrapeVine* in support of solving the ferryman problem. (For simplicity, we omit the textual definition of these rules and only show their generated graphical view. Also, please note that for artistic freedom, the cabbage was exchanged by a grape.) Rule *setup-ferryman* creates the initial graph with the two river sides and all items on one side. Created graph elements are shown in green. The other two rules *ferry_one_over* and *cross_empty* specify the two possible actions, namely to cross with or without an item. Deleted graph elements are red.

The shaded isomorphism ("`ISO`") box in the two rules specify that these rules must use *isomorphic* subgraph matching. (By default, *GrapeVine* does not enforce isomorphic matches, i.e., separate elements on a rule's left-hand side do not need to match to separate elements in the host graph.)

The bottom half of Fig.1 shows three (atomic) graph constraints. In *GrapeVine*, basic graph constraints $I \xrightarrow{c} T$ are visualized by using black colour for elements in I ("if" pattern) and blue colour for elements in T ("then" pattern). In the case of this example, the three graph constraints have the form $\varnothing \xrightarrow{c} T$, i.e., they represent simple existence constraints (of form $\exists T$). Such constraints are also called *basic constraints* [11]. Note that pattern matching for constraints always uses isomorphic matching.

To better distinguish between the names of rules and constraints in graph programs, it is a convention in *GrapeVine* to append an exclamation mark to the names of constraints.

Fig.2 shows a *GrapeVine* program for solving the ferryman problem. The program should be understandable based on the above definition of control structures and the definition of the rules and constraints. The call to `newgrape` creates a new *grape* with an empty graph. After setting up the initial state (applying rule `setup-ferryman`), we enter a loop that recurs until one of the graphs in the last element of the current *grape* satisfies the constraint `all_on_the_other_side!`.

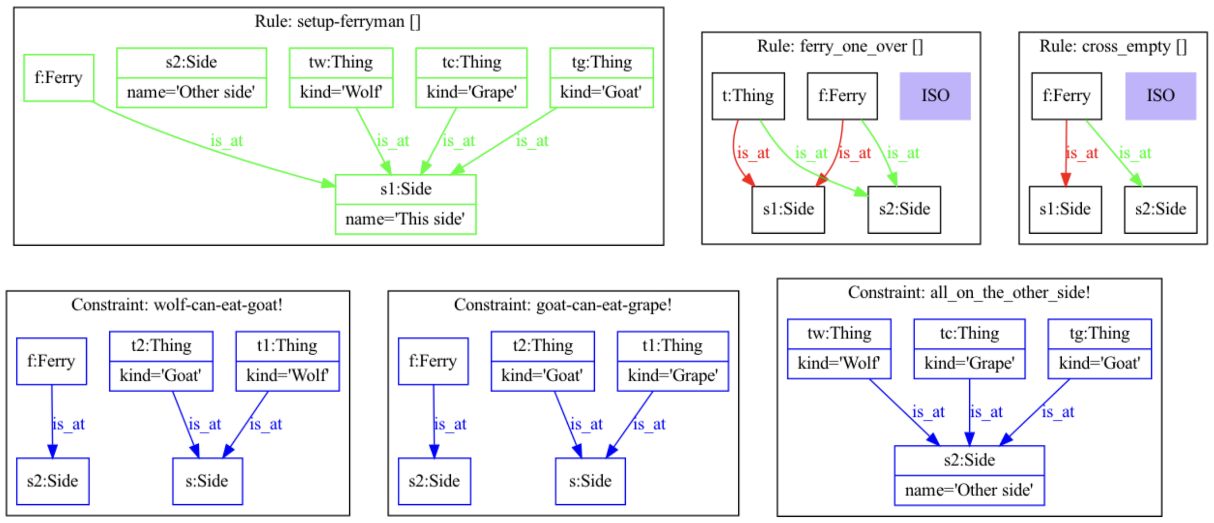


Figure 1: Ferryman problem (rules and constraints)

Note that the program in Fig. 2 is guaranteed to find a solution if there exists one. If we had used non-deterministic rule application and a depth-first exploration strategy, the execution mechanism may or may not find a solution, depending on how it was implemented. A mechanism that resolves non-deterministic choices by a fair process (e.g., by using true randomness) will eventually find a solution, but without bounded time.

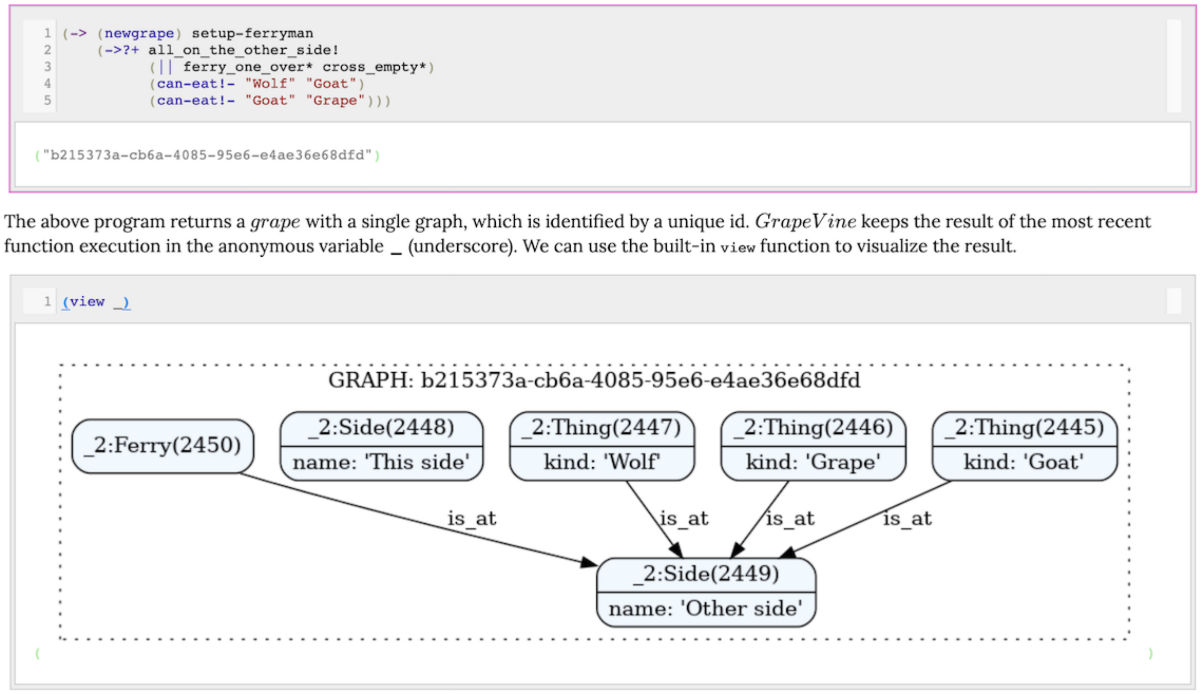
```
(-> (newgrape) setup-ferryman
    (->?+ all_on_the_other_side!
          (|| ferry_one_over cross_empty)
          wolf-can-eat-goat!-
          goat-can-eat-grape!-)))
```

Figure 2: A program to solve the ferryman problem

Fig. 3 shows an alternative, yet equivalent program for solving the ferryman problem. This program declares the two constraints we checked manually in Fig. 2 as invariant constraints on the graphs in the *grape*. Once declared, *GrapeVine* will ensure that only valid derivations are possible.

```
(-> (newgrape)
    (schema wolf-can-eat-goat!- goat-can-eat-grape!-)
    setup-ferryman
    (->?+ all_on_the_other_side
          (|| ferry_one_over cross_empty)))
```

Figure 3: An alternative program, using schema constraints



```
(-> (newgrape) setup-ferryman
    (->?+ all_on_the_other_side!
          (|| ferry_one_over* cross_empty*)
          (can-eat!- "Wolf" "Goat")
          (can-eat!- "Goat" "Grape")))
```

```
("b215373a-cb6a-4085-95e6-e4ae36e68dfd")
```

The above program returns a *grape* with a single graph, which is identified by a unique id. *GrapeVine* keeps the result of the most recent function execution in the anonymous variable _ (underscore). We can use the built-in `view` function to visualize the result.

```
(view _)
```

Figure 4: Sample execution of Ferryman program

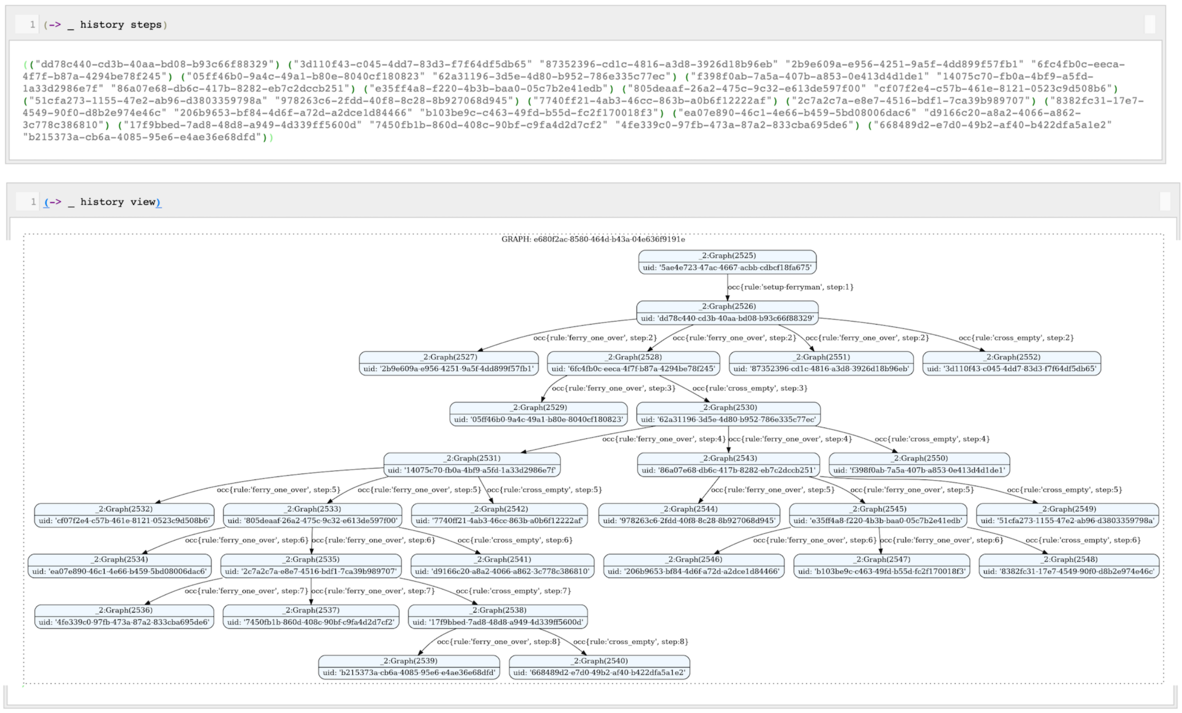
5.4 Example Execution

Fig. 4 shows a screen capture of an example execution of the Ferryman program, using *GrapeVine's* computational notebook UI. The figure shows two executable segments (boxes) where the shaded part contains executable code and the white part shows the output of the execution. The upper code segment shows that executing our Ferryman program returns a collection with a single element. (Like all Lisp-like languages, Clojure uses parentheses to indicate lists.) The returned list represents the last element of the *grape* that is returned by calling the functional graph program. Each graph is referred to by a unique ID. In this case, there is only one single graph.

Graphs can be visualized in the computational notebook using the notebook's *view* function. This is shown in the second executable segment in Fig. 4. Note that for convenience, *GrapeVine* provides a special variable "_" (underscore) to refer to the most recently computed *grape*.

The reason for returning merely the last element of a *grape* rather than the entire (history) sequence is for performance reasons and to avoid representational clutter in the computational notebook. Most of the times, we are only interested in the last element of a grape when computing with functional graph transformations. However, the full grape (history) is kept internally and can be returned when needed. This is demonstrated in Fig. 5, where the first code block shows the use of the built-in functions `history` and `steps` to output the entire *grape* for our example computation (which is represented as a list of lists of graph IDs in Clojure).

The textual output of a full *grape* history is not easy to read. *GrapeVine's* computational notebook UI has functions to visualize the history of a *grape* by means of a graph of graphs, where each edge marks an occurrence of a rule application. This is demonstrated in the second executable segment in Fig. 5, where each "row" of the shown "history graph" represents a new element in the *grape* sequence.



```
(-> _ history steps)
```

```
(("dd78c440-cd3b-40aa-bd08-b93c66f88329") ("3d110f43-c045-4dd7-83d3-f7f64df5db65" "87352396-cd1c-4816-a3d8-3926d18b96eb" "2b9e609a-e956-4251-9a5f-4dd899f57fb1" "6fc4fb0c-eeca-4f7f-b87a-4294be78f245") ("05ff46b0-9a4c-49a1-b80e-8040cf180823" "62a31196-3d5e-4d80-b952-786e335c77ec") ("f398f0ab-7a5a-407b-a853-0e413d4d1de1" "14075c70-fb0a-4bf9-a5fd-1a33d2986e7f" "86a07e68-db6c-417b-8282-eb7c2dccb251") ("e35ff4a8-f220-4b3b-baa0-05c7b2e41edb") ("805deaaf-26a2-475c-9c32-e613de597f00" "cf07f2e4-c57b-461e-8121-0523c9d508b6") ("51cfa273-1155-47e2-ab96-d3803359798a" "978263c6-2fdd-40f8-8c28-8b927068d945") ("7740ff21-4ab3-46cc-863b-a0b6f12222af") ("2c7a2c7a-e8e7-4516-bdf1-7ca39b989707") ("8382fc31-17e7-4549-90f0-d8b2e974e46c" "206b9653-bf84-4d6f-a72d-a2dce1d84466" "b103be9c-c463-49fd-b55d-fc2f170018f3") ("ea07e890-46c1-4e66-b459-5bd08006dac6" "d9166c20-a8a2-4066-a862-3c778c386810") ("17f9bbed-7ad8-48d8-a949-4d339ff5600d" "7450fb1b-860d-408c-90bf-c9fa4d2d7cf2" "4fe339c0-97fb-473a-87a2-833cba695de6") ("668489d2-e7d0-49b2-af40-b422dfa5a1e2" "b215373a-cb6a-4085-95e6-e4ae36e68dfd"))
```

```
(-> _ history view)
```

GRAPH: e680f2ac-8580-464d-b43a-04e636f9191e
_2:Graph(2525) uid: '5ae4e723-47ac-4667-acbb-cdbcf18fa675'
occ{rule:'setup-ferryman', step:1}
_2:Graph(2526) uid: 'dd78c440-cd3b-40aa-bd08-b93c66f88329'
occ{rule:'ferry_one_over', step:2}
occ{rule:'ferry_one_over', step:2}
occ{rule:'ferry_one_over', step:2}
occ{rule:'cross_empty', step:2}
_2:Graph(2527) uid: '2b9e609a-e956-4251-9a5f-4dd899f57fb1'
_2:Graph(2528) uid: '6fc4fb0c-eeca-4f7f-b87a-4294be78f245'
_2:Graph(2551) uid: '87352396-cd1c-4816-a3d8-3926d18b96eb'
_2:Graph(2552) uid: '3d110f43-c045-4dd7-83d3-f7f64df5db65'
occ{rule:'ferry_one_over', step:3}
occ{rule:'cross_empty', step:3}
_2:Graph(2529) uid: '05ff46b0-9a4c-49a1-b80e-8040cf180823'
_2:Graph(2530) uid: '62a31196-3d5e-4d80-b952-786e335c77ec'
occ{rule:'ferry_one_over', step:4}
occ{rule:'ferry_one_over', step:4}
occ{rule:'cross_empty', step:4}
_2:Graph(2531) uid: '14075c70-fb0a-4bf9-a5fd-1a33d2986e7f'
_2:Graph(2543) uid: '86a07e68-db6c-417b-8282-eb7c2dccb251'
_2:Graph(2550) uid: 'f398f0ab-7a5a-407b-a853-0e413d4d1de1'
occ{rule:'ferry_one_over', step:5}
occ{rule:'ferry_one_over', step:5}
occ{rule:'cross_empty', step:5}
occ{rule:'ferry_one_over', step:5}
occ{rule:'ferry_one_over', step:5}
occ{rule:'cross_empty', step:5}
_2:Graph(2532) uid: 'cf07f2e4-c57b-461e-8121-0523c9d508b6'
_2:Graph(2533) uid: '805deaaf-26a2-475c-9c32-e613de597f00'
_2:Graph(2542) uid: '7740ff21-4ab3-46cc-863b-a0b6f12222af'
_2:Graph(2544) uid: '978263c6-2fdd-40f8-8c28-8b927068d945'
_2:Graph(2545) uid: 'e35ff4a8-f220-4b3b-baa0-05c7b2e41edb'
_2:Graph(2549) uid: '51cfa273-1155-47e2-ab96-d3803359798a'
occ{rule:'ferry_one_over', step:6}
occ{rule:'ferry_one_over', step:6}
occ{rule:'cross_empty', step:6}
occ{rule:'ferry_one_over', step:6}
occ{rule:'ferry_one_over', step:6}
occ{rule:'cross_empty', step:6}
_2:Graph(2534) uid: 'ea07e890-46c1-4e66-b459-5bd08006dac6'
_2:Graph(2535) uid: '2c7a2c7a-e8e7-4516-bdf1-7ca39b989707'
_2:Graph(2541) uid: 'd9166c20-a8a2-4066-a862-3c778c386810'
_2:Graph(2546) uid: '206b9653-bf84-4d6f-a72d-a2dce1d84466'
_2:Graph(2547) uid: 'b103be9c-c463-49fd-b55d-fc2f170018f3'
_2:Graph(2548) uid: '8382fc31-17e7-4549-90f0-d8b2e974e46c'
occ{rule:'ferry_one_over', step:7}
occ{rule:'ferry_one_over', step:7}
occ{rule:'cross_empty', step:7}
_2:Graph(2536) uid: '4fe339c0-97fb-473a-87a2-833cba695de6'
_2:Graph(2537) uid: '7450fb1b-860d-408c-90bf-c9fa4d2d7cf2'
_2:Graph(2538) uid: '17f9bbed-7ad8-48d8-a949-4d339ff5600d'
occ{rule:'ferry_one_over', step:8}
occ{rule:'cross_empty', step:8}
_2:Graph(2539) uid: 'b215373a-cb6a-4085-95e6-e4ae36e68dfd'
_2:Graph(2540) uid: '668489d2-e7d0-49b2-af40-b422dfa5a1e2'

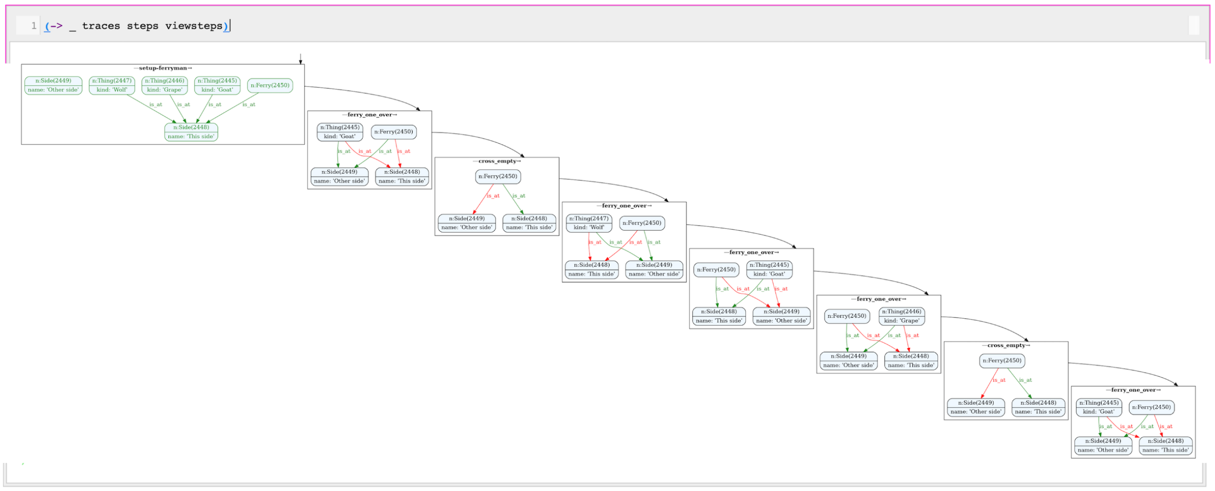
Figure 5: Visualizing the entire *grape* (history) of the example execution

GrapeVine also provides ways to visualize the details of each rule occurrence. Fig. 6 demonstrates the use of the built-in `traces` function to visualize the details of rule occurrences in the history of computing the result of our Ferryman solution.

5.5 Efficiency Considerations

An important factor for making the above described approach to GT programming feasible is the ability to avoid exploring graph states that have been explored before (by means of the $\not\equiv$ operator). This operator requires that previous versions of graphs are efficiently accessible (and comparable). Moreover, parallel exploration of possible choices generate non-linear version histories of graphs that must be maintained efficiently. *GrapeVine* uses a fully-persistent data structure for maintaining graphs in a graph database (Neo4J). Details on that persistent data structure have been reported in [21] and interested readers are referred to that publication. However, one new aspect that we want to comment on is the implementation of the $\not\equiv$ operator. While graph isomorphism checks are expensive, Rensink has demonstrated that hashed graph "certificates" can be used to speed up similarity checking in practice based on the idea of bisimulation [15]. *GrapeVine* uses this approach for comparing graphs. Moreover, since graphs are immutable, their hash certificates are computed only once, at creation time and then stored in the database using an efficient indexed search structure. This means that the run-time for similarity checks only grows logarithmically with the number of graphs in the *grape*, given the search and insertion complexity of b+ trees.

When compared to some other GT tools, specifically those that perform all computation in main memory, *GrapeVine* has a much higher latency when performing graph transformations. This is due to its client-server architecture between the Clojure (JVM) client and the database server. These two com-



```
(-> _ traces steps viewsteps)
```

Figure 6: Visualizing the detailed steps for each rule occurrence in an execution trace

ponents communicate using a Web-service API and each rule application requires multiple interactions (e.g., for rule matching, checking of application conditions, rule application, checking of graph constraints). However, *GrapeVine's* performance is scalable with respect to the number and size of graphs. While a comprehensive performance analysis is out of the scope of this paper, we have conducted a preliminary experiment with the program in Fig. 1. The platform was a 2017 iMac Pro with a 3Ghz Intel Xeon processor running both the database server (Neo4J) and *GrapeVine* in separate Docker containers connected by a bridged network.

Running the Ferryman program in Fig. 1 on an empty graph database takes approximately 7 seconds and creates 27 graphs in the process. The execution time of the program remains the same (approximately 7 seconds) even after running the program 1000 times and creating 27,000 graphs in the database. (Note: *GrapeVine* does not automatically "garbage collect" graphs, but the user can trigger a garbage collector manually [21].)

If we modify the program in Fig. 1 to use the "Until" operator without the distinction check (`->?*` operator instead of `->?+`), the program creates 216 graphs and takes 52 seconds.

We can compare these measurements with *GrapeVine's* predecessor (*Grape*), which implements a backtracking-based depth-first exploration using non-deterministic GT operators. This comparison makes sense, since it has the same Web-service based architecture (JVM client and Neo4J database server.) Client-server interactions therefore have similar latency. *Grape* is actually not able to compute a solution for the ferryman problem as stated in Fig. 1. This is because its execution engine is deterministic and would always explore non-deterministic alternatives between applicable rules in the order they are stated in the program. In the case of our program in Fig. 1, this would result in a diverging program run that would take the goat back and forth forever. Note that this problem would not go away if the order of the "alternative" statement was swapped. In that case, the goat would be moved and the ferryman would travel empty forever. [19] provides a modified program that uses bounded depth-first search (by giving the ferryman a maximum budget of seven moves). That program takes approximately 50 seconds to find a solution on the same machine as above. Of course, we note that *Grape* does not check for collisions when exploring solutions depth-first. Doing so would certainly increase efficiency. However, the comparison shows that the deterministic (breadth-first) operators in *GrapeVine* have a performance that is competitive with traditional (depth-first, backtracking-based) solutions to resolve non-determinism.

6 Conclusions and Current Work

While graph transformations have a well-defined semantics, the control structures used for programming with GTs are often either of limited expressiveness (but formalized) or they are expressive but lack a formal semantics. Implementing non-deterministic control structures in GT tools that use stateful computation requires complex and often inefficient processing such as backtracking on graph exploration. Unbounded depth-first search strategies may not find solutions that are theoretically computable. For example, a solution for the above-mentioned ferryman problem may not be found in GT tools that implement non-deterministic rule application based on depth-first search and backtracking unless repeated exploration of the same graph states are avoided or randomized exploration strategies are used.

Functional GT tools avoid many of the complexities of stateful GT tools, since graphs are immutable (i.e., derivations do not destroy/modify the input graph) and rule application can be deterministic. In this paper, we have defined a formal foundation for control structures used for functional GT, called the *Graph Transformation control Algebra* (GTA). GTA-based programs can be written as deterministic functions, i.e., they guarantee that all graphs that can be derived via a particular program can be returned. To make this feasible, GTA operators provide means to limit the exploration of the search space, for example by detecting previously explored graphs, checking graph constraints and selecting graphs based on a total ordering heuristic.

We have shown the implementation of GTA-based control structures within the functional GT tool *GrapeVine*. Our current work is on performing a more thorough performance evaluation with larger application problems and community benchmarks to provide further evidence for the feasibility of the proposed approach. For an experience report of using *GrapeVine* for one such larger problem, we refer the readers to a thesis by Machowczyk, who applied the approach to an application of graph rewriting to graph neural networks at the University of Leicester [9].

References

[1] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi & Attila Vizhanyo (2006): *The design of a language for model transformations*. *Software & Systems Modeling* 5(3), pp. 261–288, doi:10.1007/s10270-006-0027-7.

[2] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel & M. Löwe (1997): *Algebraic Approaches to Graph Transformations – Part I: Basic Concepts and Double Pushout Approach*. In G. Rozenberg, editor: *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, pp. 163–245, doi:10.1142/9789812384720_0003.

[3] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, doi:10.1007/3-540-31188-2.

[4] Xinbo Gao, Bing Xiao, Dacheng Tao & Xuelong Li (2010): *A Survey of Graph Edit Distance*. *Pattern Anal. Appl.* 13(1), p. 113–129, doi:10.1007/s10044-008-0141-y.

[5] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack & Adam Szalkowski (2006): *GrGen: A Fast SPO-Based Graph Rewriting Tool*. In: *Graph Transformations - Third International Conference, ICGT 2006, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, *LNCS* 4178, Springer, Berlin, pp. 383–397, doi:10.1007/11841883_27.

[6] Annegret Habel & Detlef Plump (2001): *Computational Completeness of Programming Languages Based on Graph Transformation*. In Furio Honsell & Marino Miculan, editors: *Foundations of Software Science and*

Computation Structures, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 230–245, doi:10.1007/3-540-45315-6_15.

[7] Reiko Heckel & Gabriele Taentzer (2020): *Beyond Individual Rules: Usage Scenarios and Control Structures*. In: *Graph Transformation for Software Engineers*, Springer International Publishing, pp. 67–85, doi:10.1007/978-3-030-43916-3_3.

[8] Reiko Heckel & Gabriele Taentzer (2020): *Graph Transformation for Software Engineers*. Springer International Publishing, doi:10.1007/978-3-030-43916-3.

[9] Adam Machowczyk (2022): *Graph Rewriting for Graph Neural Networks, CO7201 Individual Project, Department of Informatics, University of Leicester*. Technical Report, University of Leicester.

[10] Ulrich Nickel, Jörg Niere & Albert Zündorf (2000): *The FUJABA environment*. In: *Proceedings of the 22nd international conference on Software engineering - ICSE '00*, ACM Press, doi:10.1145/337180.337620.

[11] Fernando Orejas, Hartmut Ehrig & Ulrike Prange (2008): *A Logic of Graph Constraints*. In: *Fundamental Approaches to Software Engineering*, *LNCS* 4961, Springer Berlin Heidelberg, pp. 179–198, doi:10.1007/978-3-540-78743-3_14.

[12] Detlef Plump (2009): *The Graph Programming Language GP*. In: *Algebraic Informatics*, *LNCS* 5725, Springer Berlin Heidelberg, pp. 99–122, doi:10.1007/978-3-642-03564-7_6.

[13] Detlef Plump (2012): *The Design of GP 2*. *EPTCS 82, 2012, pp. 1-16*, doi:10.4204/EPTCS.82.1. arXiv:1204.5541.

[14] Arend Rensink (2004): *The GROOVE Simulator: A Tool for State Space Generation*. In John L. Pfaltz, Manfred Nagl & Boris Böhlen, editors: *Applications of Graph Transformations with Industrial Relevance*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 479–485, doi:10.1007/978-3-540-25959-6_40.

[15] Arend Rensink (2007): *Isomorphism checking in GROOVE*. *Electronic Communications of the EASST* 1, doi:10.14279/tuj.eceasst.1.77.

[16] Olga Runge, Claudia Ermel & Gabriele Taentzer (2012): *AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations*. In: *Applications of Graph Transformations with Industrial Relevance*, *LNCS* 7233, Springer Berlin Heidelberg, pp. 81–88, doi:10.1007/978-3-642-34176-2_8.

[17] Andy Schürr, Andreas J Winter & Albert Zündorf (1999): *The PROGRES approach: Language and environment*. In: *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools*, World Scientific, pp. 487–550, doi:10.1142/9789812815149_0013.

[18] Gabriele Taentzer (2004): *AGG: A Graph Transformation Environment for Modeling and Validation of Software*. In: *Applications of Graph Transformations with Industrial Relevance*, *LNCS* 3062, Springer Berlin Heidelberg, pp. 446–453, doi:10.1007/978-3-540-25959-6_35.

[19] Jens H. Weber (2017): *GRAPE – A Graph Rewriting and Persistence Engine*. In: *Graph Transformation*, *LNCS* 10373, Springer International Publishing, pp. 209–220, doi:10.1007/978-3-319-61470-0_13.

[20] Jens H. Weber (2021): *GrapePress - A Computational Notebook for Graph Transformations*. In: *Graph Transformation*, *LNCS* 12741, Springer International Publishing, pp. 294–302, doi:10.1007/978-3-030-78946-6_16.

[21] Jens H. Weber (2022): *Tool support for Fully-Persistent Graph Rewriting - GrapeVine*. In: *Graph Transformation*, *LNCS* 13349, Springer International Publishing, pp. 195–206, doi:10.1007/978-3-031-09843-7_11.

[22] Albert Zündorf & Andy Schürr (1992): *Nondeterministic control structures for graph rewriting systems*. In: *Graph-Theoretic Concepts in Computer Science*, *LNCS* 570, Springer Berlin Heidelberg, pp. 48–62, doi:10.1007/3-540-55121-2_5.