Reasoning about Graph Programs

Detlef Plump
The University of York, United Kingdom

GP 2 is a non-deterministic programming language for computing by graph transformation. One of the design goals for GP 2 is syntactic and semantic simplicity, to facilitate formal reasoning about programs. In this paper, we demonstrate with four case studies how programmers can prove termination and partial correctness of their solutions. We argue that GP 2's graph transformation rules, together with induction over the length of program executions, provide a convenient framework for program verification.

1 Introduction

The use of graphs to model dynamic structures is ubiquitous in computer science: application areas include compiler construction, pointer programming, model-driven software development, and natural language processing. The behaviour of systems in such domains can be captured by graph transformation rules specifying small state changes. Current languages based on graph transformation rules include AGG [15], GReAT [1], GROOVE [9], GrGen.Net [10] and PORGY [8]. This paper focusses on the graph programming language GP 2 [12] which aims to support formal reasoning about programs.[1]

A rigorous Hoare-logic approach to verifying programs in GP 1 (the predecessor of GP 2) is described in [13]. However, this calculus is restricted to programs in which loop bodies and guards of branching statements are sets of rules rather than arbitrary subprograms. Moreover, the assertions of [13] are first-order formulas and hence cannot express non-local properties such as connectedness or the absence of cycles. (Such properties can be expressed with the monadic second-order assertions of [14]. That paper's framework is currently extended to GP 2.)

In this paper, we take a more relaxed view on program verification and express specifications and proofs in ordinary mathematical language. Besides lifting the mentioned restrictions, this approach allows programmers to formulate invariants and induction proofs succinctly, without getting stuck in formal details. The possible reduction in rigor need not be a drawback if the liberal approach precedes and complements rigorous verification in a formal calculus such as Hoare-logic.

In Section 3 to Section 6, we verify four simple GP 2 programs, for generating the transitive closure of a graph, computing a vertex colouring, and checking graph-theoretic properties. In all case studies, we prove termination and partial correctness of the program in question. It turns out that graph transformation rules together with induction on derivations provide a convenient formalism for reasoning.

2 The Language GP 2

We briefly introduce GP 2, by describing some selected features and showing an example of a graph transformation rule. The original definition of GP 2, including a formal operational semantics, is given in [12]; an updated version can be found in [2]. There are currently two implementations of GP 2, a compiler generating C code [4] and an interpreter for exploring the language's non-determinism [3].

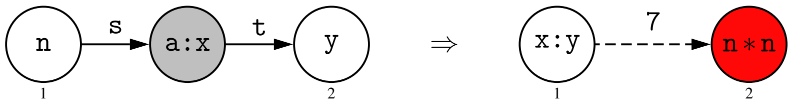
[1]GP stands for *graph programs*.

The principal programming constructs in GP 2 are conditional graph-transformation rules labelled with expressions. Rules operate on *host graphs* whose nodes and edges are labelled with lists of integers and character strings. Variables in rules are of type `int`, `char`, `string`, `atom` or `list`, where `atom` is the union of `int` and `string`. Atoms are considered as lists of length one, hence integers and strings are also lists. Given lists `x` and `y`, their concatenation is written `x:y`.

Besides a list, labels may contain a *mark* which is one of the values `red`, `green`, `blue`, `grey` and `dashed` (where `grey` and `dashed` are reserved for nodes and edges, respectively). Marks may be used to highlight items in input or output graphs, or to record which items have already been visited during a graph traversal. In this paper, we assume that programs are applied to input graphs without any marks. This allows to formulate succinct correctness claims in the case studies below.

Figure 1 shows an example of a rule which replaces the grey node and its incident edges with a dashed edge labelled with the integer 7. In addition, nodes 1 and 2 are relabelled with the values of `x:y` and `n*n`, respectively, where the actual parameters for `x`, `y` and `n` are found by matching (injectively) the left-hand graph in a host graph. The rule is applicable only if the grey node is not incident with other edges (the *dangling condition*) and if the `where`-clause is satisfied. The latter requires that `n` is instantiated with a negative integer and that there is no edge from node 1 to node 2 in the host graph.

`replace(n: int; s,t: string; a: atom; x,y: list)`



`where n < 0 and not edge(1, 2)`

Figure 1: Declaration of a conditional rule

The grammar in Figure 2 gives the abstract syntax of GP 2 programs (omitting rule declarations and graph labels). A program consists of declarations of conditional rules and procedures, and exactly one declaration of a main command sequence. The category RuleId refers to declarations of conditional rules in RuleDecl. Procedures must be non-recursive, they can be seen as macros with local declarations.

Prog	::=	Decl {Decl}
Decl	::=	RuleDecl | ProcDecl | MainDecl
ProcDecl	::=	ProcId '=' ['[' LocalDecl ']'] ComSeq
LocalDecl	::=	(RuleDecl | ProcDecl) {LocalDecl}
MainDecl	::=	`Main` '=' ComSeq
ComSeq	::=	Com {';' Com}
Com	::=	RuleSetCall | ProcCall
		| `if` ComSeq `then` ComSeq [`else` ComSeq]
		| `try` ComSeq [`then` ComSeq] [`else` ComSeq]
		| ComSeq '!' | ComSeq `or` ComSeq
		| '(' ComSeq ')' | `break` | `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId {',' RuleId}] '}'
ProcCall	::=	ProcId

Figure 2: Abstract syntax of GP 2 programs

The call of a rule set $\{r_1,\ldots,r_n\}$ non-deterministically applies one of the rules whose left-hand graph matches a subgraph of the host graph such that the dangling condition and the rule's application condition are satisfied. The call *fails* if none of the rules is applicable to the host graph.

The command `if` C `then` P `else` Q is executed on a host graph G by first executing C on a copy of G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The `try` command has a similar effect, except that P is executed on the result of C's execution.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body was entered for the last time. The `break` command inside a loop terminates that loop and transfers control to the command following the loop.

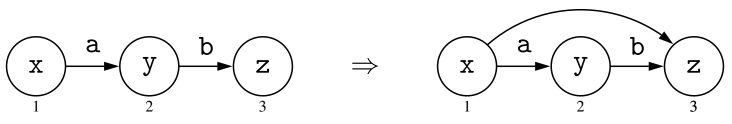
A program P `or` Q non-deterministically chooses to execute either P or Q, which can be simulated by a rule-set call and the other commands [12]. The commands `skip` and `fail` can also be expressed by the other commands.

3 Case Study: Transitive Closure

A graph is *transitive* if for each directed path from a node v to another node v', there is an edge from v to v'. The program `TransClosure` in Figure 3 computes the transitive closure of a host graph G by applying the single rule `link` as long as possible. Each application amounts to non-deterministically selecting a subgraph of G that matches `link`'s left graph, and adding to it an edge from node 1 to node 3 provided there is no such edge (with any label). Figure 4 shows an execution of `TransClosure` in which `link` is applied eight times (arcs with two arrowheads represent pairs of edges of opposite direction). The resulting graph is the complete graph of four nodes because the start graph is a cycle.

```
Main = link!

link(a,b,x,y,z: list)
```



```
where not edge(1,3)
```

Figure 3: The program `TransClosure`

The next two propositions show that `TransClosure` is correct: for every input graph G, the program produces the smallest transitive graph that results from adding unlabelled edges to G.[2]

Proposition 1 (Termination)**.** *On every host graph G, program `TransClosure` terminates after at most* $|V_G| \times |V_G|$ *rule applications.*

Proof. Given any host graph X, let

$$\#X = |\{\langle v,w\rangle \in V_X \times V_X \mid \text{there is no edge from } v \text{ to } w\}|.$$

Note that $\#X \leq |V_X| \times |V_X|$. By the application condition of `link`, every step $G \Rightarrow_{\texttt{link}} H$ satisfies $\#H = \#G - 1$. Hence `link`! terminates after at most $|V_G| \times |V_G|$ rule applications. $\square$

[2]By a graphical convention of GP 2, "unlabelled" nodes and edges are actually labelled with the empty list.

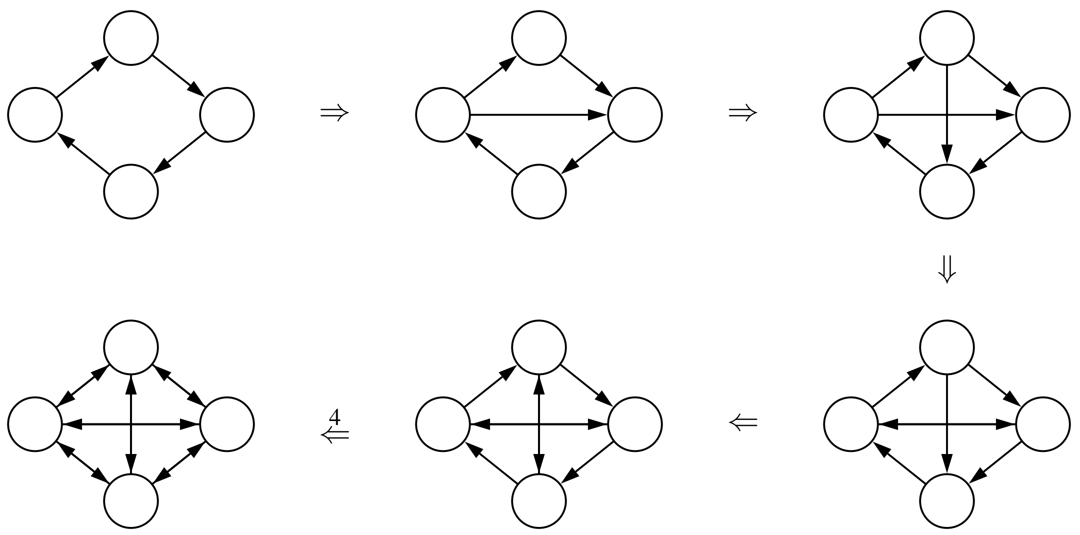


Figure 4: An execution of `TransClosure`

Proposition 2 (Correctness)**.** *Program* `TransClosure` *returns the transitive closure of the input graph.*

Proof. The previous proposition guarantees that for every input graph G, the loop `link!` returns some host graph T. Because `link` does not delete or relabel any items, each step $X \Rightarrow_{\texttt{link}} Y$ comes with an injective graph morphism $X \to Y$. It follows that T is an extension of G (up to isomorphism).

We show that T is transitive by induction on the length of paths in T. Consider a directed path $v_0, v_1, \ldots, v_n$ with $v_0 \neq v_n$. Without loss of generality, we can assume that $v_0, \ldots, v_n$ are distinct. If $n = 1$, there is nothing to show because there is an edge from v_0 to v_1. Assume now $n > 1$. By induction hypothesis, there is an edge from v_0 to v_{n-1}. Thus there exist edges $v_0 \to v_{n-1} \to v_n$ in T. Since `link` has been applied as long as possible, there must exist an edge from v_0 to v_n. (If there was no such edge, `link` would be applicable to T.)

Finally, T is the smallest transitive extension of G by the following invariant: given any derivation $G \Rightarrow^*_{\texttt{link}} H$ and any edge $v \to v'$ in H created by the derivation, there is no such edge in G but a path from v to v'. This invariant is shown by a simple induction on the length of derivations $G \Rightarrow^*_{\texttt{link}} H$. □

4 Case Study: Vertex Colouring

A *vertex colouring* for a graph G is an assignment of colours to G's nodes such that adjacent nodes have different colours. As common in the literature [6], we use positive integers as colours. The program `Colouring` in Figure 5 colours an input graph by adding an integer to each node's label. Initially, each node gets colour 1 by the loop `init!`. To prevent repeated applications of `init` to the same node, nodes are first shaded and then unmarked by `init`. Once all nodes have got colour 1, the loop `inc!` repeatedly increments the target colour of edges that have the same colour at source and target. This continues as long as there are pairs of adjacent nodes with the same colour. Note that this process is highly non-deterministic and may result in different colourings; for example, Figure 6 shows two different outcomes for the same input graph. (Finding a colouring with a minimal number of colours is an NP-complete problem [6] and requires a more complicated program.)

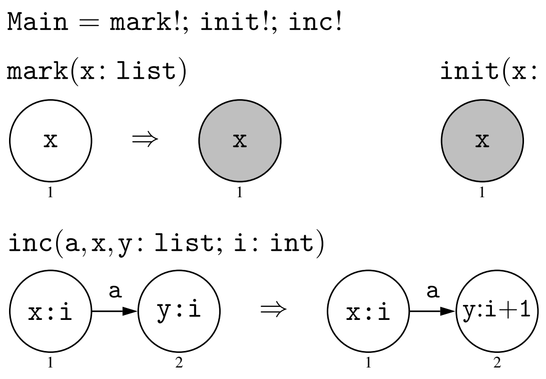


Figure 5: The program `Colouring`

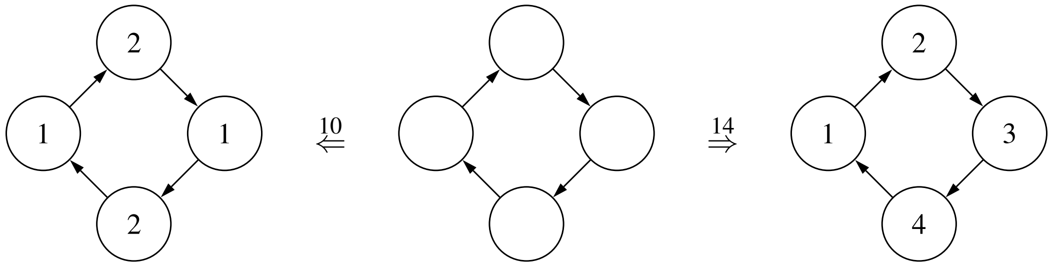


Figure 6: Different results from executing `Colouring`

The partial correctness of `Colouring` is easy to establish, we get it essentially for free from the meaning of the loop construct '`!`'. We will address termination afterwards.

Proposition 3 (Partial correctness)**.** *Given any input graph G, if `Colouring` terminates then it returns G correctly coloured.*

Proof. Let $G \Rightarrow^*_{\mathtt{mark}} G' \Rightarrow^*_{\mathtt{init}} H \Rightarrow^*_{\mathtt{inc}} M$ be an execution of `Colouring` on G. Then H is obtained from G by replacing each node label x with x:1. If M is not correctly coloured, then it must contain adjacent nodes with the same colour. Hence `inc` would be applicable to M, but the meaning of '`!`' implies that M results from applying `inc` as long as possible. □

Proving that `Colouring` terminates is more challenging, we first establish an invariant of `inc`. Given a node v with a label of the form $x{:}i$, $i \in \mathbb{N}$, we denote i by colour(v); for a host graph G with coloured nodes, we define $\text{Colours}(G) = \{\text{colour}(v) \mid v \in V_G\}$.

Lemma 1 (Invariant)**.** *Consider any derivation* $G \Rightarrow^*_{\mathtt{inc}} H$ *with* $\text{Colours}(G) = \{1\}$. *Then* $\text{Colours}(H) = \{i \mid 1 \leq i \leq n\}$ *for some* $1 \leq n \leq |V_H|$.

Proof. For every step $X \Rightarrow_{\mathtt{inc}} Y$, we have $\text{Colours}(Y) = \text{Colours}(X) \cup \Delta$ where Δ is either empty or $\{\max(\text{Colours}(X)) + 1\}$. The invariant follows then by induction on the length of $G \Rightarrow^*_{\mathtt{inc}} H$. □

Given any coloured host graph X, define $\#X = \sum_{v \in V_X} \text{colour}(v)$. Then the invariant of Lemma 1 provides an upper bound for $\#H$, viz. $\#H \leq 1 + 2 + \cdots + |V_H|$. We exploit this in the following proof.

Proposition 4 (Termination)**.** *On every input graph G, `Colouring` terminates after* $\text{O}(|V_G|^2)$ *rule applications.*

Proof. It is clear that the loops `mark!` and `init!` terminate: the first decreases in each step the number of unmarked nodes, the second decreases in each step the number of marked nodes.

To show that `inc!` is terminating, consider a coloured graph G with $\text{Colours}(G) = \{1\}$ and suppose that there is an infinite derivation $G = G_0 \Rightarrow_{\texttt{inc}} G_1 \Rightarrow_{\texttt{inc}} G_2 \Rightarrow_{\texttt{inc}} \ldots$ Then, by the labelling of `inc`, we have $\#G_i < \#G_{i+1}$ for every $i \geq 0$. However, Lemma 1 implies that for all $i \geq 0$,

$$\#G_i \leq \sum_{j=1}^{|V_{G_i}|} j = \sum_{j=1}^{|V_G|} j$$

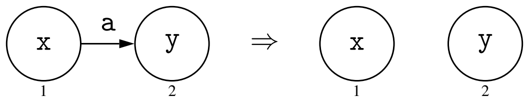
where $|V_{G_i}| = |V_G|$ holds because `inc` preserves the number of nodes. Thus the infinite derivation cannot exist. Moreover, because the upper bound for the values $\#G_i$ is quadratic in $|V_G|$, any sequence of `inc` applications starting from G cannot contain more than $\text{O}(|V_G|^2)$ rule applications. □

5 Case Study: Cycle Checking

Our third example program shows how to test the input graph for a property and then continue computing with the same graph. The program in Figure 7 checks whether a host graph G contains a directed cycle and then, depending on the result, executes either program P or program Q on G.

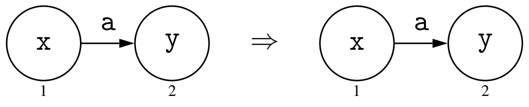
```
Main = if Cyclic then P else Q
Cyclic = delete!; {edge, loop}
```

`delete(a,x,y: list)`



`where indeg(1) = 0`

`edge(a,x,y: list)`



`loop(a,x: list)`

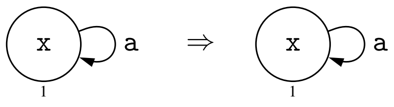


Figure 7: The program `CycleCheck`

The presence of cycles is checked by deleting, as long as possible, edges whose source nodes have no incoming edges. To ensure the latter, rule `delete` uses GP 2's built-in function `indegree` which returns the number of edges going into a node. When `delete` is no longer applicable, the resulting graph contains edges if and only if the input graph is cyclic. For example, Figure 8 shows two executions of `CycleCheck`, the left on a cyclic input graph and the right on an acyclic graph. (We also show an intermediate graph for each execution.)

The correctness of this method for cycle checking relies on the following invariant.

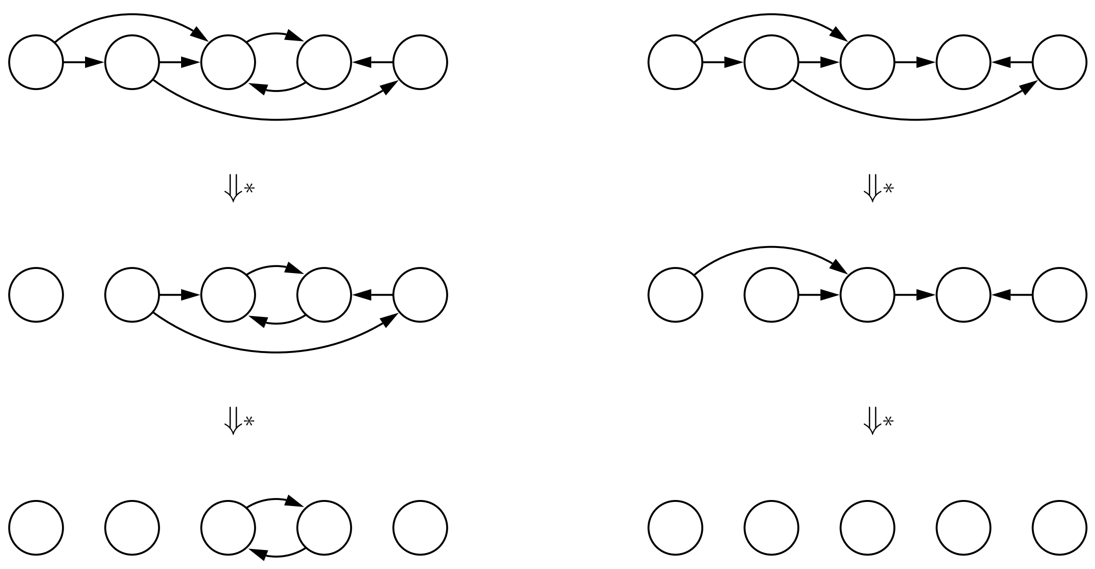


Figure 8: Two executions of `CycleCheck`

Lemma 2 (Invariant)**.** *Given any step $G \Rightarrow_{\texttt{delete}} H$, graph G is cyclic if and only if graph H is cyclic.*

Proof. Suppose that G contains a directed cycle. By the application condition of `delete`, none of the edges on the cycle are deleted. Conversely, if G is acyclic, deleting an edge cannot create a cycle. □

We also need the following property of acyclic graphs, which is easy to prove [5].

Lemma 3 (Acyclic graphs)**.** *Every non-empty acyclic graph contains a node without incoming edges.*

For stating the correctness of `CycleCheck`, we use GP 2's semantic function $[\![\_]\!]$ which maps each input graph to the set of possible execution outcomes (see [12]).

Proposition 5 (Correctness)**.** *For every host graph G,*

$$[\![\texttt{CycleCheck}]\!]G = \begin{cases} [\![P]\!]G & \text{if } G \text{ is cyclic,} \\ [\![Q]\!]G & \text{otherwise.} \end{cases}$$

Proof. We show that running procedure `Cyclic` on G returns some graph if G is cyclic, and fails otherwise. By the meaning of the if-then-else statement, this implies the proposition.

Since each application of `delete` reduces graph size, executing the loop `delete!` on G results in some graph H.

Case 1: G is cyclic. Then, by Lemma 2 and a simple induction on the derivation $G \Rightarrow^*_{\texttt{delete}} H$, graph H is also cyclic. Hence H contains at least one edge, implying that $\{\texttt{edge}, \texttt{loop}\}$ is applicable to H.

Case 2: G is acyclic. With Lemma 2 follows that H is acyclic, too. We show that H does not contain edges. Suppose that H contains some edges (which cannot be loops). Consider the subgraph S of H obtained by removing all isolated nodes. Then, by Lemma 3, there is a node v in S without incoming edges. Since v is not isolated, it must have an outgoing edge e. But then `delete` is applicable to e, contradicting the fact that `delete` is not applicable to H.

Thus H is edge-less and hence $\{\texttt{edge}, \texttt{loop}\}$ fails on H. □

6 Case Study: Series-Parallel Graphs

Our final case study is the recognition of series-parallel graphs. This graph class was introduced in [7] as a model of electrical networks and comes with an inductive definition:

- Every graph G consisting of two distinct nodes v_1, v_2 and an edge from v_1 to v_2 is series-parallel. Define $\text{source}(G) = v_1$ and $\text{sink}(G) = v_2$.
- Given series-parallel graphs G and H, each of the following operations yields a series-parallel graph when applied to the disjoint union $G + H$:
  - Series composition: Merge $\text{sink}(G)$ with $\text{source}(H)$. Define $\text{source}(G)$ to be the new source and $\text{sink}(H)$ to be the new sink.
  - Parallel composition: Merge $\text{source}(G)$ with $\text{source}(H)$ and $\text{sink}(G)$ with $\text{sink}(H)$. Define the merged source nodes to be the new source and the merged sink nodes to be the new sink.

Series-parallel graphs can be characterised by two reduction operations on graphs. On GP 2 graphs, these operations are equivalent to applying one of the rules `series` and `parallel` from Figure 9. Hence we can state the characterisation in terms of derivations with the rule set `Reduce` = {`series`, `parallel`}.

Proposition 6 ([7])**.** *A host graph G is series-parallel if and only if there is a derivation $G \Rightarrow^*_{\text{Reduce}} E$, where E is a series-parallel graph consisting of two nodes and one edge.*

Given a host graph G, the procedure `Series-parallel` in Figure 9 applies the reduction rules as long as possible to obtain some reduced graph H. (Termination is guaranteed because both rules decrease graph size.) To check whether H has the shape of graph E from Proposition 6, the procedure attempts to delete a subgraph of E's shape and then tests if there is any remaining node. If the deletion fails or a remaining node is detected, H does not have the shape of E and the procedure fails. Otherwise, if rule `delete` succeeds and rule `nonempty` fails, the procedure succeeds by returning the empty graph.

```
Series-parallel = Reduce!; delete; if nonempty then fail
Reduce = {series, parallel}
```

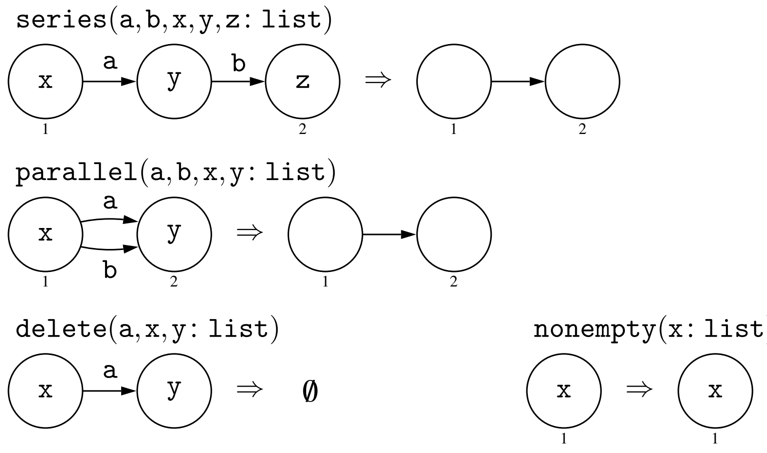
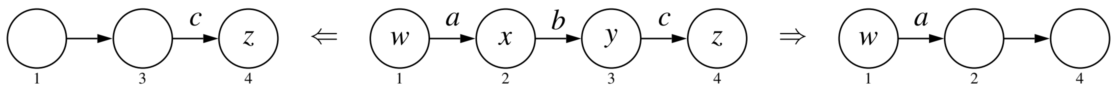


Figure 9: The program `SeriesParallel`

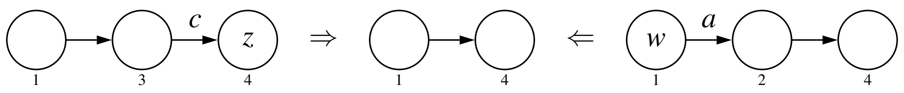
Proposition 6 alone does not guarantee the correctness of SeriesParallel. This is because if the non-deterministic application of `Reduce` does not end in E, there might be some other reduction sequence ending in E. We exclude this possibility by showing that any complete reduction of a host graph ends in a graph that is unique up to isomorphism.

Proposition 7 (Uniqueness of reduced graphs)**.** *Consider reductions $H_1 \Leftarrow^*_{\mathrm{Reduce}} G \Rightarrow^*_{\mathrm{Reduce}} H_2$ of some host graph G. If H_1 and H_2 are irreducible, then they are isomorphic.*

Proof. We show that `Reduce` is *confluent* which implies the claim. To employ the technique of [11], we analyse the *critical pairs* of all rules obtained from `series` and `parallel` by replacing variables with constant lists. There are no critical overlaps between `series` and `parallel`, hence all critical pairs are self-overlaps of either `series` or `parallel`. There are two types of critical pairs of `parallel`, which are easily shown to be *strongly joinable* in the terminology of [11]. There are also two types of critical pairs of `series`. We consider one of them:



where a to c and w to z are arbitrary host graph lists. This pair is strongly joinable by the following applications of `series`:



The other critical pair of `series` is obtained from the above pair by merging nodes 1 and 4 in all three graphs. Then the outer graphs are isomorphic in the way required by strong joinability. With [11] follows that `Reduce` is confluent. □

Proposition 8 (Correctness)**.** *For every host graph G, running `Series-parallel` on G returns the empty graph if G is series-parallel and fails otherwise.*

Proof. Let H be the graph resulting from running `Reduce` on G. By Proposition 7, H is determined uniquely up to isomorphism. Thus, if G is series-parallel, Proposition 6 implies that H consists of two nodes and an edge between them. It follows that `Series-parallel` returns the empty graph.

If G is not series-parallel, Proposition 6 implies that H has some other shape. Then either rule `delete` is not applicable or `delete` is applicable but afterwards `nonempty` is applicable. In both cases `Series-parallel` fails, as required. □

7 Conclusion

In this paper, we show by some case studies that GP 2's graph transformation rules allow high-level reasoning on partial and total correctness of graph programs. Instead of employing a rigorous formalism such as the Hoare-logic in [13], we use standard mathematical language for expressing assertions and proofs. This both increases the expressive power of specifications and frees programmers from the notational constraints of a formal verification calculus. The main mathematical tool in our sample proofs is induction over derivation sequences of graph transformation rules.

We do not propose to abandon rigorous program verification in favour of a more liberal approach to justifying correctness. In contrast, proof calculi with precise syntax and semantics are indispensable for achieving confidence in verified software. We view this paper's approach as complementary in that it allows programmers to reason about graph programs without getting stuck in formal detail. Future work should address the question how to refine proofs in ordinary mathematical language into rigorous proofs in verification calculi.

References

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

[2] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language*. Ph.D. thesis, Department of Computer Science, The University of York. Available at `http://etheses.whiterose.ac.uk/12586/`.

[3] Christopher Bak, Glyn Faulkner, Detlef Plump & Colin Runciman (2015): *A Reference Interpreter for the Graph Programming Language GP 2*. In: Proc. Graphs as Models (GaM 2015), Electronic Proceedings in Theoretical Computer Science 181, pp. 48–64, doi:10.4204/EPTCS.181.

[4] Christopher Bak & Detlef Plump (2016): *Compiling Graph Programs to C*. In: Proc. International Conference on Graph Transformation (ICGT 2016), Lecture Notes in Computer Science , Springer. To appear.

[5] Jørgen Bang-Jensen & Gregory Gutin (2009): *Digraphs: Theory, Algorithms and Applications*, second edition. Springer, doi:10.1007/978-1-84800-998-1.

[6] Thomas H. Cormen, Charles E. Leiserson, Robert L. Rivest & Clifford Stein (2009): *Introduction to Algorithms*, third edition. The MIT Press.

[7] R. J. Duffin (1965): *Topology of Series-Parallel Networks*. Journal of Mathematical Analysis and Applications 10(2), pp. 303–318, doi:10.1016/0022-247X(65)90125-3.

[8] Maribel Fernández, Hélène Kirchner, Ian Mackie & Bruno Pinaud (2014): *Visual Modelling of Complex Systems: Towards an Abstract Machine for PORGY*. In: Proc. Computability in Europe (CiE 2014), Lecture Notes in Computer Science 8493, Springer, pp. 183–193, doi:10.1007/978-3-319-08019-2_19.

[9] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and analysis using GROOVE*. International Journal on Software Tools for Technology Transfer 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[10] Edgar Jakumeit, Sebastian Buchwald & Moritz Kroll (2010): *GrGen.NET - The expressive, convenient and fast graph rewrite system*. International Journal on Software Tools for Technology Transfer 12(3–4), pp. 263–271, doi:10.1007/s10009-010-0148-8.

[11] Detlef Plump (2005): *Confluence of Graph Transformation Revisited*. In: Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday, Lecture Notes in Computer Science 3838, Springer, pp. 280–308, doi:10.1007/11601548.

[12] Detlef Plump (2012): *The Design of GP 2*. In: Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011), Electronic Proceedings in Theoretical Computer Science 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[13] Christopher M. Poskitt & Detlef Plump (2012): *Hoare-Style Verification of Graph Programs*. Fundamenta Informaticae 118(1-2), pp. 135–175, doi:10.3233/FI-2012-708.

[14] Christopher M. Poskitt & Detlef Plump (2014): *Verifying Monadic Second-Order Properties of Graph Programs*. In: Proc. International Conference on Graph Transformation (ICGT 2014), Lecture Notes in Computer Science 8571, Springer, pp. 33–48, doi:10.1007/978-3-319-09108-2_3.

[15] Olga Runge, Claudia Ermel & Gabriele Taentzer (2012): *AGG 2.0 — New Features for Specifying and Analyzing Algebraic Graph Transformations*. In: Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011), Lecture Notes in Computer Science 7233, Springer, pp. 81–88, doi:10.1007/978-3-642-34176-2_8.