Proceedings Paper:
Plump, DETLEF and SOELDNER, ROBERT (2026) Formalising and Verifying Graph Programs with Higher-Order Logic. In: Proceedings, 19th International Conference on Graph Transformation (ICGT 2026). International Conference on Graph Transformation, 29 Jun - 03 Jul 2026 Lecture Notes in Computer Science. Springer, FRA, pp. 109-128.

https://doi.org/10.1007/978-3-032-29730-3_6

Formalising and Verifying Graph Programs with Higher-Order Logic

Robert Söldner[1 [0000−0001−5295−3493]] and Detlef Plump[1 [0000−0002−1148−822X]]

University of York, UK
{rs2040,detlef.plump}@york.ac.uk

Abstract. We present an approach to verifying graph programs with higher-order logic, based on a formalisation of the graph programming language GP 2 in the HOL4 interactive theorem prover. The formalisation covers all language features, from labelled graphs and graph morphisms to rule schemata and their application, up to the full operational semantics. To prove programs correct, we use Hoare-style proof rules which have been verified with respect to the operational semantics. We introduce *track morphisms* of program executions as a new concept, to locate nodes and edges of the input graph in the output graph. Due to the power of higher-order logic, we can formalise track morphisms in HOL4 and use them in assertions. We demonstrate the use of our verification framework by a case study in which we prove the partial correctness of a program that computes the transitive closure of input graphs. The post-condition asserts not only that the result graph is transitive but also that it is a minimal extension of the input graph. The case study involves 50 mechanically verified theorems while the formalisation of GP 2 comprises approximately 20,000 lines of proof in 19 theories.

Keywords: Rule-based graph programming · GP 2 · Interactive theorem proving · HOL4 · Formalised operational semantics · Hoare-style proof rules · Track morphisms · Mechanical program verification

1 Introduction

Graph-structured data are ubiquitous, from pointer structures to program control-flow and call graphs, knowledge graphs, graph databases, model-driven software engineering, and biological networks. Rule-based graph transformation offers a visual formalism for computing with such data and underpins modelling languages, compiler optimisations, and domain-specific tools. As graph-manipulating code enters safety-critical and data-intensive domains, the correctness problem familiar from conventional programming becomes increasingly important: how do we prove that a given graph program meets its specification? In the context of imperative and functional programming, one approach to verification is the use of interactive theorem provers as witnessed by CompCert [9], seL4 [8], and CakeML [1]. However, for rule-based graph programs, verification has so far remained almost exclusively a pen-and-paper activity.

The graph programming language GP 2 [14,3] is a rule-based experimental language with control constructs such as sequential composition, as-long-as-possible iteration, branching, and procedures. It comes with a formal semantics based on (an attributed version of) the double-pushout approach to graph transformation [6] and has been shown to be computationally complete [15].

Several verification calculi for GP 2 have been developed, using as assertions either first-order logic or monadic second-order logic on graphs [16,17,18,23,24,19]. These approaches come with weakest-precondition constructions, but none of them has been implemented yet.

In this paper we present, to the best of our knowledge, the first mechanised formalisation of a complete graph programming language in a theorem prover. Our development in HOL4 covers the full GP 2 language: from labelled graphs over morphisms and rule schemata up to attributed double-pushout graph transformation and GP 2's small-step operational semantics. In addition, we introduce a compositional program verification framework based on Hoare-style proof rules.

Specifically, this paper makes the following contributions:

- We formalise the complete GP 2 language in HOL4 and establish a Hoare-style verification framework covering sequential composition, consequence, conditional branching and as-long-as-possible iteration, among other constructs.
- We introduce track morphisms of program executions that allow to locate preserved nodes of input graphs in output graphs, and show how to extend these morphisms to injective graph morphisms that additionally track edges.
- We demonstrate our verification approach by proving the partial correctness of a GP 2 program computing the transitive closure of input graphs. While the program is short, the mechanised proof is of substantial size and involves 50 theorems.

2 Background

This section briefly introduces GP 2[1] and HOL4[2]. For more background on GP 2 we refer to [14,3] and for the HOL family to [11,20].

2.1 The Graph Programming Language GP 2

GP 2 programs transform input graphs by applying conditional rule schemata through attributed double-pushout graph transformation. We use the transitive closure program of Figure 1 as a running example throughout this paper. The program applies the `link` rule schema as long as possible, which requires to find a match satisfying the application condition that there must be no edge from node 1 to node 3. If a match is found, the rule adds an edge (labelled with the empty list) according to the rule and the match. This is repeated zero or more

[1] `https://github.com/UoYCS-plasma/GP2`
[2] `https://hol-theorem-prover.org`

times until a match cannot be found anymore. Finding a match involves finding a node corresponding to node `1`, then finding an outgoing edge to another node, corresponding to node `2`, then finding an outgoing edge to yet another node, corresponding to node `3`, and then checking the application condition.

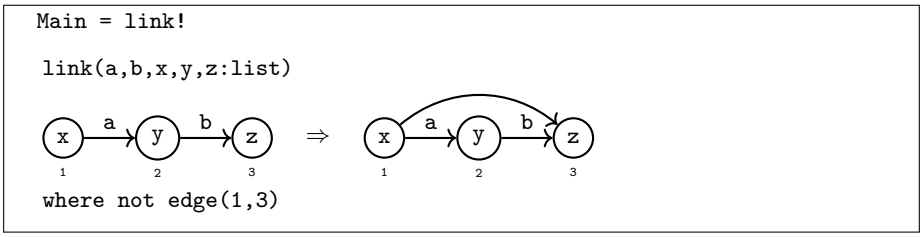

Fig. 1: GP 2 program `transitive-closure`.

GP 2 host graphs are directed and may have parallel edges and loops. Both nodes and edges are labelled with heterogeneous lists of integers and strings, organised in a small subtyping hierarchy. In addition, nodes and edges may be *marked* red, green, blue, grey (nodes only) or dashed (edges only). Our simple running example program does not require marks.

GP 2 provides several constructs for controlling the application of rules and procedures. The `transitive-closure` program uses only the loop construct $P!$ which executes the body P as long as possible, terminating when P fails and returning the graph on which the last iteration was entered.

When `link!` terminates, every pair of distinct nodes connected by a directed path in the input graph is connected by an edge in the output graph. Hence the output graph is the transitive closure of the input graph.

2.2 The Interactive Theorem Prover HOL4

HOL4 is an interactive theorem prover for higher-order logic following the LCF architecture: a small trusted kernel guarantees that every proof reduces to primitive inferences [20] . The logic extends Church's simply typed lambda calculus [2] with polymorphism: types include base types, function types, and polymorphic constructors (*e.g.* `'a list`, `('a,'b) fmap`). Quantifiers range over all types including functions and predicates, enabling assertions that quantify over morphisms, parameterised judgements, and polymorphic record types modelling graph structures.

Our formalisation uses `Datatype` (algebraic types for graphs, labels, terms), `Definition` (predicates, recursive functions), and `Inductive` definitions (operational semantics). The two central HOL4 standard-library theories we rely on are `finite_mapTheory` (finite maps for representing graphs) and `pred_setTheory` (predicate sets for node/edge sets).

3 Formalising GP 2 in HOL4

We now present our formalisation of GP 2 in HOL4. The development comprises approximately 20,000 lines of proof across ∼20 theories and has been built with HOL4 master `f02a261b` (2025-11-27) using Poly/ML 5.9.1. The complete source code is available on GitHub.[3]

3.1 Graphs and Morphisms

Following our earlier Isabelle/HOL formalisation [21], graphs are represented as a polymorphic record with finite maps for the node set `G.V`, edge set `G.E`, source and target maps, label and mark maps, and a rootedness map `G.p`. The record is polymorphic in its node-label type `'l` and edge-mark type `'m`, instantiated for host graphs and rule graphs respectively (Section 3.2). The well-formedness predicate `wf_graph` requires finite node/edge sets, consistent source/target maps, and disjoint underlying identifiers.

A *premorphism* [5] is a pair of finite maps (one for nodes, one for edges) that preserve sources, targets and rootedness. A *morphism* is a premorphism that additionally preserves node and edge labels.

3.2 Labels and Types

The polymorphic label parameter `'l` of the graph record is instantiated with concrete label types for rule graphs and host graphs, respectively. Rule labels include constants (integers, strings, characters), list constructors (empty list and concatenation), variables, arithmetic and string operations, and degree expressions. Host labels are the ground fragment of rule labels: they are built exclusively from constants and list constructors, without variables or other operations. A host label is in *spine form* when it is an atom, the empty list, or a cons cell whose head is an atom and whose tail is in spine form. This canonical form is maintained as an invariant throughout label evaluation. Marks (`nodemark`, `edgemark`) instantiate the `'m` parameter. As mentioned in Section 2.1, rule marks additionally include a wildcard `any` for pattern matching. (We omit the mark definitions to save space.)

Typing is defined as an inductive relation `label_typeof` over a variable context, a label, and a type drawn from the subtype hierarchy. Here is a representative rule:

$$\frac{\texttt{FLOOKUP}\ \mathit{vars}\ v\ \texttt{=}\ \texttt{SOME}\ \mathit{ty}}{\texttt{label\_typeof (}\mathit{vars}\texttt{,label\_variable}\ v\texttt{,}\mathit{ty}\texttt{)}}\ \textsf{Var}$$

[3] `https://github.com/UoYCS-plasma/gp2-hol`. The repository includes the full GP 2 formalisation, the transitive-closure case-study proof of Section 6, and the `gp2convert` tool that translates GP 2 source code into HOL4 theory files.

We omit the remaining rules (arithmetic, concatenation, cons, degree expressions) for space reasons.

A functional type checker `label_typeof_fun` is proved equivalent to the inductive relation, enabling computation inside proofs. Well-formedness of a host graph (`wf_hostgraph`) requires all labels to be in spine form; an analogous predicate `wf_rulegraph` constrains rule graphs to carry well-typed labels.

3.3 Attributed Double-Pushout Graph Transformation

A GP 2 rule schema comprises variable declarations, a left-hand graph and a right-hand graph (both well-formed), an interface consisting of unlabelled nodes only, and an optional application condition. Well-formedness (`wf_rule`) requires properties such as total labelling of left and right-hand graph, well-typing, restriction of RHS variables to those in the LHS, and well-formedness of the application condition.

Applying a rule requires first to instantiate the variables of the left-hand graph with values such that an injective graph morphism into the host graph exists. Subsequently, the application condition is evaluated and, if true, any expressions in the right-hand graph are evaluated. This yields an instance of the rule schema which is a rule in the double-pushout approach with relabelling [7]. This rule is then applied according to that approach and with the given match.

To keep the presentation simple, we ignore here so-called rooted rule schemata which allow to speed-up the matching process. See, for example, [3].

3.4 Operational Semantics and Track Morphisms

We follow GP 2's small-step operational semantics of Courtehoute and Plump [6]. Semantic inference rules (omitted here for lack of space) define a transition relation on *configurations*. These are either *running configurations* $\langle P, S\rangle$ consisting of a program P and a graph stack S, or *terminal configurations* of two kinds: just a single-entry graph stack S (denoting successful termination) or the element fail (denoting program failure). We write $[G]$ for the single-element stack whose only entry is graph G.

The *semantic function* $[\![P]\!]$ of a program P is then defined as follows:

$$[\![P]\!]\, G \;=\; \{X \mid \langle P, [G]\rangle \to_P^* X,\ X \text{ a terminal configuration}\} \\ \cup\ \{\bot \mid P \text{ can diverge from } G\},$$

where $\to_P^*$ denotes the reflexive-transitive closure of the small-step transition relation. Section 4 discusses the formalisation of the transition relation.

Similar to the track morphism in standard double-pushout graph transformation [13], every rule schema application transforming a graph G into a graph H induces a *track morphism* $tr\colon G \to H$: a partial, injective graph morphism that locates the preserved nodes (and unused edges) of G in H. We extend tr to a multi-step morphism $tr_{G_0 \to_P^* G_n}$ for every transition sequence transforming

an input graph G_0 into an output graph G_n. We do this by making each stack entry in the configurations of [6] a pair $(G,\ tr_{G_0 \to_P^* G_i})$ whose second component is the track morphism from the initial graph G_0 to the current graph G_i. Throughout, we highlight the new track-morphism component of a stack entry with a light grey background. By erasing the highlighted components one recovers the plain semantic inference rules of [6]. Only one inference rule changes non-trivially, namely the rule-set application $[\text{call}'_1]$ which composes the single-step track $tr_{G \to_R H}$ with the multi-step track $tr_{G_0 \to_P^* G}$:

$$[\text{call}'_1] \quad \frac{(G,\ tr_{G_0 \to_P^* G}) = \text{top}(S) \qquad G \Rightarrow_R H}{\langle R,\ S\rangle \to \text{push}\big((H,\ tr_{G \to_R H} \circ tr_{G_0 \to_P^* G}),\ \text{pop}(S)\big)}$$

All other rules in [6] remain unchanged, the track component is threaded through without modification. The branching constructs `if_then_else` and `try_then_else` which discard the condition's graph also discard the associated track morphism. The HOL4 formalisation of this extended transition relation is discussed in Section 4.2.

4 Formalising the Operational Semantics

We now discuss the HOL4 formalisation of the operational semantics outlined in Section 3.4.

4.1 Rule Matching and Application

The function `apply_rule` implements rule application as a three-phase monadic pipeline: (1) `mk_assignment` replaces each LHS variable with its matched host value and merges the partial assignments; (2) `instantiate_rule` checks the application condition and evaluates all rule labels in the RHS to produce a concrete rule instance; (3) `apply_ruleinstance` feeds the instance and the match into the DPO construction (Section 3.3). The match is an injective premorphism satisfying the dangling condition (nodes to be deleted must not be incident with host graph edges outside the match).

```
⊢ wf_rule r ∧ wf_hostgraph G ∧
  wf_rulegraph_labels r.lhs ∧
  is_prematch r.lhs r.inf m G ∧
  apply_rule r m G = SOME h ⇒
  wf_hostgraph h
```

This theorem connects the three phases: spine-form preservation, proved for variable instantiation and label evaluation, guarantees that the instantiated labels are well-formed, and the DPO results from Section 3.3 establish that the output graph is well-formed.

4.2 Program Execution

We formalise the configurations of Section 3.4 as follows. A HOL4 configuration pairs a state (`running` t, `final` or `failed`) with a stack of (hostgraph, morphism) frames, where the morphism is the track morphism $tr_{G_0 \to_P^* G}$ from the original input graph. The initial configuration is (`running` P, $[(G, id\_track\ G)]$).

The inductive `step` relation has 21 rules. On successful rule application, the per-step track is composed with the track morphism via `compose_morphism`. Sequential composition propagates steps through the first sub-term; as-long-as-possible iteration unfolds $P!$ to `try` P `then` $P!$ `else skip`. Branching constructs push a saved frame onto the stack; `skip` and `fail` transition to `final` and `failed`, respectively.

Evaluation holds when the reflexive-transitive closure of `step` reaches a `final` configuration:

```
⊢ evaluate env G P (H,tr) ⟺
  ∃S. steps env (running P,[(G,id_track G)])
        (final,S) ∧ ¬can_step (final,S) ∧
      ∃rest. S = (H,tr)::rest
```

Source-level programs preserve the stack length across evaluation, so the final stack has exactly one frame whose host graph is the output. Together with stack-length preservation, the step-level preservation of stack well-formedness yields well-formedness of the output graph:

```
⊢ wf_program env ∧ wf_hostgraph G ∧
  evaluate env G P (H,tr) ⇒
  wf_hostgraph H
```

The step relation also preserves validity of the track morphism; we return to this in Section 5.3.

5 Program Verification Framework

With the operational semantics in place, we can develop a compositional framework for proving partial correctness of GP 2 programs. Subsection 5.1 explains what partial correctness means and what kind of assertions we are using. Subsection 5.2 presents some Hoare-style proof rules and explains in what sense they are sound. Subsection 5.3 discusses the formalisation of track morphisms.

5.1 Partial Correctness and Assertions

We adopt the notion of partial correctness from Poskitt and Plump [18]. A graph program t is *partially correct* with respect to a precondition P and postcondition Q, written $\models \{P\}\, t\, \{Q\}$, if for every host graph G with $G \models P$ and every graph $H \in [\![t]\!]G$ we have $H \models Q$. Here $[\![t]\!]$ is the semantic function of t, so $[\![t]\!]G$ is the set of possible outcomes of running t on G (possibly including failure and/or

divergence). Also, given a host graph X and an assertion A, $X \models A$ means that X satisfies A. Note that partial correctness makes no guarantees about program termination or the absence of failure.

From here on we use the convention that t denotes a program term, P the precondition and Q the postcondition. So the P used in Section 3.4 for programs re-appears as t in the verification framework.

In [18,24], assertions closed formulas of certain variants of monadic second-order logic. In our framework, assertions are higher-order logic formulas. HOL4 is based on classical higher-order logic in the tradition of Church's simple theory of types [4,2]: a typed lambda calculus extended with polymorphism, Hilbert choice, and quantifiers that range over every type, including function and predicate types.

Concretely, an assertion A is a HOL4 term either of type `hostgraph` $\rightarrow$ `bool` or of type `hostgraph` $\rightarrow$ `morphism` $\rightarrow$ `bool`. Equivalently, $A = \lambda G.\, \varphi(G)$ or $A = \lambda G\, m.\, \psi(G, m)$ for arbitrary higher-order logic formulas $\varphi(G)$ and $\psi(G, m)$ in which G (resp. G and m) may occur free. Thus, assertions may quantify over both graphs and morphisms.

Partial correctness is formalised by the predicate `WSPEC env P t Q`, asserting that whenever P holds for the input graph and t terminates successfully under *env*, Q holds for the output graph *and* the accumulated track morphism:

```
⊢ WSPEC env P t Q ⟺

  wf_program env ∧ no_intermediate_terms t ∧

  ∀ G og om.

    wf_hostgraph G ∧ P G ∧

    steps env (running t,[(G,id_track G)]) (final,[(og,om)]) ⇒

    Q og om
```

5.2 Hoare Rules and their Soundness

We present a few Hoare-style proof rules which suffice to carry out the case study in Section 6. The call of a GP 2 rule-set $\mathcal{R}$ is our basic proof rule. The rule's premise requires that whenever a rule in $\mathcal{R}$ is applied to a host graph satisfying the precondition P, then the resulting graph satisfies the potscondition Q:

$$[\text{rscall}] \quad \frac{\models \{P\}\ \mathcal{R}\ \{Q\}}{\{P\}\ \mathcal{R}\ \{Q\}}$$

The corresponding HOL4 theorem is `WSPEC_rscall`.

The consequence rule allows to strengthen preconditions and to weaken post-conditions:

$$[\text{cons}] \quad \frac{P \Rightarrow P' \quad \{P'\}\ t\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ t\ \{Q\}}$$

For as-long-as-possible iteration of a rule set call, the following rule requires that an invariant I holds for the loop's body:

$$[\text{alap}] \quad \frac{\{I\}\ \mathcal{R}\ \{I\}}{\{I\}\ \mathcal{R}!\ \{I \wedge \neg\text{App}(\mathcal{R})\}}$$

Here ¬App is an assertion expressing that $\mathcal{R}$ is not applicable to any graph resulting from the loop $\mathcal{R}!$. Note that I can be a predicate on (graph, morphism) pairs. The corresponding HOL4 theorem `WSPEC_alap` additionally quantifies its conclusion over an accumulated track m_0 so that the rule composes with an outer track when nested inside sequential composition.

We stress that each of the above proof rules is a HOL4 `Theorem`, not an `Axiom`: its statement has the shape *premises* ⇒ `WSPEC env P t Q`, where `WSPEC` is the semantic predicate defined by `WSPEC_def`. Because `WSPEC` *is* the partial-correctness judgement $\models \{P\}\, t\, \{Q\}$ with respect to the operational semantics, any HOL4 theorem of this form is automatically sound; the `Proof` block is itself the soundness argument. We therefore do not need to prove a separate soundness meta-theorem of the form $\vdash \{P\}\, t\, \{Q\} \Rightarrow \models \{P\}\, t\, \{Q\}$: there is no independent syntactic judgement $\vdash$ to relate to $\models$, because every rule has been derived from the semantics. This is the shallow-embedding trade-off: rules are not inspectable as data, but they come with soundness by construction.

Verifying a program amounts to decomposing a top-level WSPEC judgement via the rules until every leaf is a rule-set call or a loop. Section 6 demonstrates this pattern.

5.3 Track Morphisms

We briefly return to the track morphisms defined in Subsection 3.4 to show the formalisation in HOL4:

```
⊢ is_track_morphism G tr H ⟺
  partial_dom_ran G tr H ∧
  partial_preserve_source G tr H ∧
  partial_preserve_target G tr H ∧
  INJ (('') tr.nodemap) (FDOM tr.nodemap) H.V ∧
  INJ (('') tr.edgemap) (FDOM tr.edgemap) H.E
```

As described in Section 3.4, the operational semantics composes these per-step tracks into a track morphism $tr_{G_0 \to_P^* G_n}$. Since the identity morphism is a valid track morphism and composition preserves the track-morphism property, whenever a program evaluates from an input graph G to a result pair (H, tr), the track morphism is a valid track from G to H:

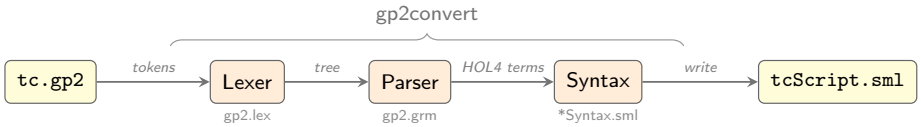

Fig. 2: The `gp2convert` pipeline. A GP 2 source file is lexed, parsed, translated into HOL4 terms, and written out as a theory file with definitions and proof obligations.

$$\vdash \texttt{wf\_program}\ env \wedge \texttt{wf\_hostgraph}\ G \wedge \texttt{evaluate}\ env\ G\ P\ (H, tr) \Rightarrow \texttt{is\_track\_morphism}\ G\ tr\ H$$

The morphism-aware ALAP rule `WSPEC_alap` used in Section 6) composes the per-iteration track at each step, combining the loop invariant with a property that must hold when the body fails.

6 Case Study: Transitive Closure

We now apply the verification framework of Section 5 to prove partial correctness of the transitive-closure program introduced in Section refsec:background. Recall that the program consists of a single procedure, `Main = link!`, which applies the `link` rule as long as possible (see Figure 1). Correctness means that whenever the program terminates, the output graph is the transitive closure of the input graph: every pair of distinct nodes connected by a directed path in the original graph is connected by an edge in the output graph, there is a total graph morphism from the input graph to the output graph that is bijective on nodes and injective on edges, and every edge not in the image of the morphism is justified by a path in the input graph. Figure 2 shows the `gp2convert` pipeline that translates the GP 2 source into HOL4 definitions, from which we derive the specification and proof.

6.1 Program and Specification

The program consists of a single procedure and a single rule. The `gp2convert` tool translates the source into a HOL4 definition of the procedure body:

```
⊢ proc_Main =
  term_alap (term_rscall {ruleid "link"})
```

The term `term_alap` encodes as-long-as-possible iteration, and `term_rscall` calls a rule set containing the single rule `ruleid "link"`. The program environment maps this identifier to the rule from Figure 1:

```
⊢ program =
  <|proc := FEMPTY |++ [("Main",proc_Main)];
    rule :=
      FEMPTY |++ [(ruleid "link",rule_link)]|>
```

Before stating the correctness property, we define the two key concepts that occur in the postcondition.

Definition 1 (Transitive graph). A graph G is *transitive* if for all nodes $v, w \in V_G$ with $v \neq w$, if there is a directed path from v to w then there exists an edge from v to w.

Note that we require $v \neq w$ as otherwise each node in a transitive graph would have to have a self-loop.

Definition 2 (Minimal extension). A graph TC is a *minimal extension* of a graph G with respect to a total injective morphism $\overline{m}\colon G \to TC$ if:

(a) $\overline{m}_V$ is bijective;
(b) each edge e in TC that is not in the image of $\overline{m}_E$,
  (b1) connects two nodes $\overline{m}(v)$ and $\overline{m}(w)$ for some $v, w \in V_G$ such that there is a directed path from v to w in G, and
  (b2) has no parallel edges in TC, that is, there is no other edge e' in TC with the same source and target as e.

Since $\overline{m}$ is a graph morphism, it preserves sources and targets of edges, and labels of both nodes and edges. By the bijectivity of the node mapping, no nodes are added or removed when transforming G into TC. Condition (b1) requires that each new edge be justified by reachability in the original graph, and condition (b2) forbids new edges to have parallel edges, as otherwise the extension would not be minimal.

Using these definitions, we can concisely state the intended correctness property of the program `transitive-closure`.

Precondition. The input graph G_0 is a well-formed, unmarked, unrooted host graph.

Postcondition. The output graph TC is transitive and there exists a total injective extension $\overline{tr}_{G_0 \to^* TC}$ of the track morphism $tr_{G_0 \to^* TC}$ such that TC is a minimal extension of G_0 with respect to $\overline{tr}_{G_0 \to^* TC}$.

The predicate `extends_morphism` (defined below) connects the morphisms tr and $\overline{tr}$ by requiring that the latter agrees with tr on tr's domain of definition. The predicate `minimal_extension` (also defined below) combines two requirements: $\overline{tr}$ must be a total injective graph morphism (`is_inj_morphism`) and TC must be a minimal extension of G with respect to $\overline{tr}$. The next subsection defines the extension of tr to $\overline{tr}$.

6.2 Total Extension of the Track Morphism

To keep GP 2's rule format simple, rule applications temporarily delete all matched edges and possibly re-create them by the DPO construction. As a consequence, track morphisms are undefined on host graph edges that are matched in any rule applications. However, the `link` rule of the program `transitive-closure` re-creates both of its edges in a unique way, allowing us to extend the partial track morphism to a total injective graph morphism.

We define the total extension inductively. The base case is the identity $\overline{tr}_{G_0 \to^* G_0} = \mathrm{id}_{G_0}$. For a rule application $G \Rightarrow_{\texttt{link}} H$ with match m and current extension $\overline{tr}_{G_0 \to^* G}$, the updated extension $\overline{tr}_{G_0 \to^* H}$ agrees with $\overline{tr}_{G_0 \to^* G}$ on nodes and maps each edge $e \in E_{G_0}$ as follows:

$$\overline{tr}_{G_0 \to^* H}(e) \;=\; \begin{cases} e' & \text{if } \overline{tr}_{G_0 \to^* G}(e) \in m(\text{LHS}), \text{ where } e' \text{ is} \\ & \text{the unique RHS edge replacing } \overline{tr}_{G_0 \to^* G}(e), \\ tr_{G \to H}(\overline{tr}_{G_0 \to^* G}(e)) & \text{otherwise (edge is not matched).} \end{cases}$$

The extension is proved to exist as part of `link_preserves_invariant` (witness `link_new_track_bar` in the repository).

6.3 Formalising the Specification

We formalise the precondition and postcondition in HOL4. The notion of minimal extension (Definition 2) is captured by the ternary relation `minimal_extension`:

```
⊢ minimal_extension G0 track_bar TC ⟺
  is_inj_morphism G0 track_bar TC ∧
  BIJ (('') track_bar.nodemap) G0.V TC.V ∧
  edge_path_justified G0 track_bar TC ∧
  parallel_free_extension track_bar TC
```

The top-level predicate `is_transitive_closure_tracked` combines transitivity and minimal extension:

```
⊢ is_transitive_closure_tracked G0 track_bar TC ⟺
  is_transitive TC ∧
  minimal_extension G0 track_bar TC
```

The predicate `extends_morphism` requires that $\overline{tr}$ extends the track morphism tr to a total morphism such that both morphisms have the same node mapping and agree on every edge in tr's domain of definition:

```
⊢ extends_morphism track track_bar ⟺
  track_bar.nodemap = track.nodemap ∧
  ∀ e. e ∈ FDOM track.edgemap ⇒
       track.edgemap ' e = track_bar.edgemap ' e
```

Transitivity (Definition 1) is formalised as:

```
⊢ is_transitive G ⟺
  ∀ v w.
    v ∈ G.V ∧ w ∈ G.V ∧ v ≠ w ∧
    reachable_in G v w ⇒
    has_edge G v w
```

The top-level correctness theorem is a WSPEC judgement:

```
⊢ WSPEC program
    (λ G.
         G = G₀ ∧ wf_hostgraph G₀ ∧
         unmarked_hostgraph G₀ ∧
         unrooted_hostgraph G₀)
    (term_proc "Main")
    (λ TC track.
         ∃ track_bar.
           is_transitive_closure_tracked G₀
             track_bar TC ∧
           extends_morphism track track_bar)
```

The postcondition requires (1) the existence of the total morphism $\overline{tr}$ extending the track morphism and (2) that the output graph is transitive and a minimal extension of the input graph. The proof decomposes into a loop invariant (Section 6.4), transitivity upon termination (Section 6.5), and preservation of the minimal extension property (Section 6.6).

6.4 Loop Invariant

The proof of `transitive_closure_correct` proceeds by instantiating the morphism-aware ALAP rule `WSPEC_alap` from Section 5.3 with a loop invariant that captures two concerns:

Loop invariant. After every iteration of `link!`, writing m for the track morphism $tr_{G_0 \to^* H}$ accumulated so far, the current graph H satisfies:

1. *Structural properties:* H is a well-formed, unmarked, unrooted host graph.
2. *Minimal extension:* H is a minimal extension of the input graph G_0 with respect to a total morphism $\overline{m}$ that extends the morphism m.

The proof maintains two parallel invariants. The *base invariant* tracks the morphism m using the weaker property `minimally_extends`, which requires m to be a partial and injective graph morphism with bijective node mapping, and every edge not in the image of m to be justified by a directed path in the input graph:

```
⊢ tc_loop_invariant G₀ H m ⟺
  wf_hostgraph H ∧ minimally_extends G₀ m H ∧
  unmarked_hostgraph H ∧ unrooted_hostgraph H
```

The *augmented invariant* addresses the total morphism $\overline{tr}$ by using `minimal_extension`, strengthening the base invariant by requiring a total injective morphism with bijective node mapping:

```
⊢ tc_loop_invariant_total G0 G track_bar ⟺
  wf_hostgraph G ∧ unmarked_hostgraph G ∧
  unrooted_hostgraph G ∧
  minimal_extension G0 track_bar G
```

The two invariants are connected by `extends_morphism`: the total morphism and tr share the node mapping and agree on every edge in tr's domain of definition.

The precondition together with the identity morphism establishes the augmented invariant:

```
⊢ wf_hostgraph G0 ∧ unmarked_hostgraph G0 ∧
  unrooted_hostgraph G0 ⇒
  tc_loop_invariant_total G0 G0 (id_track G0)
```

The ALAP rule `WSPEC_alap`, parameterised by an invariant Q on (graph, morphism) pairs and a termination property R, has three main premises: (P1) P preserves Q on successful runs, i.e. $\{Q\}\,P\,\{Q\}$, with the postcondition composed through the per-iteration track morphism; (P2) when P fails from a Q-satisfying graph G with morphism m, $R\,G\,m$ holds; and (P3) $Q\,G\,m$ implies that m maps nodes (resp. edges) of G into G's own nodes (resp. edges). The two small-step side conditions of Section 5.2 (atomic body failure, no top-level `break`) are vacuous for the rule-set body `link` and discharged in a single step each.

We instantiate `WSPEC_alap` with

$$
\begin{aligned}
Q \;=\; & \lambda\, G\; m.\; \texttt{tc\_loop\_invariant}\ \texttt{G0}\ \texttt{G}\ \texttt{m} \;\wedge \\
& \exists\, \overline{track}.\; \texttt{tc\_loop\_invariant\_total}\ G0\ G\ \overline{track} \;\wedge \\
& \texttt{extends\_morphism}\ m\ \overline{track}, \\
R \;=\; & \lambda\, G\; m.\; \texttt{is\_transitive}\ \texttt{G}.
\end{aligned}
$$

Premise (P1) is established operationally: for every iteration from a Q-satisfying graph, the body either succeeds with both invariants preserved (via morphism composition) or fails atomically:

```
⊢ wf_hostgraph G0 ∧ tc_loop_invariant G0 G m0 ∧
  tc_loop_invariant_total G0 G track_bar ∧
  extends_morphism m0 track_bar ∧
  steps program
    (running (term_rscall {ruleid "link"}),
     [(G,id_track G)]) c ∧ ¬can_step c ⇒
  (∃ H mH.
     c = (final,[(H,mH)]) ∧
     tc_loop_invariant G0 H
       (compose_morphism mH m0) ∧
     ∃ track_bar'.
```

```
      tc_loop_invariant_total G₀ H track_bar' ∧
      extends_morphism (compose_morphism mH m₀)
        track_bar') ∨
  c = (failed,[(G,id_track G)])
```

The successful branch yields the body's WSPEC judgement directly, while the atomic-failure branch discharges the corresponding side condition. The success-preservation argument relies on `link_preserves_total` (Section 6.6).

Premise (P2) requires transitivity upon failure (Section 6.5). Premise (P3) requires that the track morphism maps into the current graph, which the base invariant guarantees.

Once the loop terminates, the ALAP rule combines the augmented invariant with transitivity. The following theorem converts this conjunction into the top-level postcondition:

```
⊢ tc_loop_invariant_total G₀ TC track_bar ∧
  is_transitive TC ⇒
  is_transitive_closure_tracked G₀ track_bar TC
```

Together with the procedure rule and identity-track simplification, this completes the proof of `transitive_closure_correct`, apart from two obligations developed next: transitivity upon termination (Section 6.5) and preservation of the augmented invariant (Section 6.6).

6.5 Proving Transitivity

We now prove premise (P2) of the ALAP rule (Section 6.4): whenever the body `link` fails from a Q-satisfying graph, the current graph is transitive. Since `link!` exits exactly on body failure, this is also what guarantees transitivity at loop termination. The main step is to move from the operational semantics, which talks about rule-set call failure, to a purely graph-theoretic argument.

We introduce the predicate `link_can_apply`, which holds when three distinct nodes form a two-step path whose shortcut edge is absent. The following equivalence connects the operational and graph-theoretic views:

```
⊢ wf_hostgraph G ⇒
  ((∃ G' m'.
      step program
        (running (term_rscall {ruleid "link"}),
         [(G,id_track G)]) (final,[(G',m')])) ⟺
   link_can_apply G)
```

Operational failure of the `link` rule-set call is equivalent to ¬ `link_can_apply G`. This allows us to reason entirely within graph theory for the remainder of this subsection.

The central lemma shows that `link` can be applied to each two-edge path in the graph whose shortcut edge is missing:

```
⊢ wf_hostgraph G ∧ unmarked_hostgraph G ∧
  unrooted_hostgraph G ∧
  (∃ v u w.
     v ∈ G.V ∧ u ∈ G.V ∧ w ∈ G.V ∧ v ≠ w ∧
     has_edge G v u ∧ has_edge G u w ∧
     ¬has_edge G v w) ⇒
  link_can_apply G
```

Equivalently, when the rule cannot fire (¬ `link_can_apply G`), every two-step path already has a shortcut edge (the *2-step closure property*). The proof constructs a concrete prematch and verifies all conditions. We then prove `is_transitive` by induction on path length: the inductive step decomposes the path into a two-step prefix, applies the 2-step closure property to obtain a shortcut edge, and invokes the induction hypothesis on the shorter path.

Combining the equivalence between operational failure and ¬ `link_can_apply` with the inductive argument yields:

```
⊢ wf_hostgraph G ∧ unmarked_hostgraph G ∧
  unrooted_hostgraph G ∧
  ¬(∃ G' m'.
     step program
       (running (term_rscall {ruleid "link"}),
        [(G,id_track G)]) (final,[(G',m')])) ⇒
  is_transitive G
```

This discharges premise (P2) of the ALAP instantiation from Section 6.4.

6.6 Proving Minimal Extension

We now prove premise (P1) of the ALAP rule (Section 6.4): each successful application of `link` preserves the augmented loop invariant. The theorem `link_preserves_total` states that after a rule application, there exists an updated total morphism $\overline{tr}'$ satisfying `tc_loop_invariant_total` and extending the composed track morphism.

Well-formedness follows from `wf_dpo` (Section 3.3); the unmarked and unrooted properties are preserved because the `link` rule introduces only unmarked, unrooted elements. Since `link` preserves all nodes, the track morphism is bijective on nodes, and the composition of bijections establishes the node condition.

Every non-image edge must connect nodes reachable in G_0. The following lemma establishes reachable preimages for every RHS edge:

```
⊢ wf_hostgraph G0 ∧ wf_hostgraph G ∧
  minimally_extends G0 m G ∧
  is_match lhs' rule_link.inf m'' G ∧
  instantiate_rulegraph rule_link.lhs assign m''
    G =
  SOME lhs' ∧
```

```
instantiate_rulegraph rule_link.rhs assign m''
  G =
SOME rhs' ∧ x ∈ rhs'.E ⇒
∃ v0 w0.
  v0 ∈ G0.V ∧ w0 ∈ G0.V ∧
  m.nodemap ' v0 = m''.nodemap ' (rhs'.s ' x) ∧
  m.nodemap ' w0 = m''.nodemap ' (rhs'.t ' x) ∧
  reachable_in G0 v0 w0
```

The shortcut edge is justified because the matched edges have reachable preimages by the induction hypothesis, composing via `reachable_in_trans`. The updated edge map remains injective and preserves sources and targets because `link` only adds edges, so every original edge survives. Together with the structural properties, this establishes `link_preserves_total`.

7 Related Work

Previous work on verifying GP 2 programs includes the Hoare-style proof calculi of Poskitt and Plump [18] and Wulandari and Plump [24], which use variants of monadic second-order logic as assertion languages. Both frameworks come with weakest-precondition constructions, but neither approach has been implemented yet.

Our earlier work with Isabelle/HOL [21] mechanises the double-pushout approach to graph transformation and focuses on proving two classical results of DPO theory. The track morphism of DPO derivations has been introduced in [13]. In the current paper, we extend this concept to transition sequences in the semantics of graph programs with attributed rules. We make the track morphism of a program available in assertions so that specifications can refer to the input/output relation of individual graph elements.

Strecker [22] provides an Isabelle/HOL framework for proving invariants of individual graph-transformation rules (in an ad-hoc formalism), without extending the framework to a full graph programming language.

Noschinski [12] presents a general graph-theory library for Isabelle whose path and reachability infrastructure is similar in spirit to the path theory we build on top of our graph records.

Broadly, the current paper also falls into the area of mechanised semantics for programming languages and systems. For example, CompCert [9] and CakeML [1,10] verify a C compiler and an ML compiler, respectively, and seL4 [8] verifies an operating-system kernel written in C. Our work is narrower in scope but shares the shallow-embedding methodology: a semantic predicate (here `WSPEC`) is defined directly in the prover's logic and serves as both the specification target and the object of the derived proof rules.

8 Conclusion

We have presented the first mechanised formalisation of a complete graph programming language in a theorem prover, covering the full GP 2 language including attributed DPO graph transformation and the operational semantics.

On top of the formalisation, we have established a compositional verification framework based on Hoare-style proof rules, and we have demonstrated it with a case study proving the partial correctness of a transitive-closure program in 50 mechanically verified theorems. The postcondition expresses transitivity and a minimal extension of the input graph via the new concept of a track morphism.

HOL's expressive power lets us quantify over morphisms and specify the trace of individual graph elements throughout entire execution sequences. The underlying framework (operational semantics, DPO construction, track morphisms, Hoare-style calculus) is reusable in further case studies.

In future work, we plan to pursue three directions: (1) verifying programs such as graph colouring and shortest-path algorithms, which will exercise the loop rule on compound bodies and motivate a standard lemma library for discharging the side conditions mechanically; (2) partial proof automation by generating assertions such as loop invariants from annotated programs; and (3) deriving an executable GP 2 semantics inside HOL4 for comparison against the existing C implementation, exploiting the fast computation primitive introduced in [1].

Acknowledgements. We are grateful for the comments of two anonymous reviewers which helped to improve the presentation of this paper.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abrahamsson, O., Myreen, M.O., Norrish, M., Kanabar, H., Pohjola, J.Å.: Fast, verified computation for HOL ITPs. Journal of Automated Reasoning **69**(1), 7 (2025). https://doi.org/10.1007/s10817-025-09719-8
2. Barendregt, H.P., Dekkers, W., Statman, R.: Lambda Calculus with Types. Perspectives in logic, Cambridge University Press (2013). https://doi.org/10.1017/CBO9781139032636
3. Campbell, G., Courtehoute, B., Plump, D.: Fast rule-based graph programs. Science of Computer Programming **214**, 102727 (2022). https://doi.org/10.1016/j.scico.2021.102727, 32 pages
4. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**, 56–68 (1940)
5. Courtehoute, B.: Time and Space Complexity of Rule-Based Graph Programs. Ph.D. thesis, University of York (2023)
6. Courtehoute, B., Plump, D.: A small-step operational semantics for GP 2. In: Graph Computation Models (GCM 2021), Revised Selected Papers. Electronic Proceedings in Theoretical Computer Science, vol. 350, pp. 89–110. Open Publishing Association (2021). https://doi.org/10.4204/EPTCS.350.6

7. Habel, A., Plump, D.: Relabelling in graph transformation. In: Proc. International Conference on Graph Transformation (ICGT 2002). Lecture Notes in Computer Science, vol. 2505, pp. 135–147. Springer (2002). https://doi.org/10.1007/3-540-45832-8_12
8. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009). pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596
9. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
10. Myreen, M.O., Carneiro, M.: GOL in GOL in HOL: Verified Circuits in Conway's Game of Life. In: 16th International Conference on Interactive Theorem Proving (ITP 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 352, pp. 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2025). https://doi.org/10.4230/LIPIcs.ITP.2025.25
11. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (2014). https://doi.org/10.1007/978-3-319-10542-0
12. Noschinski, L.: A graph library for Isabelle. Mathematics in Computer Science **9**(1), 23–39 (2015). https://doi.org/10.1007/s11786-014-0183-z
13. Plump, D.: Hypergraph rewriting: Critical pairs and undecidability of confluence. In: Term Graph Rewriting: Theory and Practice, chap. 15, pp. 201–213. John Wiley (1993), https://www-users.york.ac.uk/~djp10/Papers/wiley.93.pdf
14. Plump, D.: The design of GP 2. In: Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011). Electronic Proceedings in Theoretical Computer Science, vol. 82, pp. 1–16. Open Publishing Association (2012). https://doi.org/10.4204/EPTCS.82.1
15. Plump, D.: From imperative to rule-based graph programs. Journal of Logical and Algebraic Methods in Programming **88**, 154–173 (2017). https://doi.org/10.1016/j.jlamp.2016.12.001
16. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae **118**(1-2), 135–175 (2012). https://doi.org/10.3233/FI-2012-708
17. Poskitt, C.M., Plump, D.: Verifying total correctness of graph programs. In: Graph Computation Models (GCM 2012), Revised Selected Papers. Electronic Communications of the EASST, vol. 61. Berlin Universities Publishing (2013). https://doi.org/10.14279/tuj.eceasst.61.827
18. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: Proc. International Conference on Graph Transformation (ICGT 2014). Lecture Notes in Computer Science, vol. 8571, pp. 33–48. Springer (2014). https://doi.org/10.1007/978-3-319-09108-2_3
19. Poskitt, C.M., Plump, D.: Monadic second-order incorrectness logic for GP 2. Journal of Logical and Algebraic Methods in Programming **130**, 100825 (2023). https://doi.org/10.1016/j.jlamp.2022.100825
20. Slind, K., Norrish, M.: A brief overview of HOL4. In: Proc. Theorem Proving in Higher Order Logics (TPHOLs 2008). Lecture Notes in Computer Science, vol. 5170, pp. 28–32. Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_6
21. Söldner, R., Plump, D.: Formalising the double-pushout approach to graph transformation. Logical Methods in Computer Science **19**(4), 1–42 (2023). https://doi.org/10.46298/lmcs-19(4:20)2023

22. Strecker, M.: Interactive and automated proofs for graph transformations. Mathematical Structures in Computer Science **28**(8), 1333–1362 (2018). https://doi.org/10.1017/S096012951800021X
23. Wulandari, G., Plump, D.: Verifying graph programs with first-order logic. In: Graph Computation Models (GCM 2020), Revised Selected Papers. Electronic Proceedings in Theoretical Computer Science, vol. 330, pp. 181–200. Open Publishing Association (2020). https://doi.org/10.4204/EPTCS.330.11
24. Wulandari, G.S., Plump, D.: Verifying graph programs with monadic second-order logic. In: Proc. International Conference on Graph Transformation (ICGT 2021). Lecture Notes in Computer Science, vol. 12741, pp. 240–261. Springer (2021). https://doi.org/10.1007/978-3-030-78946-6_13