TPLP: Page 1–14. © The Author(s), 2026. Published by Cambridge University Press 2026 1
doi:10.1017/xxxxx

GLP: A Grassroots, Multiagent, Concurrent, Logic Programming Language

EHUD SHAPIRO
London School of Economics and Weizmann Institute of Science

Abstract

Grassroots platforms are distributed systems with multiple instances that can (1) operate independently of each other and of any global resource other than the network, and (2) coalesce into ever larger instances, possibly resulting in a single global instance.

Here, we present Grassroots Logic Programs (GLP), a multiagent concurrent logic programming language designed for the implementation of grassroots platforms. We introduce the language incrementally: We recall the standard operational semantics of logic programs; introduce the operational semantics of Concurrent (single-agent) GLP as a restriction of that of LP; recall the notion of multiagent transition systems and atomic transactions; introduce the operational semantics of multiagent GLP via a multiagent transition system specified via atomic transactions; and prove multiagent GLP to be grassroots. The accompanying programming example is the grassroots social graph—the infrastructure grassroots platform on which all others are based.

With the mathematical foundations presented here: a workstation-based implementation of Concurrent GLP was developed by AI, based on the operational semantics of Concurrent GLP; a distributed peer-to-peer smartphone-based implementation of multiagent GLP is being developed by AI, based on the operational semantics of multiagent GLP; a moded type system for GLP was implemented by AI, to facilitate the specification of GLP programs by human and AI designers, for their programming by AI; all reported in detail in companion papers.

KEYWORDS: concurrent logic programming, grassroots platforms, operational semantics, multiagent transition systems

1 Introduction

A digital platform is *grassroots* (Shapiro 2023a; 2026a) if it can have multiple instances that can (1) operate independently of each other and of any global resource other than the network, and (2) coalesce into ever larger instances, possibly resulting in a single global instance. Grassroots platforms that have been specified (but not yet implemented) include the grassroots social graph (Shapiro 2026a), grassroots social networks (Shapiro 2023b), grassroots cryptocurrencies (Shapiro 2024; Lewis-Pye et al. 2023), and grassroots federation (Shapiro and Talmon 2025). The Scuttlebutt protocol and social network (Kermarrec et al. 2020) is perhaps the sole example of a deployed grassroots platform.

Here, we present Grassroots Logic Programs (GLP), a multiagent, concurrent, logic programming language, designed for the implementation of smartphone-based, serverless, grassroots platforms. First we present Concurrent GLP as a variation on Logic Programs (LP) (Lloyd 1987). Syntactically, GLP adds to LP (1) **Readers:** Each logic variable (now referred to as *writer*) is paired with a *reader*, which is assigned a value only once its paired writer is assigned that value; (2) **Single-Occurrence (SO):** A variable may occur at most once in a goal or a clause; and (3) **Single-Reader Single-Writer (SRSW):** A writer occurs in a clause iff its paired reader occurs in the clause.

The operational semantics of GLP is a restriction of that of LP, preserving computation-as-deduction (Kowalski 1974), yet conjuring both linear logic (Girard 1987) and futures/promises (Friedman and Wise 1976): An assignment to a variable may be produced at most once, via the sole occurrence of a writer (promise), and consumed at most once, via the sole occurrence of its paired reader (future). We illustrate Concurrent GLP via a grassroots social graph with a simulated network switch, in which agents may initiate friendships (bidirectional communication channels) via "cold-calls" as well as introduce mutual friends to each other.

Then, we present multiagent transition systems and their specification via multiagent atomic transactions, and use them to define multiagent GLP (maGLP), with (1) **Multiagents:** Named agents that operate independently while communicating to remote readers assignments made to paired local writers; (2) **Cold-calls:** A means for sending a term with variables to a named agent while retaining their paired variables locally. Cold-calls allow two agents in disconnected components of the social graph to become the owners of paired logic variables. We then present an maGLP implementation of the social graph in which the use of a network-simulator is replaced by maGLP's Cold-call transaction, and social cold-calls among agents are realised by maGLP cold-calls. Lastly, we recall the definition of grassroots platforms and how to prove that a platform specified via atomic transactions is grassroots (Shapiro 2023a; 2026a), and apply these to prove that maGLP is indeed grassroots.

The mathematical foundations presented here have enabled three companion efforts, all developed by AI: a workstation-based implementation of Concurrent GLP, based on its operational semantics; a distributed peer-to-peer smartphone-based implementation of multiagent GLP, based on its operational semantics (Shapiro 2026b); and a moded type system for GLP (Shapiro 2026c), facilitating the specification of GLP programs by human and AI designers for their programming by AI. Example GLP programs in this paper are typed.

Paper outline. Section 2 presents Concurrent GLP: transition systems, GLP extending LP with readers, operational semantics, guards, programming examples, and the grassroots social graph. Section 3 extends to multiagent GLP: multiagent transition systems, the maGLP definition, and the multiagent social graph. Section 4 proves maGLP is grassroots. Section 5 reviews related work, and Section 6 concludes. All proofs are deferred to the appendices.

2 Concurrent GLP

This section presents Grassroots Logic Programs (GLP), a concurrent logic programming language. We begin with transition systems, then define GLP. Standard LP definitions are recalled in Appendix A. We illustrate GLP with programming examples and conclude with the grassroots social graph—the infrastructure platform that all other grassroots platforms build upon.

2.1 Transition Systems

Definition 2.1 (Transition System). A *transition system* is a tuple $TS = (C, c_0, T)$ where C is an arbitrary set of *configurations*, $c_0 \in C$ is a designated *initial configuration*, and $T \subseteq C \times C$ is a *transition relation*, with transitions written $c \to c' \in T$. A transition $c \to c' \in T$ is *enabled* from configuration c. A configuration c is *terminal* if no transitions are enabled from c. A *computation* is a (finite or infinite) sequence of configurations where for each two consecutive configurations (c, c') in the sequence, $c \to c' \in T$. A *run* is a computation starting from c_0, which is *complete* if it is infinite or ends in a terminal configuration.

2.2 GLP: Extending LP with Readers

Grassroots Logic Programs (GLP) extend LP by (1) adding a paired *reader* $X?$ to every "ordinary" logic variable X, now called a *writer*; (2) restricting variables in goals and clauses to have at most a single occurrence (SO); and (3) requiring that a variable occurs in a clause iff its paired variable also occurs in it (single-reader single-writer, SRSW). The result eschews unification in favour of simple term matching, is linear-logic-like (Girard 1987), and is futures/promises-like (Friedman and Wise 1976): each assignment $X := T$ is produced at most once via the sole occurrence of a writer (promise) X, and consumed at most once via the sole occurrence of its paired reader (future) $X?$.

Definition 2.2 (GLP Variables). Let $\mathcal{V}$ denote the set of LP variables (identifiers beginning with uppercase), henceforth called *writers*. Define $\mathcal{V}? = \{X? \mid X \in \mathcal{V}\}$, called *readers*. The set of all GLP variables is $\hat{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}?$. A writer X and its reader $X?$ form a *variable pair*.

GLP terms, unit goals, goals, and clauses are as in standard LP (Appendix A) but defined over the variables in $\hat{\mathcal{V}}$.

Definition 2.3 (Single-Occurrence (SO) Invariant). A term, goal, or clause satisfies the *single-occurrence (SO) invariant* if every variable occurs in it at most once.

Definition 2.4 (GLP Program, Goals). A clause C satisfies the *single-reader/single-writer (SRSW) restriction* if it satisfies SO and a variable occurs in C iff its paired variable also occurs in C. A *GLP program* is a finite set of clauses satisfying SRSW; clauses for the same predicate form a *procedure*. The set of GLP goals $\hat{\mathcal{G}}(P)$ includes all goals over $\hat{\mathcal{V}}$ and the vocabulary of P that satisfy SO.

Example 2.5 (Fair Merge). Consider the quintessential concurrent logic program for fairly merging two streams, written in GLP:

```
Stream ::= [] ;[_|Stream].


procedure merge(Stream?, Stream?, Stream).
merge([X|Xs], Ys, [X?|Zs?]) :- merge(Ys?, Xs?, Zs).
merge(Xs, [Y|Ys], [Y?|Zs?]) :- merge(Xs?, Ys?, Zs).
merge(Xs, [], Xs?).
merge([], Ys, Ys?).
```

and the goal `merge([1,2,3|Xs?],[a,b|Ys?],Zs)`. Both the goal and each clause satisfy SO, and each clause satisfies SRSW. The first two clauses swap inputs in recursive calls, ensuring fairness when both streams are available.

As we shall see (Proposition 2.12), the SO invariant is maintained by the SRSW restriction: reducing a goal satisfying SO with a clause satisfying SRSW results in a goal satisfying SO. The purpose of the SRSW restriction is to prevent multiple writer occurrences racing to assign a variable.

2.3 GLP Operational Semantics

Definition 2.6 (Substitutions and Assignments)**.** A GLP *writer assignment* is a term of the form $X := T$, $X \in \mathcal{V}$, $T \notin \mathcal{V}$, satisfying SO. Similarly, a GLP *reader assignment* is a term of the form $X? := T$, $X? \in \mathcal{V}?$, $T \notin \mathcal{V}$, satisfying SO. A *writers (readers) substitution* σ is the substitution implied by a set of writer (reader) assignments that jointly satisfy SO. Given a writers assignment $X := T$, its *readers counterpart* is $X? := T$, and given a writers substitution σ, its *readers counterpart* $\sigma?$ is the readers substitution defined by $X?\sigma? = X\sigma$.

Definition 2.7 (GLP Renaming, Renaming Apart)**.** A *GLP renaming* is a substitution $\rho : \hat{\mathcal{V}} \rightarrow \hat{\mathcal{V}}$ such that for each $X \in \mathcal{V}$: $X\rho \in \mathcal{V}$ and $X?\rho = (X\rho)?$. Two GLP terms T_1, T_2 *have a variable in common* if for some writer $X \in \mathcal{V}$, T_1 has an occurrence of X or $X?$ and so does T_2. A GLP renaming σ *renames* T' *apart from* T if $T'\sigma$ and T have no variable in common.

Definition 2.8 (Writer MGU)**.** Given two GLP unit goals A and H, a *writer mgu* is a writers substitution σ such that $A\sigma = H\sigma$ and σ is most general among such substitutions.

Note that a writer mgu cannot assign readers.

Definition 2.9 (GLP Goal/Clause Reduction)**.** Given GLP unit goal A and clause C, with H `:-` B being the result of the GLP renaming of C apart from A, the *GLP reduction* of A with C *succeeds with result* (B, σ) if A and H have a writer mgu.

Definition 2.10 (GLP Transition System)**.** Given a GLP program P, an *asynchronous resolvent* over P is a pair (G, σ) where $G \in \hat{\mathcal{G}}(P)$ and σ is a readers substitution.

A transition system $GLP(P) = (\mathcal{C}, c_0, \mathcal{T})$ is a *GLP transition system* over P and initial goal G_0 satisfying SO if:

1. $\mathcal{C}$ is the set of all asynchronous resolvents over P
2. $c_0 = (G_0, \emptyset)$
3. $\mathcal{T}$ is the set of all transitions $(G, \sigma) \rightarrow (G', \sigma')$ satisfying either:
   (a) **Reduce:** there exists unit goal $A \in G$ such that $C \in P$ is the first clause for which the GLP reduction of A with C succeeds with result $(B, \hat{\sigma})$, $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$, and $\sigma' = \sigma \circ \hat{\sigma}?$
   (b) **Communicate:** $\{X? := T\} \in \sigma$, $X? \in G$, $G' = G\{X? := T\}$, and $\sigma' = \sigma$

GLP Reduce differs from LP in (1) the use of a writer mgu instead of a regular mgu and (2) the choice of the first applicable clause instead of any clause. The first is the fundamental use of GLP readers for communication and synchronization. The second compromises on the or-nondeterminism of LP to allow writing fair concurrent programs, such as fair merge above. Note that or-nondeterminism is not completely eliminated, as different scheduling of arrival of assignments on the two input streams of `merge` may result in different orders in its output stream.

The GLP Communicate rule realizes the use of reader/writer pairs for asynchronous communication: it communicates an assignment from its writer to its paired reader.

Monotonicity. Key differences between LP and GLP relate to monotonicity. In LP, if a goal cannot be reduced, it will never be reduced. In GLP, a goal that cannot be reduced now may be reduced in the future: if A and H have an mgu that writes on a reader $X? \in A$, and therefore have no writer mgu at present, another goal that has X may reduce, assigning X, and later $X?$, to a value that will allow A and H to have a writer mgu. Conversely, in LP, if a goal A can be reduced now with some clause H :- B, with a regular mgu of A and H, it may not be reducible in the future due to variables that A shares with other goals being assigned values by reductions of other goals, preventing unification between the instantiated A and H. In GLP, if a goal A can be reduced now (with a writers mgu), it can always be reduced in the future, as the SO invariant ensures that no other goal can assign any writer in A.

Implementation-wise, if a GLP goal A cannot be reduced now, but there is a readers substitution σ such that $A\sigma$ can be reduced, such readers are identified, the goal A *suspends* on these readers, and is rescheduled for another reduction attempt once any of them is assigned.

Despite these differences, GLP adopts the same notions of proper run, successful run, and outcome as LP (Definition Appendix A.4, Appendix A), and has the same notion of logical consequence as LP. Let $/?$ be an operator that replaces every reader by its paired writer.

Proposition 2.11 (GLP Computation is Deduction). *Let $(G_0$:- $G_n)\sigma$ be the outcome of a proper GLP run $\rho : (G_0, \sigma_0) \to \cdots \to (G_n, \sigma_n)$ of GLP(P). Then $(G_0$:- $G_n)\sigma/?$ is a logical consequence of $P/?$.*

We note two additional safety properties of GLP runs.

Proposition 2.12 (SO Preservation). *If the initial goal G_0 satisfies SO, then every goal in the GLP run satisfies SO.*

Proposition 2.13 (Monotonicity). *In any GLP run, if unit goal A can reduce with clause C at step i, then either an instance of A has been reduced by step $j > i$, or an instance of A can still reduce with C at step j.*

Definition 2.14 (Persistent Transition System). A transition system is *persistent* if every enabled transition remains enabled until taken.

Lemma 2.15. *GLP is persistent.*

Definition 2.16 (Fair Run). A complete GLP run is *fair* if every transition that becomes enabled is eventually taken.

The SO invariant of GLP allows eschewing unification in favour of *term matching*: if two terms that jointly satisfy SO are unifiable, their mgu maps each variable in one term to a subterm of the other. Term matching thus performs joint term-tree traversal and collects variable assignments along the way; the detailed definition and table appear in Appendix B.

2.4 Guards

GLP clauses may include *guards*—tests that determine clause applicability.

Definition 2.17 (Guarded Clause). A *guarded clause* has the form H `:-` G `|` B, where H is the head, G is a conjunction of guard predicates, and B is the body. The guard separator "`|`" distinguishes guards from the body, and is interpreted logically as a conjunction. Guard arguments are readers paired to head writers.

Guards have three-valued semantics. Each guard predicate explicitly defines its *success* condition. A guard *suspends* if it does not succeed but some instance of it under a readers substitution would succeed. A guard *fails* if no such instance exists. A guard conjunction succeeds if all members succeed; it suspends if any member suspends and none fail; it fails if any member fails.

Definition 2.9 of a GLP goal/clause reduction is augmented to succeed if the guard also succeeds.

Remark 2.18 (Guards and SRSW). Guard occurrences count toward SRSW satisfaction: if $X?$ occurs in a guard, its paired writer X must occur in the head and $X?$ may additionally occur once in the body. Furthermore, if the success of a guard implies that $X?$ is ground, then $X?$ as well as X may occur multiple times in the clause. Groundness-implying guards include `ground`, `integer`, `number`, `string`, `constant`, arithmetic comparisons (`<`, `>`, `=<`, `>=`, `=:=`, `=\=`), and ground equality (`=?=`). However, `known` and `compound` do not imply groundness.

Remark 2.19 (Anonymous Variables). An *anonymous variable* is any variable whose name begins with `_` (e.g., `_`, `_In?`, `_Out`). Anonymous writers may appear in the head, denoting a fresh writer with no paired reader, so that a value assigned to it is discarded. These provide a controlled exception to the SRSW restriction, allowing a process to abandon an input (e.g. an input stream) they are no longer interested in.

Guard predicates include type tests (`integer`, `number`, `string`, `constant`, `compound`, `ground`, `known`), arithmetic comparisons (`<`, `>`, `=<`, `>=`, `=:=`, `=\=`), and ground equality (`=?=`). The full specification of guards and system predicates appears in Appendix E.

2.5 Programming Examples

Additional programming examples—including stream distribution, association list lookup, channel abstractions, stream tagging, cooperative production, and metainterpreters—appear in Appendix F.

2.6 The Grassroots Social Graph

In our design, the grassroots social graph is the infrastructure platform upon which all other grassroots platforms are built. Nodes represent cryptographically self-identified agents; edges represent authenticated bidirectional channels; connected components arise spontaneously through befriending. We present the social graph as a single-agent GLP program, using a *network switch* to simulate communication between agents, before introducing multiagent GLP in Section 3.

Each agent processes messages from user and network input streams and maintains an outputs list mapping named destinations to output streams. The program supports three protocols: (1) *cold-call befriending*, where agents with no prior shared variables establish friendship by exchanging a response variable through the network; (2) *text messaging* between established friends via named output streams; and (3) *friend-mediated introduction*, where a mutual friend creates a channel pair and sends each half to the respective parties, establishing a direct connection.

A network switch process routes messages between agents in simulation; in deployment, it is replaced by maGLP's Cold-call transaction (Section 3). The detailed walkthrough with code, figures, and a comprehensive scenario involving Alice, Bob, and Charlie appears in Appendix G. The complete program—helper predicates, actor implementations, and boot variants—appears in Appendix H.

3 Multiagent GLP

This section extends GLP to multiple agents. We first recall the notion of multiagent transition systems via multiagent atomic transactions (Shapiro 2021; 2026a), then define multiagent GLP (maGLP) as a transactions-based multiagent transition system, and finally show how the social graph operates in the multiagent setting.

3.1 Multiagent Transition Systems

We assume a potentially infinite set of *agents* Π, but consider only finite subsets of it, so when we refer to a particular set of agents $P \subset \Pi$ we assume P to be nonempty and finite. We use $\subset$ to denote the strict subset relation and $\subseteq$ when equality is also possible.

We use S^P to denote the set S indexed by the set P, and if $c \in S^P$ we use c_p to denote the member of c indexed by $p \in P$. Intuitively, think of such a $c \in S^P$ as an array of cells indexed by members of P with cell values in S.

Definition 3.1 (Local States, Configuration, Transaction, Participants)**.** Given agents $Q \subset \Pi$ and an arbitrary set S of *local states*, a *configuration* over Q and S is a member of $C := S^Q$. An *atomic transaction*, or just *transaction*, over Q and S is any pair of configurations $t = c \to c' \in C^2$ such that $c \neq c'$, with $t_p := c_p \to c'_p$ for any $p \in Q$, and with p being an *active participant* in t if $c_p \neq c'_p$, *stationary participant* otherwise.

Definition 3.2 (Degree)**.** The *degree* of a transaction t (unary, binary, ..., k-ary) is the number of active participants in t, and the *degree* of a set of transactions T is the maximal degree of any $t \in T$.

Definition 3.3 (Multiagent Transition System)**.** Given agents $P \subset \Pi$ and an arbitrary set S of *local states* with a designated *initial local state* $s0 \in S$, a *multiagent transition system* over P and S is a transition system $TS = (C, c0, T)$ with *configurations* $C := S^P$, *initial configuration* $c0 := \{s0\}^P$, and *transitions* $T \subseteq C^2$ being a set of transactions over P and S, with the *degree* of TS being the degree of T.

Rather than specifying a multiagent transition system over a set of agents P directly, we specify it via atomic transactions, which are typically of bounded degree smaller than $|P|$.

Definition 3.4 (Transaction Closure)**.** Let $P \subset \Pi$, S a set of local states, and $C := S^P$. For a transaction $t = (c \to c')$ over local states S with participants $Q \subseteq P$, the *P-closure of t*, $t{\uparrow}P$, is the set of transitions over P and S defined by:

$$t{\uparrow}P := \{t' \in C^2 : \forall q \in Q.(t_q = t'_q) \wedge \forall p \in P \setminus Q.(p \text{ is stationary in } t')\}$$

If R is a set of transactions, each $t \in R$ over some $Q \subseteq P$ and S, then the *P-closure of R*, $R{\uparrow}P$, is the set of transitions over P and S defined by:

$$R{\uparrow}P := \bigcup_{t \in R} t{\uparrow}P$$

Namely, the closure over $P \supseteq Q$ of a transaction t over Q includes all transitions t' over P in which members of Q do the same in t and in t', and the rest remain in their current (arbitrary) state.

Definition 3.5 (Transactions-Based Multiagent Transition System)**.** Given agents $P \subset \Pi$, local states S with initial local state $s0 \in S$, and a set of transactions R, each $t \in R$ over some $Q \subseteq P$ and S, the *transactions-based multiagent transition system* over P, S, and R is the multiagent transition system $TS = (S^P, \{s0\}^P, R{\uparrow}P)$.

In other words, one can fully specify a multiagent transition system over S and P simply by providing a set of atomic transactions over S, each with participants $Q \subseteq P$.

3.2 From GLP to Multiagent GLP

In extending GLP to multiple agents, each agent maintains its own asynchronous resolvent as its local state. The key insight is that GLP's variable pairs provide natural binary communication channels: when agent p assigns a writer X for which the paired reader $X?$ is held by agent q, the assignment $X := T$ must be communicated to q.

A key difference between single-agent GLP and multiagent GLP is in the initial state. In a multiagent transition system all agents must have the same initial local state $s0$ (Definition 3.3). This precludes setting up an initial configuration in which agents share logic variables, as this would imply different initial states for different agents.

We resolve this in two steps. First, we employ only anonymous logic variables "`_`" in the initial local states of agents: Anonymous variables are, on the one hand, syntactically identical, hence allow all initial states to be syntactically identical, and on the other hand represent unique variables, hence semantically all initial goals have unique, local, non-shared variables. The initial state of all agents is the atomic goal `agent(ch(_?,_),ch(_?,_))`, with the first channel serving communication with the user and the second with the network.

Second, the Cold-call transaction enables agents to bootstrap communication by establishing shared variables through the network infrastructure, realizing the cold-call protocol for connecting previously-disconnected agents.

3.3 Multiagent GLP Definition

Definition 3.6 (Multiagent GLP)**.** Given agents $P \subset \Pi$ and GLP program M, the *maGLP transition system* over P and M is the transactions-based multiagent transition system (Definition 3.5) over P, local states being asynchronous resolvents over M, initial local state $s0 = (\{$`agent(ch(_?,_),ch(_?,_))`$\}, \emptyset)$, and the following transactions $c \to c'$:

1. **Reduce** p**:** A unary transaction with participant p where $c_p \to c'_p$ is a GLP Reduce transition (Definition 2.10).
2. **Communicate** p **to** q**:** A transaction with participants $p, q \in P$ where $c_p = (G_p, \sigma_p)$, $c_q = (G_q, \sigma_q)$, $\{X? := T\} \in \sigma_p$, $X?$ occurs in G_q, $c'_p = (G_p, \sigma_p \setminus \{X? := T\})$, and $c'_q = (G_q\{X? := T\}, \sigma_q)$.
3. **Cold-call** p **to** q**:** A binary transaction with participants $p \neq q \in P$ where the network output stream in c_p has a new message $\texttt{msg}(q, X)$, c'_p is the result of advancing the network output stream in c_p, and c'_q is the result of adding $X?$ to the network input stream in c_q.

Note that Communicate may be unary or binary, depending on whether $p = q$. Both Communicate and Cold-call transfer assignments from writers to readers: Communicate operates between agents sharing a paired reader and writer, while Cold-call operates through the network streams established in each agent's initial configuration, enabling the creation of paired variables among previously-disconnected agents. Naturally, Cold-call would be the exception, as once agents share a paired variable they can use it to communicate indefinitely; moreover, an agent with two friends (with which it shares channels) may introduce them to each other, also eschewing the need for a Cold-call.

3.4 The Multiagent Social Graph

The social graph agent code presented in Section 2.6 runs unchanged in maGLP. The only difference is the boot code: the GLP network switch is replaced by maGLP's Cold-call transaction, and each agent runs on a separate isolate (later – separate smartphone). Two boot variants are presented in Appendix H: a headless (UI-less) boot with actors (Appendix H.12) and an interactive UI boot (Appendix H.13).

When Alice executes the cold-call protocol to befriend Bob, her agent sends `msg(bob, intro(alice, Resp))` on the `'_net'` output stream. The Cold-call transaction (Definition 3.6) transfers this message to Bob's network input stream, adding the reader `Resp?` to Bob's resolvent. When Bob accepts, assigning `Resp` to `accept(ch(FIn?, FOut))`, the Communicate transaction transfers this assignment back to Alice.

In contrast, friend-mediated introduction works solely through Communicate transactions: when Bob introduces Alice to Charlie, the channel pair he creates contains readers that are transferred to Alice and Charlie via Communicate, establishing their direct connection. The network switch in the concurrent GLP version (Section 2.6) is thus a faithful simulation of maGLP's Cold-call transaction, allowing single-process testing of multiagent protocols.

3.5 Safety Properties of maGLP

The safety properties established for single-agent GLP in Section 2 extend to maGLP. SO preservation (cf. Proposition 2.12) generalizes directly:

Proposition 3.7 (maGLP SO Preservation)**.** *If the initial goals of all agents satisfy SO, then every goal in every agent's resolvent throughout the run satisfies SO.*

Monotonicity (cf. Proposition 2.13) likewise extends:

Proposition 3.8 (maGLP Monotonicity)**.** *If unit goal A in agent p's resolvent can reduce with clause C at step i, then at any step $j > i$, either A has been reduced or there exists A' in p's resolvent where $A' = A\tau$ for some reader substitution τ, and A' can reduce with C.*

To relate maGLP to deduction, consider the *lifted system L*: the GLP transition system whose resolvent is the union of all agents' local resolvents, whose initial goal includes a `network` goal with channels paired to each agent's network channels, and whose program is M augmented with the GLP definition of `network`.

Proposition 3.9. *maGLP runs correspond to GLP runs of L, and their outcomes are logical consequences of the augmented program.*

Lemma 3.10. *maGLP is persistent.*

Definition 3.11 (maGLP Fair Run)**.** A complete maGLP run is *fair* if every transaction that becomes enabled is eventually taken.

Implementation. The implementation-ready deterministic variants of GLP and maGLP (the latter deterministic at the agent, not system, level), along with formal correctness proofs that they implement their respective specifications, are presented in a companion paper (Shapiro 2026b).

4 Multiagent GLP is Grassroots

This section establishes that maGLP is grassroots, following the framework of (Shapiro 2023a; 2026a). Formal definitions appear in Appendix Appendix C.

Informally, a protocol defined via atomic transactions is *grassroots* if its transactions are *interactive*: for any group of agents P running within a larger group P', it is always possible for an outsider in $P' \setminus P$ to interact with a member of P in a way that leaves an "alien trace" in P's state—a configuration that P could not have produced on its own.

The Cold-call transaction (Definition 3.6) is the fundamental interactive transaction of maGLP. It allows an agent q outside a group P to connect to an agent $p \in P$, establishing a pair of shared variables: a writer held by one agent and its paired reader held by the other. The resulting shared variable constitutes an alien trace in p's state—a reference that could not have arisen from P running alone.

Theorem 4.1. *The maGLP protocol is grassroots.*

The grassroots property of maGLP extends to applications built on top of it, provided they use the cold-call mechanism.

Definition 4.2 (GLP Application)**.** A *GLP application* is a GLP program M together with the maGLP infrastructure. An application *uses cold calls* if agents can execute the Cold-call transaction to establish communication with previously-disconnected agents.

Proposition 4.3. *Any GLP application that uses cold calls is grassroots.*

Corollary 4.4. *The GLP implementation of the grassroots social graph (Section 2.6) is grassroots.*

5 Related Work

Concurrent Logic Programming. GLP belongs to the family of concurrent logic programming (CLP) languages that emerged in the 1980s: Concurrent Prolog (Shapiro 1983), GHC (Ueda 1986), and PARLOG (Clark and Gregory 1986). These languages interpret goals as concurrent processes communicating through shared logical variables, using committed-choice execution with guarded clauses. Shapiro (1989)'s comprehensive survey documents this family and its design space. A key evolution was *flattening*: restricting guards to primitive tests only. Flat Concurrent Prolog (FCP) (Mierowsky et al. 1985) and Flat GHC (Ueda 1986) demonstrated that flat guards suffice for practical parallel programming while dramatically simplifying semantics and implementation.

GLP can be understood as FCP with the Single-Occurrence (SO) invariant. Concurrent Prolog introduced read-only annotations (`?`) distinguishing readers from writers of shared variables, enabling dataflow synchronization. However, read-only unification proved semantically problematic: Levi and Palamidessi (1985) showed it is order-dependent, and Mierowsky et al. (1985) documented non-modularity issues. GHC dispensed with read-only annotations entirely, relying on guard suspension semantics. GLP's SO invariant—requiring that each variable occurs at most once in a goal or clause—resolves these difficulties by ensuring that (1) no races occur on variable assignment, and (2) term matching suffices, eschewing unification entirely. The result is a cleaner semantic foundation while preserving the expressiveness of stream-based concurrent programming. GLP retains logic programming's metaprogramming capabilities (Safra and Shapiro 1988; Lichtenstein and Shapiro 1988; Shapiro 1984b), essential for platform tooling development.

Modes in Concurrent Logic Programming. Mode systems for CLP have a rich history. PARLOG used mode declarations at the predicate level, with input modes enforcing one-way matching. Ueda (1994); Ueda and Morita (1995)'s work on moded Flat GHC is most directly relevant: his mode system assigns polarity to every variable occurrence (positive for input/read, negative for output/write), with the *well-modedness* property guaranteeing each variable is written exactly once. Ueda (2001)'s subsequent linearity analysis identifies variables read exactly once, enabling compile-time garbage collection. GLP enforces single-occurrence universally as a syntactic restriction, whereas Ueda's system guarantees single-writer with single-reader as an optional refinement.

Linear logic and futures/promises. GLP's SO invariant has deep connections to two foundational ideas. Girard (1987)'s linear logic introduced the principle that resources must be used exactly once; GLP's SO invariant enforces this at the variable level, ensuring each assignment is produced exactly once and consumed exactly once. Friedman and Wise (1976)'s futures and promises introduced placeholders for values computed asynchronously; GLP's reader/writer pairs realize this pattern, with the writer as promise (single producer) and reader as future (single consumer). The line of work by Caires and Pfenning (2010); Caires et al. (2016) established deep connections between linear logic and session types; GLP achieves similar resource-discipline properties through syntactic restrictions rather than type systems.

Grassroots platforms. The mathematical foundations for grassroots platforms were established in (Shapiro 2023a), with atomic transactions introduced in (Shapiro 2026a). Grassroots social networks (Shapiro 2023b), cryptocurrencies (Shapiro 2024; Lewis-Pye et al. 2023), and federations (Halpern et al. 2024; Shapiro and Talmon 2025) have been formally specified and proven grassroots. GLP provides the programming language to implement these specifications.

Companion papers. This paper is the founding paper on GLP. Based on it, a companion paper (Shapiro 2026c) developed moded types for GLP, extending LP types (Frühwirth et al. 1991) with modes that capture directionality of communication, enabling typing of interactive partial computations. Another companion paper (Shapiro 2026b) provided implementation-ready operational semantics for both single-agent and multiagent GLP, proved their correctness with respect to the abstract definitions presented here, and provided an independent proof that the implementation is grassroots.

6 Conclusion

We have presented GLP, a multiagent, concurrent logic programming language designed for implementing grassroots platforms. The key contributions are:

1. **GLP Definition:** We defined GLP as an extension of logic programs with reader/writer variable pairs satisfying the Single-Occurrence (SO) invariant and the Single-Reader/Single-Writer (SRSW) restriction, providing asynchronous communication through term matching rather than unification.
2. **Multiagent GLP:** We extended GLP to multiple agents using multiagent transition systems with atomic transactions, defining maGLP with Reduce, Communicate, and Cold-call transactions.
3. **Grassroots Proof:** We proved that maGLP is grassroots by showing it is a transactions-based protocol with interactive transactions.
4. **Grassroots Social Graph:** We demonstrated the expressiveness of GLP by implementing the grassroots social graph protocol, which serves as the infrastructure layer for other grassroots platforms.

Companion papers extend this work: Shapiro (2026c) develops a moded type system for GLP enabling the typing of interactive partial computations, and Shapiro (2026b) provides implementation-ready operational semantics with correctness proofs, supporting the Dart/Flutter implementation for smartphone deployment.

References

CAIRES, L. AND PFENNING, F. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR)* 2010, volume 6269 of *Lecture Notes in Computer Science*, pp. 222–236. Springer.

CAIRES, L., PFENNING, F., AND TONINHO, B. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, *26*, 3, 367–423.

CLARK, K. AND GREGORY, S. 1986. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *8*, 1, 1–49.

FRIEDMAN, D. P. AND WISE, D. S. 1976. The impact of applicative programming on multiprocessing. *Indiana University Computer Science Department Technical Report*,, TR-26.

FRÜHWIRTH, T., SHAPIRO, E., VARDI, M. Y., AND YARDENI, E. Logic programs as types for logic programs. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS)* 1991, pp. 300–309. IEEE Computer Society.

GAIFMAN, H. AND SHAPIRO, E. Fully abstract compositional semantics for logic programs. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* 1989, pp. 134–142.

GIRARD, J.-Y. 1987. Linear logic. *Theoretical Computer Science*, *50*, 1, 1–101.

HALPERN, D., PROCACCIA, A. D., SHAPIRO, E., AND TALMON, N. 2024. Federated assemblies.

KERMARREC, A.-M., LAVOIE, E., AND TSCHUDIN, C. Gossiping with append-only logs in secure-scuttlebutt. In *Proceedings of the 1st international workshop on distributed infrastructure for common good* 2020, pp. 19–24.

KOWALSKI, R. Predicate logic as programming language. In *IFIP congress* 1974, volume 74, pp. 569–574.

LEVI, G. AND PALAMIDESSI, C. The semantics of the read-only variable. In *Proc. Symposium on Logic Programming* 1985, pp. 128–137. IEEE.

LEWIS-PYE, A., NAOR, O., AND SHAPIRO, E. 2023. Grassroots flash: A payment system for grassroots cryptocurrencies. *arXiv preprint arXiv:2309.13191*,.

LICHTENSTEIN, Y. AND SHAPIRO, E. 1988. Concurrent algorithmic debugging. *ACM SIGPLAN Notices*, *24*, 1, 248–260.

LLOYD, J. W. 1987. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition.

MIEROWSKY, C., TAYLOR, S., SHAPIRO, E., LEVY, J., AND SAFRA, M. 1985. On the implementation of flat concurrent prolog. *Proceedings of the 1985 Symposium on Logic Programming*,, 276–286.

SAFRA, S. AND SHAPIRO, E. Meta interpreters for real. In *Concurrent Prolog: Collected Papers* 1988, pp. 166–179. MIT Press.

SHAPIRO, E. 1982. *Algorithmic Program Debugging.* MIT Press.

SHAPIRO, E. 1983. A subset of concurrent prolog and its interpreter. *ICOT Technical Report, TR-003*,.

SHAPIRO, E. 1984a. Alternation and the computational complexity of logic programs. *The Journal of Logic Programming*, *1*a, 1, 19–33.

SHAPIRO, E. Systems programming in concurrent prolog. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages* 1984b, pp. 93–105.

SHAPIRO, E. 1989. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR)*, *21*, 3, 413–510.

SHAPIRO, E. 2021. Multiagent transition systems: Protocol-stack mathematics for distributed computing. *arXiv preprint arXiv:2112.13650*,.

SHAPIRO, E. Grassroots distributed systems: Concept, examples, implementation and applications (brief announcement). In *37th International Symposium on Distributed Computing (DISC 2023). (Extended version: arXiv:2301.04391)* 2023a, pp. 47:1, 47:7, Italy. LIPICS.

Shapiro, E. Grassroots social networking: Serverless, permissionless protocols for twitter/linkedin/whatsapp. In *OASIS '23* 2023b. Association for Computing Machinery.

Shapiro, E. 2024. Grassroots currencies: Foundations for grassroots digital economies. *arXiv preprint arXiv:2202.05619*,.

Shapiro, E. Grassroots platforms with atomic transactions: Social graphs, cryptocurrencies, and democratic federations. In *Proceedings of the 27th International Conference on Distributed Computing and Networking* 2026a, pp. 71–81, arXiv preprint arXiv:2502.11299.

Shapiro, E. 2026b. Implementing grassroots logic programs with monotonic multiagent transition systems. arXiv preprint arXiv:XXXX:XXXXX.

Shapiro, E. 2026c. Types for grassroots logic programs. *arXiv preprint arXiv:2601.17957*,.

Shapiro, E. and Talmon, N. 2025. Grassroots federation: Fair governance of large-scale, decentralized, sovereign digital communities. *arXiv preprint arXiv:2505.02208*,.

Silverman, W., Hirsch, M., Houri, A., and Shapiro, E. The logix system user manual version 1.21. In *Concurrent Prolog: Collected Papers* 1988, pp. 46–77.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog: Advanced Programming Techniques*. MIT press.

Ueda, K. Guarded horn clauses. In *Logic Programming '85* 1986, volume 221 of *Lecture Notes in Computer Science*, pp. 168–179. Springer.

Ueda, K. 1994. Moded flat ghc and its message-oriented implementation technique. *New Generation Computing*, *12*, 4, 337–368.

Ueda, K. 2001. Resource-passing concurrent programming. *Proceedings of TACS 2001*,, 95–126.

Ueda, K. and Morita, M. I/o mode analysis in concurrent logic programming. In *Proceedings of the International Symposium on Theory and Practice of Parallel Programming* 1995, pp. 356–368. Springer.

Appendix A Logic Programs

We recall standard Logic Programs (LP) notions of syntax, most-general unifier (mgu), and semantics via goal reduction.

Definition Appendix A.1 (Logic Programs Syntax)**.** We employ standard LP notions. Let $\mathcal{V}$ denote the set of *variables* (identifiers beginning with uppercase). A *term* is a variable, a constant (numbers, strings, or the empty list `[]`), or a compound term $f(T_1, \ldots, T_n)$ with functor f and subterms T_i. Let $\mathcal{T}$ denote the set of all terms. We use standard list notation: `[X|Xs]` for a list cell, `[X1,...,Xn]` for finite lists. A term is *ground* if it contains no variables.

A *unit goal* is a compound term, also commonly referred to as an *atom*. A *goal* is a multiset of unit goals; the empty goal is written `true`. A *clause* A `:-` B has head A (a unit goal) and body B (a goal); a *unit clause* has empty body. A *logic program* is a finite set of clauses; clauses for the same predicate form a *procedure*. Let $\mathcal{G}(P)$ denote the set of goals over the vocabulary of the program P.

A *substitution* σ is an idempotent function $\sigma : \mathcal{V} \to \mathcal{T}$, a mapping from variables to terms applied to a fixed point. By convention, $\sigma(x) = x\sigma$. Let Σ denote the set of all substitutions. We assume standard notions of instance, ground, renaming, renaming apart, unifier, and most-general unifier (mgu).

Definition Appendix A.2 (LP Goal/Clause Reduction)**.** Given an LP unit goal A and clause C, with H `:-` B being the result of renaming C apart from A, the *LP reduction* of A with C *succeeds with* (B, σ) if A and H have an mgu σ.

Definition Appendix A.3 (Logic Programs Transition System)**.** A transition system $LP(P) = (C, c_0, T)$ is a *Logic Programs transition system* for a logic program P and initial goal $G_0 \in \mathcal{G}(P)$, if $C = \mathcal{G}(P) \times \Sigma$, $c_0 = (G_0, \emptyset)$, and T is the set of all transitions $(G, \sigma) \to (G', \sigma')$ such that for some unit goal $A \in G$ and clause $C \in P$ the LP reduction of A with C succeeds with $(B, \hat{\sigma})$, $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$, and $\sigma' = \sigma \circ \hat{\sigma}$.

LP has two forms of nondeterminism: the choice of $A \in G$, called *and-nondeterminism*, and the choice of $C \in P$, called *or-nondeterminism*, and as such are closely-related to Alternating Turing Machines (Shapiro 1984a).

Definition Appendix A.4 (Proper and Successful Run, Outcome)**.** A run ρ : $(G_0, \sigma_0) \to \cdots \to (G_n, \sigma_n)$ of $LP(P)$ is *proper* if for any $1 \leq i < n$, a variable that occurs in G_{i+1} but not in G_i also does not occur in any G_j, $j < i$. If proper, the *outcome* of ρ is $(G_0$ `:-` $G_n)\sigma_n$. Such a run is *successful* if $G_n = \emptyset$.

The following proposition justifies the computation-as-deduction view of LP (Kowalski 1974), calling a proper LP run a *derivation* and a complete proper run ending in the empty goal a *successful derivation.*

Proposition Appendix A.5 (LP Computation is Deduction)**.** *The outcome* $(G_0$ `:-` $G_n)\sigma$ *of a proper run of* $LP(P)$ *is a logical consequence of* P*.*

The $LP(P)$ transition system allows defining several denotational semantic notions for a program P: (1) the *clause semantics* is the set of all outcomes of all proper runs with an initial most-general unit goal (arguments are distinct variables), closely related to the fully-abstract compositional semantics of LP (Gaifman and Shapiro 1989); (2) the *atom semantics* is the set of all outcomes of all successful derivations with an initial most-general unit goal; (3) the *ground atom semantics* is the standard model-theoretic semantics, the set of ground instances of the atom semantics over the Herbrand universe of P (Lloyd 1987).

Appendix B Term Matching

If two terms T_1 and T_2 that jointly satisfy SO are unifiable with an mgu σ, then σ maps any variable in T_1 to a subterm of T_2 and vice versa. Hence, the SO invariant of GLP allows eschewing unification in favour of *term matching* that performs joint term-tree traversal and collects variable assignments along the way.

Definition Appendix B.1 (Term Matching)**.** Given two terms T_1 and T_2 that jointly satisfy SO, their *term matching* proceeds via the joint traversal of the term-trees of T_1 and T_2, consulting the following table at each pair of joint vertices, where X_1, X_2 denote writers, $X_1?, X_2?$ denote readers, and f/n denotes a non-variable term, a constant when $n = 0$ and a compound term when $n > 0$:

$T_1 \backslash T_2$	Writer X_2	Reader $X_2?$	Term f_2/n_2
Writer X_1	fail	$X_1 := X_2?$	$X_1 := T_2$
Reader $X_1?$	$X_2 := X_1?$	fail	suspend on $X_1?$
Term f_1/n_1	$X_2 := T_1$	fail	fail if $f_1 \neq f_2$ or $n_1 \neq n_2$

The writer mgu is the union of all writer assignments if no *fail* was encountered and the suspension set is empty.

Remark Appendix B.2. In an actual implementation, assuming T_1 is a goal term and T_2 a head term, the case of $X_1?$ and T_2 would add $X_1?$ to the set of readers the goal would suspend upon.

Appendix C Grassroots Definitions

This appendix contains the full definitions for protocols and the grassroots property referenced in Section 4.

C.1 Protocols and the Grassroots Property

A protocol is a family of multiagent transition systems, one for each set of agents $P \subset \Pi$, which share an underlying set of local states $\mathcal{S}$ with a designated initial state $s0$.

Definition Appendix C.1 (Local-States Function)**.** A *local-states function* $S : 2^{\Pi} \to 2^{\mathcal{S}}$ maps every set of agents $P \subset \Pi$ to a set of local states $S(P) \subset \mathcal{S}$ that includes $s0$ and satisfies $P \subset P' \subset \Pi \implies S(P) \subset S(P')$.

Given a local-states function S, we use C to denote configurations over S, with $C(P) := S(P)^P$ and $c0(P) := \{s0\}^P$.

Definition Appendix C.2 (Protocol)**.** A *protocol* $\mathcal{F}$ over a local-states function S is a family of multiagent transition systems that has exactly one transition system $\mathcal{F}(P) = (C(P), c0(P), T(P))$ for every $P \subset \Pi$.

Definition Appendix C.3 (Projection)**.** Let $\emptyset \subset P \subset P' \subset \Pi$. If c' is a configuration over P' then c'/P, the *projection of* c' *over* P, is the configuration c over P defined by $c_p := c'_p$ for every $p \in P$.

Note that in the definition above, c_p, the state of p in c, is in $S(P')$, not in $S(P)$, and hence may include elements "alien" to P, e.g., logic variables shared with $q \in P' \setminus P$.

Being oblivious implies that if a run of $\mathcal{F}(P')$ reaches some configuration c', then anything P could do on their own in the configuration c'/P (with a transition from $T(P)$), they can still do in the larger configuration c' (with a transition from $T(P')$), effectively being oblivious to members of $P' \setminus P$.

Being interactive is a weak liveness requirement: no matter what members of P do, if they run within a larger set of agents it is always the case that they can eventually interact with non-P's in a way that leaves "alien traces" in the local states of P, so that the resulting configuration c'/P could not have been produced by P running on their own.

C.2 Transactions-Based Protocols

Definition Appendix C.4 (Transactions Over a Local-States Function)**.** Let S be a local-states function. A set of transactions R is *over* S if every transaction $t \in R$ is a multiagent transition over Q and $S(Q')$ for some $Q \subseteq Q' \subset \Pi$. Given such a set R and $P \subset \Pi$, $R(P) := \{t \in R : t \text{ is over } Q \text{ and } S(Q'), Q \subseteq Q' \subseteq P\}$.

Definition Appendix C.5 (Transactions-Based Protocol)**.** Let S be a local-states function and R a set of transactions over S. Then a *protocol* $\mathcal{F}$ *over* R *and* S includes for each set of agents $P \subset \Pi$ the transactions-based multiagent transition system $\mathcal{F}(P)$ over P, $S(P)$, and $R(P)$: $\mathcal{F}(P) := (S(P)^P, \{s0\}^P, R(P){\uparrow}P)$.

Definition Appendix C.6 (Interactive Transactions)**.** A set of transactions R over a local-states function S is *interactive* if for every $\emptyset \subset P \subset P' \subset \Pi$ and every configuration $c \in C(P')$ such that $c/P \in C(P)$, there is a computation $(c \xrightarrow{*} c') \subseteq R(P'){\uparrow}P'$ for which $c'/P \notin C(P)$.

C.3 Grassroots Protocols

Definition Appendix C.7 (Oblivious, Interactive, Grassroots)**.** A protocol $\mathcal{F}$ is:

1. *oblivious* if for every $\emptyset \subset P \subset P' \subseteq \Pi$, $T(P){\uparrow}P' \subseteq T(P')$
2. *interactive* if for every $\emptyset \subset P \subset P' \subseteq \Pi$ and every configuration $c \in C(P')$ such that $c/P \in C(P)$, there is a computation $c \xrightarrow{*} c'$ of $\mathcal{F}(P')$ for which $c'/P \notin C(P)$.
3. *grassroots* if it is oblivious and interactive.

Proposition Appendix C.8. *A transactions-based protocol is oblivious.*

Theorem Appendix C.9. *A protocol over an interactive set of transactions is grassroots.*

Appendix D Deferred Proofs

This appendix contains deferred proofs from Sections 2, 3, and 4.

D.1 GLP Proofs

Proposition Appendix D.1 (GLP Computation is Deduction)**.** *Let* $(G_0 \text{ :- } G_n)\sigma$ *be the outcome of a proper GLP run* $\rho : (G_0, \sigma_0) \to \cdots \to (G_n, \sigma_n)$ *of* $GLP(P)$*. Then* $(G_0 \text{ :- } G_n)\sigma/?$ *is a logical consequence of* $P/?$.

Proof. The /? operator replaces every reader $X?$ by its paired writer X, transforming GLP terms into LP terms. We show that the GLP run ρ corresponds to an LP run $\rho/?$ of $LP(P/?)$.

Consider a GLP transition $(G, \sigma) \to (G', \sigma')$:

- *Reduce transition*: Goal A reduces with clause C via writer mgu $\hat{\sigma}$. Applying /?, the clause $C/?$ is an LP clause, and $A/?$ unifies with the head $H/?$ via the mgu $\hat{\sigma}/?$ (since writers map to writers). This is a valid LP reduction.

- *Communicate transition*: A reader $X? \in G$ is replaced by the value T assigned to its paired writer. Under $/?$, both $X?$ and X map to X, so this transition becomes the identity—the variable X is already assigned T in the LP view.

Thus each GLP transition corresponds to zero or one LP transitions, and the GLP run ρ projects to an LP run $\rho/?$ of $LP(P/?)$. By standard LP soundness, the outcome of $\rho/?$ is a logical consequence of $P/?$. □

Proposition Appendix D.2 (SO Preservation)**.** *If the initial goal G_0 satisfies SO, then every goal in the GLP run satisfies SO.*

Proof. By induction on the length of the run. The base case is immediate: G_0 satisfies SO by assumption.

For the inductive step, assume G satisfies SO and consider a transition $(G, \sigma) \to (G', \sigma')$:

- *Reduce transition*: Goal $A \in G$ reduces with clause $C = (H \texttt{ :- } B)$ via writer mgu $\hat{\sigma}$, yielding $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$. Since C satisfies SRSW, it satisfies SO. Since C is renamed apart from G, the variables in B are fresh. The writer mgu $\hat{\sigma}$ maps writers in A to subterms of H and vice versa; by SO of both G and C, each variable is assigned at most once. Applying $\hat{\sigma}$ to $(G \setminus \{A\} \cup B)$ replaces each variable by a term containing fresh variables (from B) or ground subterms. Since no variable in $G \setminus \{A\}$ occurs in A (by SO of G), and no variable in B occurs in G (by renaming apart), G' satisfies SO.
- *Communicate transition*: $G' = G\hat{\sigma}?$ where $\hat{\sigma}? = \{X? := T\}$. Since G satisfies SO, $X?$ occurs at most once in G. Replacing this single occurrence by T (which satisfies SO by Definition 2.6) preserves SO, provided T shares no variables with the rest of G. By the proper run condition, variables in T are fresh, so G' satisfies SO.

□

Proposition Appendix D.3 (Monotonicity)**.** *In any GLP run, if unit goal A can reduce with clause C at step i, then either an instance of A has been reduced by step $j > i$, or an instance of A can still reduce with C at step j.*

Proof. Suppose goal A can reduce with clause C at step i, meaning the writer mgu of A and the head H of (a renaming of) C succeeds. Consider what can change between steps i and $j > i$:

- *Reduce transitions on other goals*: These do not affect A directly. By SO, no other goal shares a writer with A, so no other reduction can assign a writer in A.
- *Communicate transitions*: These assign readers, not writers. A communicate transition $X? := T$ may instantiate a reader $X? \in A$, yielding $A' = A\{X? := T\}$. We show A' can still reduce with C:
  The original writer mgu succeeded, meaning at position p where $X?$ occurred in A, either (a) H had a writer Y at position p, yielding assignment $Y := X?$, or (b) H had a reader $Y?$ at position p, which would have caused failure (reader-reader), contradiction. In case (a), after the communicate transition, A' has T at position p. The clause C (renamed apart for A') has a fresh writer Y' at position p. The writer mgu now yields $Y' := T$, which succeeds.
- *Reduce transition on A*: If A itself is reduced at some step k with $i < k \leq j$, then an instance of A has been reduced, satisfying the proposition.

Thus, if A has not been reduced by step j, the (possibly instantiated) goal A' at step j can still reduce with a fresh renaming of C. □

Lemma Appendix D.4. *GLP is persistent.*

Proof. For Reduce: if a Reduce transition is enabled for goal A in configuration (G, σ_r), it remains enabled in any configuration (G, σ_r') with $\sigma_r \subseteq \sigma_r'$, since other Reduce transitions assign disjoint writers (by SO) and Communicate transitions only instantiate readers. For Communicate: applying $\{X? := T\} \in \sigma$ to $X? \in G$ remains enabled since Reduce transitions do not remove assignments from σ and other Communicate transitions apply different assignments. □

D.2 maGLP Proofs

Proposition Appendix D.5 (maGLP SO Preservation)**.** *If the initial goals of all agents satisfy SO, then every goal in every agent's resolvent throughout the run satisfies SO.*

Proof. Identical to single-agent GLP (Proposition 2.12), substituting "agent p's resolvent" for "resolvent" and noting that Communicate and Cold-call transitions transfer only reader assignments, which do not affect SO. □

Proposition Appendix D.6 (maGLP Monotonicity)**.** *If unit goal A in agent p's resolvent can reduce with clause C at step i, then at any step $j > i$, either A has been reduced or there exists A' in p's resolvent where $A' = A\tau$ for some reader substitution τ, and A' can reduce with C.*

Proof. Identical to single-agent GLP (Proposition 2.13), substituting "agent p's resolvent" for "resolvent" and noting that Reduce transitions operate locally within each agent whilst Communicate and Cold-call transitions preserve reducibility through reader assignment transfer. □

Lemma Appendix D.7. *maGLP is persistent.*

Proof. Reduce persistence follows from GLP persistence (Lemma 2.15). Communicate and Cold-call transactions remain enabled since Reduce transitions do not remove reader assignments or network messages. □

D.3 Grassroots Proofs

Proposition Appendix D.8. *A transactions-based protocol is oblivious.*

Proof. Let $\mathcal{F}$ be a protocol over transactions R and local-states function S. Let $P \subset P' \subset \Pi$. We must show $T(P){\uparrow}P' \subseteq T(P')$.

By Definition Appendix C.5, $T(P) = R(P){\uparrow}P$. A transition $t' \in T(P){\uparrow}P'$ is derived from some $t \in R(P)$ by extending to agents in P' who remain stationary. Since $t \in R(P)$ has participants $Q \subseteq P \subset P'$ and is over $S(Q')$ for some $Q' \subseteq P \subset P'$, we have $t \in R(P')$. Hence $t' \in R(P'){\uparrow}P' = T(P')$. □

Theorem Appendix D.9. *A protocol over an interactive set of transactions is grassroots.*

Proof. Let $\mathcal{F}$ be a protocol over a set of transactions R, where R is interactive. Since $\mathcal{F}$ is a transactions-based protocol then, according to Proposition Appendix C.8, $\mathcal{F}$ is oblivious. And since $\mathcal{F}$ is over an interactive set of transactions, $\mathcal{F}$ is interactive. Therefore, by Definition Appendix C.7, $\mathcal{F}$ is grassroots. □

Theorem Appendix D.10. *The maGLP protocol is grassroots.*

Proof. By Theorem Appendix C.9, it suffices to show that maGLP is a transactions-based protocol with interactive transactions.

maGLP is transactions-based: By Definition 3.6, maGLP is defined via a set of transactions (Reduce, Communicate, Cold-call) over a local-states function (asynchronous resolvents) with a common initial state. Hence maGLP is a transactions-based protocol (Definition Appendix C.5).

maGLP transactions are interactive: Let $\emptyset \subset P \subset P' \subseteq \Pi$ and let $c \in C(P')$ be a configuration such that $c/P \in C(P)$. We must show there exists a computation $c \xrightarrow{*} c'$ of maGLP(P') such that $c'/P \notin C(P)$.

Since $c/P \in C(P)$, the local states of agents in P contain no variables shared with agents in $P' \setminus P$—all their reader/writer pairs are "internal" to P.

Consider any agent $p \in P$ and any agent $q \in P' \setminus P$. Agent p can execute a Reduce transaction that sends a message $\mathtt{msg}(q, X)$ on its network output stream, where X is a fresh writer. Then the Cold-call transaction from p to q adds $X?$ to agent q's network input stream. Agent q can then execute Reduce transactions that assign $X?$ some term T, creating a reader assignment $X? := T$. Finally, the Communicate transaction from q to p applies this assignment to p's resolvent.

The result is that agent p's local state now contains a term T that originated from agent $q \in P' \setminus P$. This constitutes an "alien trace"—the configuration c'/P contains references to variables or terms that could not have been produced by agents in P running alone. Hence $c'/P \notin C(P)$.

Since such a computation exists for any valid configuration c, the maGLP transactions are interactive (Definition Appendix C.6).

By Theorem Appendix C.9, maGLP is grassroots. □

Proposition Appendix D.11. *Any GLP application that uses cold calls is grassroots.*

Proof. A GLP application using cold calls is a restriction of maGLP to a specific program M. The Cold-call transaction remains available, as cold calls are used. The proof of Theorem 4.1 relies only on the availability of the Cold-call transaction to establish interactivity. Since cold calls provide this mechanism, the application inherits the interactive property.

The application is transactions-based by construction (being a restriction of maGLP). By Proposition Appendix C.8, it is oblivious. By the interactivity argument above, it is interactive. Therefore, by Definition Appendix C.7, it is grassroots. □

Corollary Appendix D.12. *The GLP implementation of the grassroots social graph (Section 2.6) is grassroots.*

Proof. The social graph uses cold calls for initial contact between disconnected agents. By Proposition 4.3, it is grassroots. □

Appendix E Guards and System Predicates

Guards and system predicates extend GLP programs with access to the GLP runtime state, operating system and hardware capabilities.

Guard predicates. Guards provide read-only access to the runtime state of GLP computation. A guard appears after the clause head, separated by `|`, and must be satisfied for the clause to be selected. The "Ground" column indicates whether success of the guard implies that the argument is ground; such guards permit multiple occurrences of the reader in the clause body (Remark 2.18).

Guard	Signature	Ground
`integer`	`procedure integer(Integer?).`	yes
`number`	`procedure number(Number?).`	yes
`string`	`procedure string(String?).`	yes
`atom`	`procedure atom(String?).`	yes
`constant`	`procedure constant(Constant?).`	yes
`compound`	`procedure compound(_?).`	no
`is_list`	`procedure is_list(Stream?).`	no
`ground`	`procedure ground(_?).`	yes
`known`	`procedure known(_?).`	no
`unknown`	`procedure unknown(_?).`	no
`writer`	`procedure writer(_?).`	no
`reader`	`procedure reader(_?).`	no*
`no_readers`	`procedure no_readers(_?).`	no
`otherwise`	(no arguments)	—
`=?=`	`procedure =?=(_?, _?).`	yes (both)

*`reader(X)` is non-monotonic: it succeeds if `X` is an unassigned reader, but once `X` is assigned it fails.

`otherwise` succeeds if all previous clauses for this procedure failed (or suspended).

`=?=` succeeds if both arguments are ground and equal. For example, `f(a,X?) =?= f(b,Z?)` fails (not suspends) because the ground subterms `a` and `b` already differ. Note that the implementation checks terms left-to-right and does not guarantee early failure detection; `f(X?,a) =?= f(Y?,b)` may suspend rather than fail.

Arithmetic comparison guards. Arithmetic comparison guards evaluate their arguments as arithmetic expressions and compare the results. Success implies both arguments are ground.

Guard	Signature
`<`	`procedure <(Exp?, Exp?).`
`>`	`procedure >(Exp?, Exp?).`
`=<`	`procedure =<(Exp?, Exp?).`
`>=`	`procedure >=(Exp?, Exp?).`
`=:=`	`procedure =:=(Exp?, Exp?).`
`=\=`	`procedure =\=(Exp?, Exp?).`

Monotonicity and implications. `ground/1`, `no_readers/1`, and `known/1` are monotonic. `ground(X)` implies both `no_readers(X)` and `known(X)`, which do not imply each other: `no_readers(X)` succeeds but `known(X)` fails for an unassigned writer; `no_readers(f(X?))` suspends but `known(f(X?))` succeeds.

Defined guard predicates. To support abstract data types and cleaner code organization, GLP provides for user-defined guards via unit clauses. A unit clause `p(T1,...,Tn).` defines a guard predicate; the call `p(S1,...,Sn)` in guard position is unfolded to the term matching of `T1` with `S1`, ..., `Tn` with `Sn`. For example, the equality guard is defined by the clause `X = X.`, so the guard `A = B` unfolds to matching both `A` and `B` against the same variable `X`.

Channel operations (`new_channel`, `send`, `receive`) are defined as guard predicates; see Appendix F.

System predicates. System predicates execute atomically with goal/clause reduction and provide access to underlying runtime services:

Predicate	Description
`=(_, _?).`	Unification
`=..(_, Stream?).`	Term composition: constructs term from list
`..=(Stream, _?).`	Term decomposition: decomposes term into list
`evaluate(Exp?,Result).`	Evaluates ground arithmetic expressions
`current_time(T).`	Provides system timestamps
`variable_name(X,Name).`	Returns a unique identifier for variable `X` and its pair

The `=` predicate performs unification: `X? = X` succeeds, assigning both sides. The `=..` predicate composes a term from a list: `Term =.. [Functor, Arg1, ..., ArgN]?` constructs `Term` as `Functor(Arg1, ..., ArgN)`. The `..=` predicate decomposes a term into a list: `List ..= Term?` produces `[Functor, Arg1, ..., ArgN]` from `Term`. Both follow the convention of input on the right (reader) and output on the left (writer).

Arithmetic evaluation in assignments. Arithmetic expressions are defined by the following clause:

```
X? := E :- ground(E) | evaluate(E?,X).
```

Ensuring the expression is ground before calling the system evaluator, maintaining program safety whilst providing convenient notation for mathematical computations.

Appendix F Additional Programming Techniques

This appendix presents GLP programs that were referenced in the main text, as well as additional programs that demonstrate the language's capabilities.

F.1 Stream Distribution and Lookup

Example Appendix F.1 (Stream Distribution)**.** Broadcasting to multiple consumers uses the `ground` guard to enable safe replication:

```
procedure distribute(Stream?, Stream, Stream).
```

```
distribute([X|Xs], [X?|Ys1?], [X?|Ys2?]) :- ground(X?) |
    distribute(Xs?, Ys1, Ys2).
distribute([], [], []).
```

Since `X?` is ground, its multiple occurrences in the head do not violate SRSW.

Example Appendix F.2 (Lookup in Association List). `procedure lookup(_?, _?, _).`

```
lookup(Key, [(K, Value)|_], Value?) :- Key? =?= K? | true.
lookup(Key, [_|Rest], Value?) :- otherwise | lookup(Key?, Rest?, Value).
```

The `=?=` guard tests ground equality; `otherwise` succeeds when no prior clause applies.

F.2 Channel Abstractions

Bidirectional channels are fundamental to concurrent communication in GLP. We represent a channel as the term `ch(In?,Out)` where `In?` is the input stream reader and `Out` is the output stream writer. The following predicates encapsulate channel operations and are defined as guard predicates through unit clauses:

Program 1: Channel Operations

```
send(X,ch(In,[X?|Out?]),ch(In?,Out)).
receive(X?,ch([X|In],Out?),ch(In?,Out)).
new_channel(ch(Xs?,Ys),ch(Ys?,Xs)).
```

The `send` predicate adds a message to the output stream, `receive` removes a message from the input stream, and `new_channel` creates a pair of channels where each channel's input is paired with the other's output. When used as guards in clause heads, these predicates enable readable code that abstracts the underlying stream mechanics:

Program 2: Stream-Channel Relay

```
procedure relay(Stream?, Stream, Channel?).
relay(In,Out?,Ch) :-
    In?=[X|In1], send(X?,Ch?,Ch1) | relay(In1?,Out,Ch1?).
relay(In,Out?,Ch) :-
    receive(X,Ch?,Ch1), Out=[X?|Out1?] | relay(In?,Out1,Ch1?).
```

The relay reads from its input stream and sends to the channel in the first clause, while the second clause receives from the channel and writes to the output stream. The channel state threads through the recursive calls, maintaining the bidirectional communication link.

F.3 Stream Tagging for Source Identification

When multiple input streams merge into a single stream, the source identity of each message is lost. Stream tagging preserves this information by wrapping each message with its source identifier:

Program 3: Stream Tagging

```
procedure tag_stream(_?, Stream?, Stream).
tag_stream(Name, [M|In], [msg(Name?, M?)|Out]) :-
    tag_stream(Name?, In?, Out?).
```

```
tag_stream(_, [], []).
```

The procedure recursively processes the input stream, wrapping each message `M` in a `msg(Name, M)` term that includes the source name. The tagged stream can then be safely merged with other tagged streams while preserving source information, essential for multiplexed message processing where receivers must determine message origin.

F.4 Stream Observation

For non-ground data requiring observation without consumption, the observer technique forwards communication bidirectionally while producing a replicable audit stream:

Program 4: Concurrent Observer

```
procedure observe(_?, _, _).
observe(X?, Y, Z) :- observe(Y?, X, Z).
observe(X, X?, X?) :- ground(X) | true.
observe(Xs, [Y1?|Ys1?], [Y2?|Ys2?]) :-
    Xs? = [X|Xs1] |
    observe(X?, Y1, Y2),
    observe(Xs1?, Ys1, Ys2).
```

F.5 Cooperative Stream Production

While the single-writer constraint prevents competitive concurrent updates, GLP enables sophisticated cooperative techniques where multiple producers coordinate through explicit handover:

Program 5: Cooperative Producers

```
procedure producer_a(_?).
producer_a(control(Xs,Next)) :-
    produce_batch_a(Xs,Xs1,Done),
    handover(Done?,Xs1,Next).

procedure producer_b(_?).
producer_b(control(Xs,Next)) :-
    produce_batch_b(Xs,Xs1,Done),
    handover(Done?,Xs1,Next).

handover(done,Xs,control(Xs,Next)).

produce_batch_a([a,b,c|Xs],Xs,done).
produce_batch_b([d,e,f|Xs],Xs,done).
```

The `control(Xs,Next)` term encapsulates both the stream tail writer and the continuation for transferring control, enabling round-robin production, priority-based handover, or dynamic producer pools.

These examples demonstrate GLP as a powerful concurrent programming language where reader/writer pairs provide natural synchronization, the single-writer constraint ensures race-free concurrent updates, and stream-based communication enables scalable concurrent architectures.

F.6 Network Switch

For three agents `p, q ,r` and three channels with them `Chp, Chq, Chr`, it is initialized with `network((p,Chp?),(q,Chq?),(r,Chr?))`.

Program 6: 3-Way Network Switch

```
procedure network(_?, _?, _?).

% P to Q forwarding
network((P,ChP),(Q,ChQ),(R,ChR)) :-
    ground(Q), receive(ChP?,msg(Q,X),ChP1), send(ChQ?,X?,ChQ1) |
    network((P,ChP1?),(Q,ChQ1?),(R,ChR?)).

% P to R forwarding
network((P,ChP),(Q,ChQ),(R,ChR)) :-
    ground(R), receive(ChP?,msg(R,X),ChP1), send(ChR?,X?,ChR1) |
    network((P,ChP1?),(Q,ChQ?),(R,ChR1?)).

% Q to P forwarding
network((P,ChP),(Q,ChQ),(R,ChR)) :-
    ground(P), receive(ChQ?,msg(P,X),ChQ1), send(ChP?,X?,ChP1) |
    network((P,ChP1?),(Q,ChQ1?),(R,ChR?)).

% Q to R forwarding
network((P,ChP),(Q,ChQ),(R,ChR)) :-
    ground(R), receive(ChQ?,msg(R,X),ChQ1), send(ChR?,X?,ChR1) |
    network((P,ChP?),(Q,ChQ1?),(R,ChR1?)).

% R to P forwarding
network((P,ChP),(Q,ChQ),(R,ChR)) :-
    ground(P), receive(ChR?,msg(P,X),ChR1), send(ChP?,X?,ChP1) |
    network((P,ChP1?),(Q,ChQ?),(R,ChR1?)).

% R to Q forwarding
network((P,ChP),(Q,ChQ),(R,ChR)) :-
    ground(Q), receive(ChR?,msg(Q,X),ChR1), send(ChQ?,X?,ChQ1) |
    network((P,ChP?),(Q,ChQ1?),(R,ChR1?)).
```

F.7 Replication of Non-Ground Terms

While the main text demonstrated distribution of ground terms to multiple consumers, many applications require replicating incrementally-constructed terms that may contain uninstantiated readers. The following replicator procedure handles nested lists and other structured terms, provided the input contains no writers. This technique suspends when encountering readers and resumes as values become available, enabling incremental replication of partially instantiated data structures.

Program 7: Non-Ground Term Replicator

```
replicate(X, X?,..., X?) :-
    ground(X) | true.                          % Ground terms can be shared
replicate(Xs, [Y1?|Ys1?],..., [Yn?|Ysn?]) :-   % List recursion on both parts
    Xs? = [X|Xs1] |
    replicate(X?, Y1,..., Yn),
    replicate(Xs1?, Ys1,..., Ysn).
```

The replicator operates recursively on list structures, creating multiple copies that maintain the same incremental construction behavior as the original. When the input list head becomes available, all replica heads receive the replicated value simultaneously. This technique extends naturally to tuples through conversion to lists of arguments, enabling replication of arbitrary term structures that contain readers but no writers.

F.8 Interlaced Streams as Distributed Blocklace

A blocklace represents a partially-ordered generalization of the blockchain where each block contains references to multiple preceding blocks, forming a directed acyclic graph. This structure maintains the essential properties of blockchains while enabling concurrent block creation without consensus. GLP's concurrent programming model naturally realizes blocklace structures through interlaced streams, where multiple concurrent processes maintain individual streams while observing and referencing each other's progress.

Program 8: Interlaced Streams (Blocklace)

```
% Three agents maintaining interlaced streams
% Initial goal:
%   p(streams(P_stream, [Q_stream?, R_stream?])),
%   q(streams(Q_stream, [P_stream?, R_stream?])),
%   r(streams(R_stream, [P_stream?, Q_stream?]))

procedure streams(Stream, _?).
streams(MyStream, Others) :-
    produce_payloads(Payloads),
    interlace(Payloads?, MyStream, [], Others?).

interlace([Payload|Payloads], [block(Payload?,Tips?)|Stream?], PrevTips, Others) :-
    collect_new_tips(Others?, Tips, Others1),
    interlace(Payloads?, Stream, Tips?, Others1?).
interlace([], [], _, _).

% Using reader(X) to identify fresh tips not yet incorporated
collect_new_tips([[Block|Bs]|Others], [Block?|Tips?], [Bs?|Others1?]) :-
    reader(Bs) |  % Bs unbound means Block is the current tip
    collect_new_tips(Others?, Tips, Others1).
collect_new_tips([[B|Bs]|Others], Tips?, [[Bs]?|Others1?]) :-
```

```
    % Skip B as it's already been referenced
    collect_new_tips([[Bs]?|Others?], Tips, Others1).
collect_new_tips([], [], []).
```

Each concurrent process maintains its own stream of blocks containing application payloads and references to the most recent blocks observed from other processes. The 'reader(X)' guard predicate identifies unprocessed blocks by detecting unbound tail variables, enabling each process to reference exactly those blocks it has not previously incorporated. This creates a distributed acyclic graph structure where the partial ordering reflects the causal relationships between blocks produced by different processes.

The interlaced streams technique demonstrates how GLP's reader/writer synchronization mechanism naturally implements sophisticated distributed data structures. The resulting blocklace provides eventual consistency guarantees similar to CRDTs while maintaining the integrity and non-repudiation properties of blockchain structures. This technique has applications in distributed consensus protocols, collaborative editing systems, and Byzantine fault-tolerant dissemination networks.

F.9 Metainterpreters

Program development is essentially a single-agent endeavour: The programmer trying to write and debug a GLP program. As in Concurrent Prolog, a key strength of GLP is metainterpretation: The ability to write GLP interpreters with various functions in GLP. This allows writing a GLP program development environment and a GLP operating system within GLP itself (Sterling and Shapiro 1994; Safra and Shapiro 1988; Shapiro 1982; Lichtenstein and Shapiro 1988; Silverman et al. 1988), as well as writing a GLP operating system in GLP (Shapiro 1984b). These two scenarios are the focus of this section: a programmer developing a program and running it with enhanced metainterpreters that support the various needs of program development, and an operating system written in GLP that supports the execution, monitoring and control of GLP programs.

Plain metainterpreter. Next we show a plain GLP metainterpreter. It follows the standard granularity of logic programming metainterpreters, using the predicate `reduce` to encode each program clause. This approach avoids the need for explicit renaming and, in the case of concurrent logic programs such as GLP also guard evaluation, while maintaining explicit goal reduction and body evaluation. The encoding is such that if in a call to `reduce` a given goal unifies with its first argument then the body is returned in its second argument. Here we show it together with a `reduce` encoding of `merge`.

Program 9: GLP plain metainterpreter

```
procedure run(_?).
run(true).  % halt
run((A,B)) :- run(A?), run(B?). % fork
run(A) :- known(A) | reduce(A?,B), run(B?) %  reduce

reduce(merge([X|Xs],Ys,[X?|Zs?]),merge(Xs?,Ys?,Zs)).
reduce(merge(Xs,[Y|Ys],[Y?|Zs?]),merge(Xs?,Ys?,Zs)).
reduce(merge([],[],[]),true).
```

For example, when called with an initial goal:

```
run((merge([1,2,3],[4,5],Xs), merge([a,b],[c,d,e],Ys), merge(Xs?,Ys?,Zs)).
```

after two forks using the second clause of `run`, its goal would become:

```
run((merge([1,2,3],[4,5],Xs)), run(merge([a,b],[c,d,e],Ys)), run(merge(Xs?,Ys?,Zs)).
```

and its finite run would produce some merge of the four input lists.

Fail-safe metainterpreter. The operational semantics of Logic Programs and GLP specifies that a run is aborted once a goal fails. Following this rule would make impossible the writing in GLP of a metainterpreter that identifies and diagnoses failure. The following metainterpreter addresses this by assuming that the representation of the interpreted program ends with the clause:

```
reduce(A,failed(A)) :- otherwise | true.
```

Returning the failed goal `A` as the term `failed(A)` for further processing, the simplest being just reporting the failure, as in the following metainterpreter:

Program 10: GLP fail-safe metainterpreter
```
procedure run(_?, Stream).
run(true,[]).  % halt
run((A,B),Zs?) :- run(A?,Xs), run(B?,Ys), merge(Xs?,Ys?,Zs). % fork
run(fail(A),[fail(A?)]).    % report failure
run(A,Xs?) :- known(A) | reduce(A?,B), run(B?,Xs) %  reduce
```

Failure reports can be used to debug a program, but do not prevent a faulty run from running forever.

Metainterpreter with run control. Here we augment the metainterpreter with run control, via which a run can be suspended, resumed, and aborted. As control messages are intended to be ground, the control stream of a run can be distributed to all metainterpreter instances that participate in its execution.

Program 11: GLP metainterpreter with run control
```
procedure run(_?, Stream?).
run(true,_).  % halt
run((A,B),Cs) :- distribute(Cs?,Cs1,Cs2), run(A?,Cs1?), run(B?,Cs1). % fork
run(A,[suspend|Cs]) :- suspended_run(A,Cs?).   % suspend
run(A,Cs) :- known(A) |        % reduce
        distribute(Cs?,Cs1,Cs2), reduce(A?,B,Cs1?), run(B?,Xs,Cs2?).

procedure suspended_run(_?, Stream?).
suspended_run(A,[resume|Cs]) :- run(A,Cs?).
suspended_run(A,[abort|Cs]).
```

The metainterpreter suspends reductions as soon as the control stream is assigned `[suspend|Cs?]`, upon which the run can be resumed or aborted by assigning `Cs` accordingly. Combining Programs F.9 and F.9 would allow the programmer to abort the run as soon as a goal fails. But we wish to introduce additional capabilities before integrating them all.

Termination detection. The following metainterpreter allows the detection of the termination of a concurrent GLP program. It uses the 'short-circuit' technique, in which a chain of paired variables extends while goals fork, contracts when goals terminate, and closes when all goals have terminated.

Program 12: GLP termination-detecting metainterpreter

```
procedure run(_?, _?, _).
run(true,L,L?).  % halt
run((A,B),Cs,L,R?) :-  run(A?,Cs1?,L?,M), run(B?,Cs1,M?,R). % fork
run(A,L,R?) :- known(A) |        % reduce
        reduce(A?,B,Cs1?), run(B?,Xs,Cs2?,L?,R).
```

When called with `run(A,done,R)`, the reader `R?` will be assigned `done` iff the run terminates.

Collecting a snapshot of an aborted run. The short-circuit technique can be used to extend the metainterpreter with run control to collect a snapshot of the run, if aborted before termination. Upon abort, the resolvent is passed from left to right in the short circuit, with each metainterpreter instance adding their interpreted goal to the growing resolvent. We only show the `suspended_run` procedure:

Program 13: GLP metainterpreter with run control and snapshot collection

```
procedure suspended_run(_?, Stream?, _, _).
suspended_run(A,[resume|Cs],L,R?) :- run(A,Cs?,L?,R).
suspended_run(A,[abort|_],L,[A?|L?]).
```

When called with `run(A,Cs?,[],R)`, if `Cs` is assigned `[suspend,abort]`, the reader `R?` will be assigned the current resolvent of the run (which could be empty if the run has already terminated before).

Note that taking a snapshot of a suspended run and then resuming it requires extra effort, as two copies of the goal are needed, a 'frozen' one for the snapshot, and a 'live' one to continue the run. Addressing this is necessary for interactive debugging, to allow a developer to watch a program under development as it runs. We discuss it below.

Producing a trace of a run. Tracing a run of a program and then single-stepping through its critical sections are basic debugging techniques, but applying them to concurrent programs is both difficult and less useful due to their nondeterminism. Here is a metainterpreter that produces a trace of the run, which can then be used by a retracing metainterpreter to single-step through the very same run, making the same nondeterministic scheduling choices. It assumes that each program clause `A:- D | B` is represented by a unit clause `reduce(A,B,I) :- G | true`, with I being the serial number of the clause in the program.

Program 14: GLP a tracing metainterpreter

```
procedure run(_?, _).
run(true,true).  % halt
run((A,B),(TA?,TB?)) :- run(A?,TA), run(B?,TB). % fork
run(A,((I?:Time?):-TB?)) :- known(A) |
    time(Time), reduce(A?,B,I), run(B?, TB).
```

As another example, here is a GLP metainterpreter, inspired by (Shapiro 1984b), that can suspend, resume, and abort a GLP run and produce a dump of the processes of the aborted run. It employs the guard predicate `otherwise`, which succeeds if and only if all previous clauses in the procedure fail (as opposed to suspend). This enables default case handling when no other clause applies.

Program 15: GLP metainterpreter with runtime control

```
procedure run(_?, Stream?, _?, _).
run(true,Cs,L?,L).  % halt and close the dump
```

```
run((A,B),Cs,L?,R) :- run(A?,Cs?,L,M?), run(B?,Cs?,M,R?). % fork
run(A,Cs,L?,R) :- otherwise, unknown(Cs) | reduce(A?,B), run(B?,Cs,L,R?) %  reduce
run(A,[abort|Cs],[A?|R?],R). % abort and dump
run(A,[suspend|Cs],L?,R) :- suspended_run(A?,Cs?,L,R?). % suspend

procedure suspended_run(_?, Stream?, _?, _).
suspended_run(A,[resume|Cs],L?,R) :-  run(A?,Cs?,L,R?).  % resume
suspended_run(A,[C|Cs],L?,R) :- otherwise | run(A?,[C?|Cs?],L,R?).
```

Its first argument is the process (goal) to be executed, its second argument `Cs` is the observed interrupt stream, and its last two arguments form a 'difference-list', a standard logic programming technique (Sterling and Shapiro 1994) by which a list can be accumulated in a distributed way (the program is not fail-stop resilient; it can be extended to be so).

Appendix G Social Graph Walkthrough

This appendix presents the detailed walkthrough of the grassroots social graph program introduced in Section 2.6.

G.1 Types and Channels

The social graph uses the following type definitions:

```
Response ::= accept(Channel) ; no.
OutputEntry ::= output(String, Stream?).
OutputsList ::= [] ; [OutputEntry|OutputsList].
NetMsg ::= msg(Constant, _).
NetStream ::= [] ; [NetMsg|NetStream].
AgentId ::= Constant.
Decision ::= yes ; no.
```

A `Channel` is a pair of streams for bidirectional communication. The `new_channel` guard creates complementary channel endpoints:

```
new_channel(ch(Xs?, Ys), ch(Ys?, Xs)).
```

Both endpoints read from their first stream and write to their second: The first reads from `Xs?` and writes to `Ys`; the second reads from `Ys?` and writes to `Xs`.

G.2 Agent Structure

Each agent processes messages from two separate input streams—user and network—and maintains an outputs list mapping named destinations to output streams:

```
procedure agent(AgentId?, Stream?, Stream?, OutputsList?).
```

The four arguments are: agent identity, user input stream, network input stream, and an outputs list. The outputs list initially contains two entries—`output('_user', ...)` and `output('_net', ...)`—providing streams to the user interface and the network, respectively. As the agent befriends others, new entries are added.

```
%% User initiates cold call
agent(Id, [msg('_user', Id1, connect(Target))|UserIn], NetIn, Outs) :-
    Id? =?= Id1?, ground(Target?) |
    lookup_send('_net', msg(Target?, intro(Id?, Resp)), Outs?, Outs1),
    inject_msg(Resp?, Target?, Id?, UserIn?, UserIn1),
    agent(Id?, UserIn1?, NetIn?, Outs1?).

%% Received cold-call introduction from net (2-arg msg)
agent(Id, UserIn, [msg(Id1, intro(From, Resp))|NetIn], Outs) :-
    Id? =?= Id1? |
    lookup_send('_user', msg(agent, '_user', befriend(From?, Resp?)), Outs?, Outs1),
    agent(Id?, UserIn?, NetIn?, Outs1?).

%% User decision on cold-call
agent(Id, [msg('_user', Id1, decision(Dec, From, Resp?))|UserIn], NetIn, Outs) :-
    Id? =?= Id1? |
    bind_response(Dec?, From?, Resp, Outs?, Outs1, NetIn?, NetIn1),
    agent(Id?, UserIn?, NetIn1?, Outs1?).

%% Response to sent cold-call (injected into UserIn by inject_msg)
agent(Id, [msg(From, Id1, response(Resp))|UserIn], NetIn, Outs) :-
    Id? =?= Id1? |
    handle_response(Resp?, From?, Outs?, Outs1, NetIn?, NetIn1),
    agent(Id?, UserIn?, NetIn1?, Outs1?).
```

Fig. G 1. Cold-call befriending protocol

G.3 Cold-Call Befriending Protocol

The cold-call protocol enables agents to establish friendship without prior shared variables. Figure G 1 shows the protocol clauses.

The protocol works as follows: (1) Alice sends `connect(bob)` to her agent via the `'_user'` stream; (2) her agent sends a 2-argument `msg(bob, intro(alice, Resp))` via the `'_net'` output, including a fresh response variable `Resp`; (3) Bob receives `msg(bob, intro(alice, Resp))` on his network input and forwards the request to his user interface; (4) Bob decides `yes` or `no`; (5) Bob's agent creates a channel pair and sends `accept(Ch)` back via `Resp`; (6) Alice's agent receives the response and both agents add each other to their outputs lists.

G.4 Channel Establishment

When a cold-call is accepted, both agents establish symmetric channels (Figure G 2).

The accepting agent creates a channel pair via `new_channel(RetCh, LocalCh)`. It sends `RetCh` back to the initiator and keeps `LocalCh` for itself. Both channels are complementary: each agent's input is the other's output. Upon acceptance, `add_output` adds the new friend's output stream to the outputs list, and the friend's input stream is merged into the agent's network input via `merge`.

G.5 Text Messaging

Once agents are friends, they can exchange text messages (Figure G 3).

```
procedure bind_response(Decision?, AgentId?, Response,
    OutputsList?, OutputsList, Stream?, Stream).
bind_response(yes, From, accept(RetCh?), Outs, Outs1?, In, In1?) :-
    new_channel(RetCh, LocalCh) |
    handle_response(accept(LocalCh?), From?, Outs?, Outs1, In?, In1).
bind_response(no, _, no, Outs, Outs1?, In, In?) :-
    lookup_send('_user', msg(agent, '_user', rejected), Outs?, Outs1).

procedure handle_response(Response?, AgentId?,
    OutputsList?, OutputsList, Stream?, Stream).
handle_response(accept(ch(FIn, FOut?)), From, Outs, Outs2?, In, In1?) :-
    ground(From?) |
    add_output(From?, FOut, Outs?, Outs1),
    lookup_send('_user', msg(agent, '_user', connected(From?)), Outs1?, Outs2),
    merge(In?, FIn?, In1).
handle_response(no, From, Outs, Outs1?, In, In?) :-
    ground(From?) |
    lookup_send('_user', msg(agent, '_user', rejected(From?)), Outs?, Outs1).
```

Fig. G 2. Channel establishment

```
%% User sends text to friend
agent(Id,
  [msg('_user', Id1,
    send(Target, Text))|UserIn],
  NetIn, Outs) :-
    Id? =?= Id1?, ground(Target?) |
    lookup_send(Target?,
      msg(Id?, Target?, text(Text?)),
      Outs?, Outs1),
    agent(Id?, UserIn?, NetIn?,
      Outs1?).
```

```
%% Received text from friend
agent(Id, UserIn,
  [msg(From, Id1, text(Text))|NetIn],
  Outs) :-
    Id? =?= Id1? |
    lookup_send('_user',
      msg(agent, '_user',
        received(From?, Text?)),
      Outs?, Outs1),
    agent(Id?, UserIn?, NetIn?,
      Outs1?).
```

Fig. G 3. Text messaging between friends

G.6 Friend-Mediated Introduction

Once agents are friends, they can introduce each other to third parties. The introducer creates a fresh channel pair and sends each half to the respective parties (Figure G 4).

When Bob types `introduce(alice, charlie)`, he creates a channel pair via `new_channel(PQCh, QPCh)`. Alice receives `ch(QtoP?, PtoQ)`—she reads from Charlie via `QtoP?` and writes to Charlie via `PtoQ`. Charlie receives the complementary `ch(PtoQ?, QtoP)`. When both accept, they become direct friends without Bob's further involvement.

Note the distinction between cold-call messages and friend-mediated messages: cold-call uses a 2-argument `msg(Target, Content)` sent via the `'_net'` output, while friend-mediated introduction uses a 3-argument `msg(From, To, intro(Other, Ch))` sent directly to a friend via their named output.

```
%% User commands: introduce P to Q
agent(Id,
  [msg('_user', Id1,
    introduce(P, Q))|UserIn],
  NetIn, Outs) :-
    Id? =?= Id1?, ground(P?),
    ground(Q?),
    new_channel(PQCh, QPCh) |
    lookup_send(P?,
      msg(Id?, P?, intro(Q?, QPCh?)),
      Outs?, Outs1),
    lookup_send(Q?,
      msg(Id?, Q?, intro(P?, PQCh?)),
      Outs1?, Outs2),
    agent(Id?, UserIn?, NetIn?,
      Outs2?).
```

```
%% Received introduction from friend
agent(Id, UserIn,
  [msg(From, Id1,
    intro(Other, Ch))|NetIn],
  Outs) :-
    Id? =?= Id1?, ground(Other?) |
    lookup_send('_user',
      msg(agent, '_user',
        befriend_intro(
          From?, Other?, Ch?)),
      Outs?, Outs1),
    agent(Id?, UserIn?, NetIn?,
      Outs1?).

%% User accepts friend introduction
agent(Id,
  [msg('_user', Id1,
    accept_intro(Other, Ch))|UserIn],
  NetIn, Outs) :-
    Id? =?= Id1?, ground(Other?) |
    handle_intro_accept(Ch?, Other?,
      Outs?, Outs1, NetIn?, NetIn1),
    agent(Id?, UserIn?, NetIn1?,
      Outs1?).
```

Fig. G 4. Friend-mediated introduction protocol

G.7 Network Switch Simulation

In deployment, agents communicate through a physical network. In simulation, a `network3` process routes messages between three agents (Figure G 5).

```
procedure network3(
  Channel?, Channel?, Channel?).

%% Cold-call routing (2-arg msg)

%% Alice cold-calls Bob
network3(
  ch([msg(bob, X)|AliceIn],
     AliceOut?),
  ch(BobIn,
     [msg(bob, X?)|BobOut?]),
  ch(CharlieIn, CharlieOut?)) :-
    network3(
      ch(AliceIn?, AliceOut),
      ch(BobIn?, BobOut),
      ch(CharlieIn?, CharlieOut)).

%% (Five more 2-arg clauses for
%%  other sender/receiver pairs)

%% Friend-to-friend (3-arg msg)

%% Alice -> Bob
network3(
  ch([msg(alice, bob, X)|AliceIn],
     AliceOut?),
  ch(BobIn,
     [msg(alice, bob, X?)
      |BobOut?]),
  ch(CharlieIn, CharlieOut?)) :-
    network3(
      ch(AliceIn?, AliceOut),
      ch(BobIn?, BobOut),
      ch(CharlieIn?, CharlieOut)).

%% Bob -> Charlie
network3(
  ch(AliceIn, AliceOut?),
  ch([msg(bob, charlie, X)|BobIn],
     BobOut?),
  ch(CharlieIn,
     [msg(bob, charlie, X?)
      |CharlieOut?])) :-
    network3(
      ch(AliceIn?, AliceOut),
      ch(BobIn?, BobOut),
      ch(CharlieIn?, CharlieOut)).

%% (Four more 3-arg clauses for
%%  other sender/receiver pairs)

%% Termination
network3(
  ch([], []),
  ch([], []),
  ch([], [])).
```

Fig. G 5. Network switch simulation for three agents

The network switch routes both 2-argument cold-call messages (where `msg(Target, Content)` is addressed by target name) and 3-argument friend-to-friend messages (where `msg(From, To, Content)` carries both sender and receiver identities). In deployment, the network switch is replaced by maGLP's Cold-call transaction (Section 3).

G.8 The Scenario and Actors

The complete scenario demonstrates all three protocols:

1. Alice cold-calls Bob (Bob accepts) — Alice and Bob become friends
2. Alice sends Bob: "Hi Bob, this is Alice"
3. Bob cold-calls Charlie (Charlie accepts) — Bob and Charlie become friends
4. Charlie sends Bob: "Hi Bob, this is Charlie"

5. Bob introduces Alice to Charlie (both accept) — Alice and Charlie become direct friends
6. Alice sends Charlie: "Hi Charlie, this is Alice"
7. Charlie responds: "Hi Alice, this is Charlie"

Each agent is driven by an *actor*—a GLP procedure that implements a state machine, reacting to messages from the agent and producing commands. The actor's state is encoded in procedure names (e.g., `alice_wait_bob_connected`, `bob_wait_charlie_msg`), with transitions via recursive calls. See Appendix H for the complete program: helper predicates, actor implementations, the play that ties everything together, and the multiagent boot variants.

Appendix H Complete Social Graph Program

This appendix presents the complete GLP social graph program that demonstrates cold-call befriending, text messaging, and friend-mediated introduction. The program includes three agents (Alice, Bob, Charlie) driven by actor scripts that execute a comprehensive scenario. We present the shared agent code, then three boot variants: a single-process play with a network switch (Concurrent GLP), a multiagent boot with actors (maGLP), and a multiagent UI boot with interactive windows (maGLP).

H.1 The Scenario

The play executes the following sequence:

1. Alice cold-calls Bob (Bob accepts) — Alice and Bob become friends
2. Alice sends message to Bob: "Hi Bob, this is Alice"
3. Bob cold-calls Charlie (Charlie accepts) — Bob and Charlie become friends
4. Charlie sends message to Bob: "Hi Bob, this is Charlie"
5. Bob introduces Alice to Charlie (both accept) — Alice and Charlie become direct friends
6. Alice sends message to Charlie: "Hi Charlie, this is Alice"
7. Charlie responds to Alice: "Hi Alice, this is Charlie"

H.2 Type Definitions

```
Response ::= accept(Channel) ; no.
OutputEntry ::= output(String, Stream?).
OutputsList ::= [] ; [OutputEntry|OutputsList].
NetMsg ::= msg(Constant, _).
NetStream ::= [] ; [NetMsg|NetStream].
AgentId ::= Constant.
Decision ::= yes ; no.
```

H.3 Channel Operations

```
send(X, ch(In, [X?|Out?]), ch(In?, Out)).
```

```
receive(X?, ch([X|In], Out?), ch(In?, Out)).
new_channel(ch(Xs?, Ys), ch(Ys?, Xs)).
```

H.4 Stream Utilities

```
procedure merge(Stream?, Stream?, Stream).
merge([X|Xs], Ys, [X?|Zs?]) :- merge(Ys?, Xs?, Zs).
merge(Xs, [Y|Ys], [Y?|Zs?]) :- merge(Xs?, Ys?, Zs).
merge([], Ys, Ys?).
merge(Xs, [], Xs?).
```

H.5 Outputs List Operations

```
procedure lookup_send(String?, _?, OutputsList?, OutputsList).
lookup_send(Key, Msg, Outs, Outs1?) :-
    ground(Key?) |
    lookup_send_step(Key?, Msg?, Outs?, Outs1).

procedure lookup_send_step(String?, _?, OutputsList?, OutputsList).
lookup_send_step(Key, Msg,
    [output(K, [Msg?|Out1?])|Rest],
    [output(K?, Out1)|Rest?]) :-
        Key? =?= K? | true.
lookup_send_step(Key, Msg,
    [output(K, Out?)|Rest],
    [output(K?, Out)|Rest1?]) :-
        otherwise |
        lookup_send_step(Key?, Msg?, Rest?, Rest1).
lookup_send_step(_, _, [], []).

procedure inject_msg(Response?, AgentId?, AgentId?, Stream?, Stream).
inject_msg(Resp, Target, Id, Ys,
    [msg(Target?, Id?, response(Resp?))|Ys?]) :-
        known(Resp?) | true.
inject_msg(Resp, Target, Id, [Y|Ys], [Y?|Ys1?]) :-
    otherwise |
    inject_msg(Resp?, Target?, Id?, Ys?, Ys1).

procedure add_output(String?, Stream, OutputsList?, OutputsList).
add_output(Name, Out?, Outs, [output(Name?, Out)|Outs?]).

procedure close_outputs(OutputsList?).
close_outputs([output(_, [])|Outs]) :- close_outputs(Outs?).
close_outputs([]).
```

H.6 Response Handling

```
procedure bind_response(Decision?, AgentId?, Response,
    OutputsList?, OutputsList, Stream?, Stream).
bind_response(yes, From, accept(RetCh?),
    Outs, Outs1?, In, In1?) :-
        new_channel(RetCh, LocalCh) |
        handle_response(accept(LocalCh?), From?,
            Outs?, Outs1, In?, In1).
bind_response(no, _, no, Outs, Outs1?, In, In?) :-
    lookup_send('_user',
        msg(agent, '_user', rejected),
        Outs?, Outs1).

procedure handle_response(Response?, AgentId?,
    OutputsList?, OutputsList, Stream?, Stream).
handle_response(accept(ch(FIn, FOut?)), From,
    Outs, Outs2?, In, In1?) :-
        ground(From?) |
        add_output(From?, FOut, Outs?, Outs1),
        lookup_send('_user',
            msg(agent, '_user', connected(From?)),
            Outs1?, Outs2),
        merge(In?, FIn?, In1).
handle_response(no, From, Outs, Outs1?, In, In?) :-
    ground(From?) |
    lookup_send('_user',
        msg(agent, '_user', rejected(From?)),
        Outs?, Outs1).
```

H.7 Introduction Acceptance

```
procedure handle_intro_accept(Channel?, AgentId?,
    OutputsList?, OutputsList, Stream?, Stream).
handle_intro_accept(ch(FIn, FOut?), Other,
    Outs, Outs2?, In, In1?) :-
        ground(Other?) |
        add_output(Other?, FOut, Outs?, Outs1),
        lookup_send('_user',
            msg(agent, '_user', connected(Other?)),
            Outs1?, Outs2),
        merge(In?, FIn?, In1).
```

H.8 The Agent

The agent procedure is the main event loop handling all protocols. It takes four arguments: agent identity, user input stream, network input stream, and an outputs list mapping names to output streams.

```
procedure agent(AgentId?, Stream?, Stream?, OutputsList?).

%% --- User messages (from '_user') ---

%% User initiates cold call
agent(Id, [msg('_user', Id1, connect(Target))|UserIn],
    NetIn, Outs) :-
        Id? =?= Id1?, ground(Target?) |
        lookup_send('_net',
            msg(Target?, intro(Id?, Resp)),
            Outs?, Outs1),
        inject_msg(Resp?, Target?, Id?,
            UserIn?, UserIn1),
        agent(Id?, UserIn1?, NetIn?, Outs1?).

%% User decision on cold-call introduction
agent(Id, [msg('_user', Id1,
    decision(Dec, From, Resp?))|UserIn],
    NetIn, Outs) :-
        Id? =?= Id1? |
        bind_response(Dec?, From?, Resp,
            Outs?, Outs1, NetIn?, NetIn1),
        agent(Id?, UserIn?, NetIn1?, Outs1?).

%% User sends text message to friend
agent(Id, [msg('_user', Id1,
    send(Target, Text))|UserIn],
    NetIn, Outs) :-
        Id? =?= Id1?, ground(Target?) |
        lookup_send(Target?,
            msg(Id?, Target?, text(Text?)),
            Outs?, Outs1),
        agent(Id?, UserIn?, NetIn?, Outs1?).

%% User commands: introduce P to Q
agent(Id, [msg('_user', Id1,
    introduce(P, Q))|UserIn],
    NetIn, Outs) :-
        Id? =?= Id1?, ground(P?), ground(Q?),
        new_channel(PQCh, QPCh) |
        lookup_send(P?,
```

```
            msg(Id?, P?, intro(Q?, QPCh?)),
            Outs?, Outs1),
        lookup_send(Q?,
            msg(Id?, Q?, intro(P?, PQCh?)),
            Outs1?, Outs2),
        agent(Id?, UserIn?, NetIn?, Outs2?).

%% User accepts friend introduction
agent(Id, [msg('_user', Id1,
    accept_intro(Other, Ch))|UserIn],
    NetIn, Outs) :-
        Id? =?= Id1?, ground(Other?) |
        handle_intro_accept(Ch?, Other?,
            Outs?, Outs1, NetIn?, NetIn1),
        agent(Id?, UserIn?, NetIn1?, Outs1?).

%% User rejects friend introduction
agent(Id, [msg('_user', Id1,
    reject_intro(_))|UserIn],
    NetIn, Outs) :-
        Id? =?= Id1? |
        agent(Id?, UserIn?, NetIn?, Outs?).

%% Response to cold-call (injected into UserIn)
agent(Id, [msg(From, Id1, response(Resp))|UserIn],
    NetIn, Outs) :-
        Id? =?= Id1? |
        handle_response(Resp?, From?,
            Outs?, Outs1, NetIn?, NetIn1),
        agent(Id?, UserIn?, NetIn1?, Outs1?).

%% User catch-all
agent(Id, [_|UserIn], NetIn, Outs) :-
    ground(Id?), otherwise |
    agent(Id?, UserIn?, NetIn?, Outs?).

%% --- Network messages ---

%% Received cold-call introduction from net
agent(Id, UserIn,
    [msg(Id1, intro(From, Resp))|NetIn], Outs) :-
        Id? =?= Id1? |
        lookup_send('_user',
            msg(agent, '_user',
                befriend(From?, Resp?)),
            Outs?, Outs1),
```

```
        agent(Id?, UserIn?, NetIn?, Outs1?).

%% Response to sent cold-call introduction
agent(Id, UserIn,
    [msg(From, Id1, response(Resp))|NetIn],
    Outs) :-
        Id? =?= Id1? |
        handle_response(Resp?, From?,
            Outs?, Outs1, NetIn?, NetIn1),
        agent(Id?, UserIn?, NetIn1?, Outs1?).

%% Received text message from friend
agent(Id, UserIn,
    [msg(From, Id1, text(Text))|NetIn], Outs) :-
        Id? =?= Id1? |
        lookup_send('_user',
            msg(agent, '_user',
                received(From?, Text?)),
            Outs?, Outs1),
        agent(Id?, UserIn?, NetIn?, Outs1?).

%% Received introduction from friend
agent(Id, UserIn,
    [msg(From, Id1, intro(Other, Ch))|NetIn],
    Outs) :-
        Id? =?= Id1?, ground(Other?) |
        lookup_send('_user',
            msg(agent, '_user',
                befriend_intro(From?, Other?, Ch?)),
            Outs?, Outs1),
        agent(Id?, UserIn?, NetIn?, Outs1?).

%% Network catch-all
agent(Id, UserIn, [_|NetIn], Outs) :-
    ground(Id?), otherwise |
    agent(Id?, UserIn?, NetIn?, Outs?).

%% --- Termination ---

agent(_, [], _, Outs) :- close_outputs(Outs?).
agent(_, _, [], Outs) :- close_outputs(Outs?).
```

H.9 Network Switch

The three-way network switch routes messages between Alice, Bob, and Charlie. It handles both 2-argument cold-call messages and 3-argument friend-to-friend messages.

```
procedure network3(Channel?, Channel?, Channel?).

%% --- Cold-call routing (2-arg msg format) ---

%% Alice cold-calls Bob
network3(ch([msg(bob, X)|AliceIn], AliceOut?),
         ch(BobIn, [msg(bob, X?)|BobOut?]),
         ch(CharlieIn, CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Alice cold-calls Charlie
network3(
    ch([msg(charlie, X)|AliceIn], AliceOut?),
    ch(BobIn, BobOut?),
    ch(CharlieIn,
        [msg(charlie, X?)|CharlieOut?])) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Bob cold-calls Alice
network3(
    ch(AliceIn,
        [msg(alice, X?)|AliceOut?]),
    ch([msg(alice, X)|BobIn], BobOut?),
    ch(CharlieIn, CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Bob cold-calls Charlie
network3(ch(AliceIn, AliceOut?),
    ch([msg(charlie, X)|BobIn], BobOut?),
    ch(CharlieIn,
        [msg(charlie, X?)|CharlieOut?])) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Charlie cold-calls Alice
network3(
    ch(AliceIn,
        [msg(alice, X?)|AliceOut?]),
    ch(BobIn, BobOut?),
```

```
    ch([msg(alice, X)|CharlieIn],
        CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Charlie cold-calls Bob
network3(ch(AliceIn, AliceOut?),
    ch(BobIn, [msg(bob, X?)|BobOut?]),
    ch([msg(bob, X)|CharlieIn],
        CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% --- Friend-to-friend routing (3-arg msg) ---

%% Alice -> Bob
network3(
    ch([msg(alice, bob, X)|AliceIn],
        AliceOut?),
    ch(BobIn,
        [msg(alice, bob, X?)|BobOut?]),
    ch(CharlieIn, CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Alice -> Charlie
network3(
    ch([msg(alice, charlie, X)|AliceIn],
        AliceOut?),
    ch(BobIn, BobOut?),
    ch(CharlieIn,
        [msg(alice, charlie, X?)
         |CharlieOut?])) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Bob -> Alice
network3(
    ch(AliceIn,
        [msg(bob, alice, X?)|AliceOut?]),
    ch([msg(bob, alice, X)|BobIn],
        BobOut?),
```

```
    ch(CharlieIn, CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Bob -> Charlie
network3(ch(AliceIn, AliceOut?),
    ch([msg(bob, charlie, X)|BobIn],
        BobOut?),
    ch(CharlieIn,
        [msg(bob, charlie, X?)
         |CharlieOut?])) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Charlie -> Alice
network3(
    ch(AliceIn,
        [msg(charlie, alice, X?)
         |AliceOut?]),
    ch(BobIn, BobOut?),
    ch([msg(charlie, alice, X)|CharlieIn],
        CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Charlie -> Bob
network3(ch(AliceIn, AliceOut?),
    ch(BobIn,
        [msg(charlie, bob, X?)|BobOut?]),
    ch([msg(charlie, bob, X)|CharlieIn],
        CharlieOut?)) :-
    network3(ch(AliceIn?, AliceOut),
        ch(BobIn?, BobOut),
        ch(CharlieIn?, CharlieOut)).

%% Termination
network3(ch([], []), ch([], []), ch([], [])).
```

H.10 Actors as State Machines

Each actor is a GLP procedure that implements a state machine, reacting to messages from the agent and producing commands for it. The actor's state is encoded in the procedure name (e.g., `alice_wait_bob_connected`, `alice_wait_intro`), and transitions occur via recursive calls to different procedures.

H.10.1 Alice's Actor

Alice's script: (1) cold-call Bob, (2) wait for connection confirmation, (3) send greeting to Bob, (4) wait for introduction to Charlie and accept it, (5) send greeting to Charlie, (6) wait for Charlie's reply.

```
procedure alice_actor(Channel?).
alice_actor(ch(In,
    [msg('_user', alice, connect(bob))|Out?])) :-
        alice_wait_bob_connected(In?, Out).

procedure alice_wait_bob_connected(Stream?, Stream).
alice_wait_bob_connected(
    [msg(agent, '_user', connected(bob))|In],
    [msg('_user', alice,
        send(bob, 'Hi Bob, this is Alice'))
     |Out?]) :-
        alice_wait_intro(In?, Out).
alice_wait_bob_connected([_|In], Out?) :-
    otherwise |
    alice_wait_bob_connected(In?, Out).
alice_wait_bob_connected([], []).

procedure alice_wait_intro(Stream?, Stream).
alice_wait_intro(
    [msg(agent, '_user',
        befriend_intro(_, Other, Ch))|In],
    [msg('_user', alice,
        accept_intro(Other?, Ch?))|Out?]) :-
        ground(Other?) |
        alice_send_to_charlie(In?, Out).
alice_wait_intro([_|In], Out?) :-
    otherwise | alice_wait_intro(In?, Out).
alice_wait_intro([], []).

procedure alice_send_to_charlie(Stream?, Stream).
alice_send_to_charlie(In,
    [msg('_user', alice,
        send(charlie,
            'Hi Charlie, this is Alice'))
     |Out?]) :-
        alice_wait_charlie_reply(In?, Out).

procedure alice_wait_charlie_reply(Stream?, Stream).
```

```
alice_wait_charlie_reply(
    [msg(agent, '_user',
        received(charlie, _))|_], []).
alice_wait_charlie_reply([_|In], Out?) :-
    otherwise |
    alice_wait_charlie_reply(In?, Out).
alice_wait_charlie_reply([], []).
```

H.10.2 Bob's Actor

Bob's script: (1) accept Alice's cold-call, (2) receive Alice's message, (3) cold-call Charlie, (4) wait for connection, (5) receive Charlie's message, (6) introduce Alice to Charlie.

```
procedure bob_actor(Channel?).
bob_actor(ch(
    [msg(agent, '_user',
        befriend(From, Resp))|In],
    [msg('_user', bob,
        decision(yes, From?, Resp?))
     |Out?])) :-
        ground(From?) |
        bob_wait_alice_msg(In?, Out).
bob_actor(ch([_|In], Out?)) :-
    otherwise | bob_actor(ch(In?, Out)).

procedure bob_wait_alice_msg(Stream?, Stream).
bob_wait_alice_msg(
    [msg(agent, '_user', connected(_))|In],
    Out?) :-
        bob_wait_alice_msg(In?, Out).
bob_wait_alice_msg(
    [msg(agent, '_user',
        received(alice, _))|In],
    [msg('_user', bob, connect(charlie))
     |Out?]) :-
        bob_wait_charlie_connected(In?, Out).
bob_wait_alice_msg([_|In], Out?) :-
    otherwise |
    bob_wait_alice_msg(In?, Out).
bob_wait_alice_msg([], []).

procedure bob_wait_charlie_connected(Stream?, Stream).
bob_wait_charlie_connected(
    [msg(agent, '_user',
        connected(charlie))|In], Out?) :-
        bob_wait_charlie_msg(In?, Out).
bob_wait_charlie_connected([_|In], Out?) :-
```

```
    otherwise |
    bob_wait_charlie_connected(In?, Out).
bob_wait_charlie_connected([], []).

procedure bob_wait_charlie_msg(Stream?, Stream).
bob_wait_charlie_msg(
    [msg(agent, '_user',
        received(charlie, _))|_],
    [msg('_user', bob,
        introduce(alice, charlie))]).
bob_wait_charlie_msg([_|In], Out?) :-
    otherwise |
    bob_wait_charlie_msg(In?, Out).
bob_wait_charlie_msg([], []).
```

H.10.3 Charlie's Actor

Charlie's script: (1) accept Bob's cold-call, (2) send greeting to Bob, (3) wait for introduction to Alice and accept it, (4) receive Alice's message, (5) send reply to Alice.

```
procedure charlie_actor(Channel?).
charlie_actor(ch(
    [msg(agent, '_user',
        befriend(From, Resp))|In],
    [msg('_user', charlie,
        decision(yes, From?, Resp?)),
     msg('_user', charlie,
        send(bob,
            'Hi Bob, this is Charlie'))
     |Out?])) :-
        ground(From?) |
        charlie_wait_intro(In?, Out).
charlie_actor(ch([_|In], Out?)) :-
    otherwise |
    charlie_actor(ch(In?, Out)).

procedure charlie_wait_intro(Stream?, Stream).
charlie_wait_intro(
    [msg(agent, '_user',
        befriend_intro(_, Other, Ch))|In],
    [msg('_user', charlie,
        accept_intro(Other?, Ch?))|Out?]) :-
        ground(Other?) |
        charlie_wait_alice_msg(In?, Out).
charlie_wait_intro([_|In], Out?) :-
    otherwise |
    charlie_wait_intro(In?, Out).
```

```
charlie_wait_intro([], []).

procedure charlie_wait_alice_msg(Stream?, Stream).
charlie_wait_alice_msg(
    [msg(agent, '_user',
        received(alice, _))|_],
    [msg('_user', charlie,
        send(alice,
            'Hi Alice, this is Charlie'))]).
charlie_wait_alice_msg([_|In], Out?) :-
    otherwise |
    charlie_wait_alice_msg(In?, Out).
charlie_wait_alice_msg([], []).
```

H.11 The Play (Concurrent GLP)

The play initializes three agents with their user and network channels, connects them via the network switch, and starts each actor. All agents run in a single process.

```
procedure play.
play :-
    network3(ch(AliceNetOut?, AliceNetIn),
             ch(BobNetOut?, BobNetIn),
             ch(CharlieNetOut?, CharlieNetIn)),
    agent(alice, AliceUserOut?, AliceNetIn?,
        [output('_user', AliceUserIn),
         output('_net', AliceNetOut)]),
    agent(bob, BobUserOut?, BobNetIn?,
        [output('_user', BobUserIn),
         output('_net', BobNetOut)]),
    agent(charlie, CharlieUserOut?,
        CharlieNetIn?,
        [output('_user', CharlieUserIn),
         output('_net', CharlieNetOut)]),
    alice_actor(ch(AliceUserIn?, AliceUserOut)),
    bob_actor(ch(BobUserIn?, BobUserOut)),
    charlie_actor(
        ch(CharlieUserIn?, CharlieUserOut)).
```

The play terminates successfully when all streams are closed and all agents reach their terminal states. The network switch simulates the Cold-call transaction introduced in Section 3.

H.12 Multiagent Boot (maGLP with Actors)

In the multiagent setting, each agent runs on a separate isolate. The `boot` procedure spawns agents using the `@agent` syntax, which places each goal on the named agent's isolate. The `send_to_net` system predicate returns the agent's network output stream. No network switch is needed—maGLP's Cold-call transaction handles inter-agent communication directly.

```
procedure boot.
boot :-
    agent_init(alice, _)@alice,
    agent_init(bob, _)@bob,
    agent_init(charlie, _)@charlie.

procedure agent_init(Constant?, Stream?).
agent_init(Id, NetIn) :-
    ground(Id?) |
    send_to_net(NetOut?),
    agent(Id?, UserOut?, NetIn?,
        [output('_user', UserIn),
         output('_net', NetOut)]),
    actor(Id?, ch(UserIn?, UserOut)).

procedure actor(_?, Channel?).
actor(alice, Ch) :- alice_actor(Ch?).
actor(bob, Ch) :- bob_actor(Ch?).
actor(charlie, Ch) :- charlie_actor(Ch?).
```

H.13 Multiagent UI Boot (maGLP with Windows)

In the interactive UI variant, each agent runs in its own window. The UI provides user and network channels directly—no actors are needed, as the user interacts by typing GLP terms (e.g., `connect(bob)`, `send(bob, hello)`, `introduce(alice, charlie)`).

```
procedure agent_init(Constant?, Channel?, Channel?).
agent_init(Id, ch(UserIn, UserOut?),
    ch(NetIn, NetOut?)) :-
        ground(Id?) |
        agent(Id?, UserOut?, NetIn?,
            [output('_user', UserIn),
             output('_net', NetOut)]).
```