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

Ehud Shapiro
London School of Economics and Weizmann Institute of Science
(e-mail: ehud.shapiro@weizmann.ac.il)

Abstract

A digital platform is *grassroots* if it can have multiple instances that operate independently of each other and of any global resource other than the network, yet may coalesce into ever larger instances, possibly into a single global instance.

Grassroots Logic Programs (GLP) is a multiagent concurrent logic programming language designed to implement grassroots platforms. Here, we review the mathematical foundation of grassroots platforms, define GLP, prove that it is grassroots, and demonstrate its utility in implementing grassroots platforms via examples of GLP programs for grassroots social networks and grassroots cryptocurrencies.

1 Introduction

The democratic deficit in the digital realm. While the physical lives of many of us are in democracies (one person, one vote), our digital lives are governed by autocracies (one person, all votes) or plutocracies (one coin, one vote). Cloud platforms like Facebook and Uber operate under surveillance capitalism [64]: corporate executives embody all three branches of government, members have no civil rights, and personal information is commercially exploited with little remuneration. Blockchain-based systems like Bitcoin and Ethereum offer decentralization but not democracy—their governance is intrinsically plutocratic, granting power in proportion to capital investment and thus exacerbating economic inequality [6]. Looking for digital systems that offer egalitarian control and democratic governance intrinsically—not at the courtesy of an underlying autocratic or plutocratic platform—we find none.

Grassroots platforms. Grassroots platforms [50, 53] are distributed applications designed to restore digital sovereignty to ordinary people using only their smartphones. A distributed system is *grassroots* if it is permissionless and can have autonomous, independently-deployed instances—geographically and over time—that may interoperate once interconnected. This requires that agents can operate within their group without interference from agents outside it, while retaining the ability to interact when desired.

Grassroots platforms are run by people on their networked personal devices, who are identified cryptographically [41], communicate only with authenticated friends, and can participate in multiple instances of multiple platforms simultaneously. The grassroots social graph [51] serves as both a platform in its own right and the infrastructure layer for all other grassroots platforms: nodes represent people, edges represent authenticated friendships, and connected components arise spontaneously through befriending. Upon this foundation, grassroots social networks [51], grassroots cryptocurrencies [52], and grassroots democratic federations [54] are built.

Why existing alternatives fail. Federated systems like Mastodon [39] remain server-dependent: users are still at the mercy of server operators who control their accounts autocratically. Global shared data structures—whether replicated (blockchains), distributed (IPFS, DHT), or pub/sub with global directories—prevent true independence, as members cannot ignore changes made by others [50]. Single-instance architectures create lock-in: two instances would clash over global resources (domain names, boot nodes) hardwired into their code. No community can independently bootstrap and later interoperate—they must join existing platforms on those platforms' terms.

Grassroots platforms require a different architecture: one where small groups can form and operate independently, yet coalesce into larger communities when they interconnect, all without central coordination or global consensus.

Programming grassroots platforms. A key challenge in implementing grassroots platforms is overcoming faulty and malicious participants [24]. Without secure language support, correct participants cannot reliably identify each other, establish secure communication channels, or verify each other's code integrity [11, 42]. While grassroots platforms have been formally specified and their properties proven [50–54], they remain mathematical constructions without actual implementations. To the best of our knowledge, no existing programming language provides the necessary combination of distributed execution, cryptographic security, safety, and liveness guarantees required to realize these specifications. GLP aims to close the gap between mathematical specification and implementation of grassroots platforms.

GLP. We present GLP, a secure, multiagent, concurrent, logic programming language designed for implementing grassroots platforms. GLP extends logic programs [31, 57] with paired single-reader/single-writer variables (akin to futures and promises [3, 12]), each establishing a secure single-message communication channel between the single writer and the single reader, enabling subsequent secure multidirectional communication by sharing readers and writers in messages.

We present GLP and prove its properties in four steps:

(1) **Mathematical Foundations:** We recall the mathematical framework for grassroots platforms [53]: transition systems, multiagent transition systems with atomic transactions, protocols, and the key theorem that a protocol over interactive transactions is grassroots.
(2) **Logic Programs:** We recall logic programs (LP) [31, 57] and their operational semantics via transition systems.
(3) **GLP:** We extend LP with reader/writer pairs satisfying the Single-Reader/Single-Writer (SRSW) requirement; extend unification to suspend upon an attempt to bind a reader; extend

configurations to include pending assignments to readers; and provide nondeterministic interleaving-based asynchronous operational semantics. We prove safety properties, including that GLP computations are deductions [23, 31].

(4) **Multiagent GLP:** We define multiagent GLP (maGLP) as a transactions-based multiagent transition system with three transactions: Reduce (unary, local goal reduction), Communicate (binary, writer-to-reader message transfer between agents), and Network (binary, cold-call channel establishment). We prove maGLP is grassroots and establish that any GLP application using cold calls is grassroots.

Paper outline. Section **??** recalls the mathematical foundations of grassroots platforms. Section **??** recalls logic programs. Section 2 extends them to GLP. Section 3 defines multiagent GLP and proves it grassroots. Section **??** presents basic GLP programming techniques. Section **??** implements the grassroots social graph protocols. Section **??** discusses implementation. Section 5 reviews related work, and Section 6 concludes.

2 Concurrent GLP

This section presents Grassroots Logic Programs (GLP), a concurrent logic programming language. We begin with transition systems, recall logic programs (LP) as transition systems, and then extend LP to GLP. We illustrate GLP with programming examples and conclude with the grassroots social graph—the foundational platform that all other grassroots platforms build upon.

2.1 Transition Systems

Definition 2.1 (Transition System, Computation, Run)**.** A **transition system** is a tuple $TS = (S, s_0, T)$, where S is a non-empty set of **states**, $s_0 \in S$ is the **initial state**, and $T \subset S^2$ is a set of **transitions**, where each transition $t \in T$ is a pair (s, s') of distinct states, written $s \to s'$. A **computation** is a sequence of states $s_1, s_2, \ldots$ such that $s_i \to s_{i+1} \in T$ for each consecutive pair. A **run** is a computation starting from s_0.

2.2 Logic Programs as Transition Systems

We assume familiarity with standard Logic Programs (LP): terms, goals, clauses, substitutions, and most-general unifiers (mgu). We cast LP operationally as a transition system.

Definition 2.2 (LP Transition System)**.** Given a logic program P and initial goal G_0, the **LP transition system** $LP(P) = (C, c_0, T)$ has configurations $C = \mathcal{G}(P) \times \Sigma$ (goal–substitution pairs), initial configuration $c_0 = (G_0, \emptyset)$, and transitions $(G, \sigma) \to (G', \sigma')$ such that for some unit goal $A \in G$ and clause H :- $B \in P$ (renamed apart from G), A and H have mgu $\hat{\sigma}$, $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$, and $\sigma' = \sigma \circ \hat{\sigma}$.

LP has two forms of nondeterminism: choice of goal $A \in G$ (*and-nondeterminism*) and choice of clause $C \in P$ (*or-nondeterminism*).

2.3 GLP: Extending LP with Reader/Writer Variables

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), (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, and is futures/promises-like: each assignment $X := T$ is produced at most once via the sole writer (promise) X, and consumed at most once via its sole paired reader (future) $X?$.

Definition 2.3 (GLP Variables)**.** Let $\mathcal{V}$ be the set of LP variables, 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**.

Definition 2.4 (Single-Occurrence (SO) and Single-Reader/Single-Writer (SRSW))**.** A term, goal, or clause satisfies **SO** if every variable occurs in it at most once. A clause C satisfies **SRSW** 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.

The SO invariant is preserved: reducing a goal satisfying SO with a clause satisfying SRSW yields a goal satisfying SO (Proposition 2.11). The SRSW restriction prevents multiple writers racing to bind a variable.

Example 2.5 (Fair Merge)**.** The quintessential concurrent logic program for fairly merging two streams:

```
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?).
```

Each clause satisfies SRSW. The first two clauses swap inputs in recursive calls, ensuring fairness when both streams are available.

2.4 GLP Operational Semantics

Definition 2.6 (Writer and Reader Assignments)**.** A **writer assignment** $X := T$ has $X \in \mathcal{V}$ and $T \notin \mathcal{V}$ satisfying SO. A **reader assignment** $X? := T$ has $X? \in \mathcal{V}?$ and $T \notin \mathcal{V}$ satisfying SO. Given a writer assignment $X := T$, its **readers counterpart** is $X? := T$.

Definition 2.7 (GLP Transition System)**.** Given a GLP program P and initial goal G_0 satisfying SO, the **GLP transition system** $GLP(P) = (C, c_0, \mathcal{T})$ has:

- Configurations C: pairs (G, σ) where G is a goal and σ is a readers substitution
- Initial configuration $c_0 = (G_0, \emptyset)$
- Transitions $(G, \sigma) \to (G', \sigma')$ of two kinds:
  (1) **Reduce:** For unit goal $A \in G$, clause $C \in P$ is the *first* clause for which the writer mgu of A and (renamed) head H succeeds with $(B, \hat{\sigma})$; then $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$ and $\sigma' = \sigma \circ \hat{\sigma}$?
  (2) **Communicate:** $\{X := T\} \in \sigma$, $X? \in G$, $G' = G\{X? := T\}$, $\sigma' = \sigma$

GLP differs from LP in two key ways: (1) using writer mgu instead of regular mgu—if matching requires binding a reader, the goal *suspends* rather than fails; (2) choosing the *first* applicable clause, enabling fair concurrent programs like merge.

The Communicate transition realizes asynchronous message passing: when a writer X is bound to T, the reader assignment $X? :=$

T is added to σ; later, when $X?$ occurs in a goal, the assignment is applied.

2.5 Term Matching

The SO invariant allows eschewing unification in favour of *term matching*:

Definition 2.8 (Term Matching)**.** Given terms T_1 and T_2 jointly satisfying SO, their **term matching** traverses both term-trees jointly:

$T_1 \backslash T_2$	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 occurred and the suspension set is empty.

2.6 Guards

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

Definition 2.9 (Guarded Clause)**.** A **guarded clause** has the form H `:-` G `|` B, where H is the head, G is a guard conjunction, and B is the body.

Guards have three-valued semantics: *succeed*, *suspend* (may succeed after further instantiation), or *fail* (can never succeed). A clause is applicable only when all guards succeed; if any guard fails, the next clause is tried; if a guard suspends, clause selection suspends.

Guard predicates include type tests (`integer`, `ground`, `known`, etc.), arithmetic comparisons (`<`, `=:=`, etc.), and ground equality (`=?=`).

2.7 Safety Properties

Proposition 2.10 (GLP Computation is Deduction)**.** *Let $/?$ replace every reader by its paired writer. If $(G_0$ `:-` $G_n)\sigma$ is the outcome of a proper GLP run, then $(G_0$ `:-` $G_n)\sigma/?$ is a logical consequence of $P/?$.*

Proposition 2.11 (SO Preservation)**.** *If the initial goal G_0 satisfies SO, then every goal in the GLP run satisfies SO.*

Proposition 2.12 (Monotonicity)**.** *If unit goal A can reduce with clause C at step i, then either A has been reduced by step $j > i$, or an instance of A can still reduce with C at step j.*

Proofs appear in Appendix A.

2.8 Programming Examples

Example 2.13 (Stream Distribution)**.** Broadcasting to multiple consumers uses the `ground` guard to enable safe replication:

```
distribute([X|Xs], Ys1, Ys2) :- ground(X?) |
   Ys1 = [X?|Ys1a], Ys2 = [X?|Ys2a], distribute(Xs?, Ys1a?, Ys2a?).
distribute([], [], []).
```

When `X?` is ground, multiple occurrences in the body do not violate SRSW.

Example 2.14 (Lookup in Association List)**.**

```
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.

Example 2.15 (Deferred Message Injection)**.** The `inject` predicate inserts a message into a stream when a trigger variable becomes bound:

```
inject(Trigger, Msg, Ys, [Msg?|Ys?]) :- known(Trigger?) | true.
inject(Trigger, Msg, [Y|Ys], [Y?|Ys1?]) :-
    otherwise | inject(Trigger?, Msg?, Ys?, Ys1).
```

Until `Trigger` is bound, messages pass through unchanged; when bound, `Msg` is inserted.

2.9 The Grassroots Social Graph

The grassroots social graph is the foundational platform upon which all other grassroots platforms are built. Nodes represent cryptographically-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. This demonstrates the program structure before introducing multiagent GLP in Section 3.

2.9.1 Agent Structure. Each agent processes messages from a unified input stream, maintaining a friends list that maps names to output streams:

```
agent(Id, ch(UserIn, UserOut), ch(NetIn, NetOut)) :-
    merge(UserIn?, NetIn?, In),
   social_graph(Id?, In?, [(user, UserOut), (net, NetOut)]).
```

The agent merges user and network input streams, then enters the main event loop `social_graph` with the agent's identity, merged input, and initial friends list containing channels to the user interface and network.

2.9.2 Cold-Call Befriending Protocol. The cold-call protocol enables agents to establish friendship without prior shared variables:

```
%% User requests connection
social_graph(Id, [msg(user, Id?, connect(Target))|In], Fs) :-
    ground(Id?), ground(Target?) |
   lookup_send(net, msg(Id?, Target?, intro(Id?, Id?, Resp)), Fs?, Fs
   inject(Resp?, msg(Target?, Id?, response(Resp?)), In?, In1),
    social_graph(Id?, In1?, Fs1?).

%% Receive introduction offer
social_graph(Id, [msg(From, Id?, intro(From?, From?, Resp))|In], Fs)
    ground(Id?), ground(From?) |
   lookup_send(user, msg(agent, user, befriend(From?, Resp?)), Fs?,
    social_graph(Id?, In?, Fs1?).

%% User accepts/rejects
social_graph(Id, [msg(user, Id?, decision(Dec, From, Resp?))|In], Fs
    ground(Id?) |
   bind_response(Dec?, From?, Resp, Fs?, Fs1, In?, In1),
    social_graph(Id?, In1?, Fs1?).

%% Process response to our introduction
social_graph(Id, [msg(From, Id?, response(Resp))|In], Fs) :-
    ground(Id?) |
   handle_response(Resp?, From?, Fs?, Fs1, In?, In1),
```

```
    social_graph(Id?, In1?, Fs1?).
```

2.9.3 Channel Establishment. When an offer is accepted, both agents establish symmetric channels:

```
bind_response(yes, From, accept(ch(FOut?, FIn)), Fs, Fs1?, In, In1?) :-
   handle_response(accept(ch(FIn?, FOut)), From?, Fs?, Fs1, In?, In1).
bind_response(no, _, no, Fs, Fs?, In, In?).

handle_response(accept(ch(FIn, FOut)), From, Fs, [(From?, FOut?)|Fs?], In, In?) :-
    ground(From?) |
    tag_stream(From?, FIn?, Tagged),
    merge(In?, Tagged?, In1).
handle_response(no, _, Fs, Fs?, In, In?).
```

The channel pair `ch(FOut?, FIn)` and `ch(FIn?, FOut)` are complementary: each agent's input is the other's output.

2.9.4 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:

```
%% User requests introduction: introduce(P, Q)
social_graph(Id, [introduce(P, Q)|In], Fs) :-
    ground(Id?), ground(P?), ground(Q?) |
   lookup_send(P?, msg(Id?, P?, intro(Q?, ch(QtoP?, PtoQ))), Fs?, Fs1),
   lookup_send(Q?, msg(Id?, Q?, intro(P?, ch(PtoQ?, QtoP))), Fs1?, Fs2, _),
    social_graph(Id?, In?, Fs2?).

%% Receive introduction offer from friend
social_graph(Id, [msg(From, Id?, intro(Other, Ch))|In], Fs) :-
    ground(Id?), ground(From?), ground(Other?) |
   lookup_send(user, intro_offer(From?, Other?, Ch?), Fs?, Fs1, _),
    social_graph(Id?, In?, Fs1?).

%% User accepts introduction
social_graph(Id, [accept_intro(Other, ch(FIn, FOut))|In], Fs) :-
    ground(Id?), ground(Other?) |
    tag_stream(Other?, FIn?, Tagged),
    merge(In?, Tagged?, In1),
    social_graph(Id?, In1?, [(Other?, FOut?)|Fs?]).
```

When Bob types `introduce(alice, charlie)`, he creates a channel pair with writers `PtoQ` and `QtoP`. 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.

2.9.5 Network Switch Simulation. In deployment, agents communicate through a physical network. In simulation, a `network` process routes messages between agents. For two agents:

```
%% Alice sends to Bob
network2((alice, ch([msg(alice, bob, X)|AliceIn], AliceOut?)),
      (bob, ch(BobIn, [msg(alice, bob, X?)|BobOut?]))) :-
    network2((alice, ch(AliceIn?, AliceOut)),
             (bob, ch(BobIn?, BobOut))).

%% Bob sends to Alice
network2((alice, ch(AliceIn, [msg(bob, alice, X?)|AliceOut?])),
      (bob, ch([msg(bob, alice, X)|BobIn], BobOut?))) :-
    network2((alice, ch(AliceIn?, AliceOut)),
             (bob, ch(BobIn?, BobOut))).
```

For three agents, the switch routes between all pairs. We show two representative clauses:

```
%% Alice to Bob
network3((alice, ChA), (bob, ChB), (charlie, ChC)) :-
    ChA? = ch([msg(alice, bob, X)|InA], OutA),
    ChB? = ch(InB, [msg(alice, bob, X?)|OutB]) |
   network3((alice, ch(InA?, OutA?)), (bob, ch(InB?, OutB)), (charli

%% Bob to Charlie
network3((alice, ChA), (bob, ChB), (charlie, ChC)) :-
    ChB? = ch([msg(bob, charlie, X)|InB], OutB),
    ChC? = ch(InC, [msg(bob, charlie, X?)|OutC]) |
   network3((alice, ChA?), (bob, ch(InB?, OutB?)), (charlie, ch(InC?
```

The remaining four clauses (Alice to Charlie, Bob to Alice, Charlie to Alice, Charlie to Bob) follow the same structure. The network switch simulates the Network transaction introduced in Section 3.

2.9.6 Complete Plays. A *play* simulates a scenario by spawning agents and driving them with scripted *actors*.

Cold-Call Play. Alice initiates contact with Bob through the network:

```
play_cold_call :-
    network2((alice, ch(AliceNetOut?, AliceNetIn)),
             (bob, ch(BobNetOut?, BobNetIn))),
   agent(alice, ch(AliceUserIn?, AliceUserOut), ch(AliceNetIn?, Ali
   agent(bob, ch(BobUserIn?, BobUserOut), ch(BobNetIn?, BobNetOut))
    alice_actor(AliceUserOut?, AliceUserIn),
    bob_actor(BobUserOut?, BobUserIn).

alice_actor(_, [msg(user, alice, connect(bob))]).
bob_actor([msg(agent, user, befriend(From, Resp))|_],
      [msg(user, bob, decision(yes, From?, Resp?))|_]).
```

Alice's actor initiates connection; Bob's actor waits for the befriend request and accepts. The play terminates with both agents as friends.

Friend-Mediated Introduction Play. Bob, who knows both Alice and Charlie, introduces them:

```
play_introduction :-
    network3((alice, ch(AliceNetOut?, AliceNetIn)),
             (bob, ch(BobNetOut?, BobNetIn)),
          (charlie, ch(CharlieNetOut?, CharlieNetIn))),
   agent(alice, ch(AliceUserIn?, AliceUserOut), ch(AliceNetIn?, Ali
   agent(bob, ch(BobUserIn?, BobUserOut), ch(BobNetIn?, BobNetOut))
    agent(charlie, ch(CharlieUserIn?, CharlieUserOut),
          ch(CharlieNetIn?, CharlieNetOut)),
    alice_intro_actor(AliceUserOut?, AliceUserIn),
    bob_intro_actor(BobUserOut?, BobUserIn),
   charlie_intro_actor(CharlieUserOut?, CharlieUserIn).

bob_intro_actor(_, [introduce(alice, charlie)]).
alice_intro_actor([intro_offer(bob, charlie, Ch)|_], [accept_intro(
charlie_intro_actor([intro_offer(bob, alice, Ch)|_], [accept_intro(
```

Bob's actor initiates the introduction; Alice and Charlie's actors wait for the offer and accept. The play terminates with Alice and Charlie as direct friends, able to communicate without Bob's mediation.

3 Multiagent GLP

This section extends GLP to multiple agents. We first define multiagent transition systems, 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:

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

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 reader/writer pairs provide natural binary communication channels: when agent p binds a writer X shared with agent q, the corresponding reader 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 Network 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 $s0 = (\{\texttt{agent(ch(\_?,\_),ch(\_?,\_))}\}, \emptyset)$, 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 2.7).
(2) **Communicate** p **to** q**:** A binary transaction with participants $\{p, q\} \subseteq P$ 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 $c'_q = (G_q\{X? := T\}, \sigma_q)$. Note: this includes $p = q$.
(3) **Network** p **to** q**:** A binary transaction with participants $\{p, q\} \subseteq P$ 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 Reduce is unary while Communicate and Network are binary. Both Communicate and Network transfer assignments from writers to readers: Communicate operates between agents sharing logic variables, while Network operates through the network input/output streams established in each agent's initial configuration, enabling the cold-call protocol for connecting previously-disconnected agents.

3.4 The Multiagent Social Graph

The social graph code presented in Section 2.9 runs unchanged in maGLP. The difference is in how inter-agent communication is realized: rather than a simulated network switch, the maGLP Network transaction directly transfers messages between agents' network streams.

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

Similarly, friend-mediated introduction works 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 simulation in Section 2.9 is thus a faithful model of maGLP's Network transaction, allowing single-process testing of multiagent protocols.

3.5 Safety Properties of maGLP

The safety properties established for single-agent GLP extend to maGLP.

Proposition 3.7 (Safety Properties of maGLP)**.** *The safety properties established for GLP in Section 2 extend to maGLP:*

(1) ***SO Preservation*** *(cf. Proposition 2.11): If the initial goals of all agents satisfy SO, then every goal in every agent's resolvent throughout the run satisfies SO.*

(2) ***Acyclicity*** *(cf. Proposition* ***??****): If the initial goals of all agents contain no circular terms, then no goal in any agent's resolvent contains a circular term.*

(3) ***Monotonicity*** *(cf. Proposition 2.12): 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. The proofs are identical to those for single-agent GLP, substituting "agent p's resolvent" for "resolvent" and noting that Reduce transitions operate locally within each agent whilst Communicate and Network transitions preserve the properties through reader assignment transfer. □

Proposition 3.8 (maGLP Computation is Deduction)**.** *Let L be the transition system whose resolvent is the union of all local resolvents, the initial goal includes a* `network` *goal with channels paired to each agent's network channels, and the program is M augmented with the GLP definition of* `network`*. Then maGLP runs correspond to GLP runs of L, and their outcomes are logical consequences of the augmented program.*

4 maGLP is Grassroots

This section establishes that maGLP is grassroots. We first define protocols and the grassroots property, then prove that maGLP satisfies it, and finally show that any GLP application using cold calls inherits the grassroots property.

4.1 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 4.1 (Local-States Function)**.** A **local-states function** $S : 2^{\Pi} \mapsto 2^{\mathcal{S}}$ 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 4.2 (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 4.3 (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 $q \in P' \setminus P$.

Definition 4.4 (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.

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.

4.2 Transactions-Based Protocols are Oblivious

Definition 4.5 (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 4.6 (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 $R(P)$: $\mathcal{F}(P) := (S(P)^P, \{s0\}^P, R(P)\uparrow P)$.

Proposition 4.7 (Transactions-Based Protocols are Oblivious)**.** *A transactions-based protocol is oblivious.*

Definition 4.8 (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 $c'/P \notin C(P)$.

Theorem 4.9 (Interactive Transactions Induce Grassroots Protocols)**.** *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 4.7, $\mathcal{F}$ is oblivious. And since $\mathcal{F}$ is over an interactive set of transactions, $\mathcal{F}$ is interactive. Therefore, by Definition 4.4, $\mathcal{F}$ is grassroots. □

4.3 maGLP is Grassroots

We now prove that maGLP is grassroots.

Theorem 4.10 (maGLP is Grassroots)**.** *The maGLP protocol is grassroots.*

Proof. By Theorem 4.9, 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, Network) over a local-states function (asynchronous resolvents) with a common initial state. Hence maGLP is a transactions-based protocol (Definition 4.6).

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 $\texttt{msg}(q, X)$ on its network output stream, where X is a fresh writer. Then the Network transaction from p to q adds $X?$ to agent q's network input stream. Agent q can then execute Reduce transactions that bind $X?$ to 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 4.8).

By Theorem 4.9, maGLP is grassroots. □

4.4 GLP Applications Using Cold Calls are Grassroots

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

Definition 4.11 (GLP Application, Cold Call)**.** A **GLP application** is a GLP program M together with the maGLP infrastructure. An application **uses cold calls** if agents can execute the Network transaction to establish communication with previously-disconnected agents.

Proposition 4.12 (GLP Applications Using Cold Calls are Grassroots)**.** *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 Network transaction remains available, as cold calls are used. The proof of Theorem 4.10 relies only on the availability of the Network 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 4.7, it is oblivious. By the interactivity argument above, it is interactive. Therefore, by Definition 4.4, it is grassroots. □

This proposition provides a simple criterion for verifying that a GLP application is grassroots: if it uses cold calls for initial contact between disconnected agents, it inherits the grassroots property from maGLP.

Corollary 4.13 (The Grassroots Social Graph is Grassroots)**.** *The GLP implementation of the grassroots social graph (Section 2.9) is grassroots.*

Proof. The social graph uses cold calls for initial contact between disconnected agents. By Proposition 4.12, it is grassroots. □

5 Related Work

Grassroots platforms require agents to verify cryptographic identity and protocol compatibility upon contact, form authenticated channels, and coalesce spontaneously without global coordination. The language must support multiple concurrent platform instances and metaprogramming for tooling development. We examine how existing systems address these requirements.

Concurrent Logic Programming. GLP belongs to the family of concurrent logic programming (CLP) languages that emerged in the 1980s: Concurrent Prolog [47], GHC [59], and PARLOG [8]. These languages interpret goals as concurrent processes communicating through shared logical variables, using committed-choice execution with guarded clauses. Shapiro's comprehensive survey [49] documents this family and its design space.

A key evolution was *flattening*: restricting guards to primitive tests only. Flat Concurrent Prolog (FCP) [34] and Flat GHC [59]

demonstrated that flat guards suffice for practical parallel programming while dramatically simplifying semantics and implementation.

GLP can be understood as **Flat Concurrent Prolog with the Single-Reader Single-Writer (SRSW) constraint**. FCP introduced read-only annotations (?) distinguishing readers from writers of shared variables, enabling dataflow synchronization. However, read-only unification proved semantically problematic: Levi and Palamidessi [25] showed it is order-dependent, and Mierowsky et al. [34] documented non-modularity issues. GHC dispensed with read-only annotations entirely, relying on guard suspension semantics.

GLP's SRSW constraint—requiring that each variable has exactly one writer and one reader occurrence—resolves these difficulties by ensuring that (1) no races occur on variable binding, 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 [28, 43, 48], 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's work on moded Flat GHC [60, 62] 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's subsequent linearity analysis [61] identifies variables read exactly once, enabling compile-time garbage collection. GLP enforces both single-reader and single-writer universally as a syntactic restriction, whereas Ueda's system guarantees single-writer with single-reader as an optional refinement.

Distributed actor and process languages. Actor-based languages (Erlang/OTP [2], Akka [29], Pony [9]) and active object languages [4, 5] provide message-passing concurrency and fault isolation. However, their security models operate at the transport layer (TLS in Akka Remote [29], Erlang's cookie-based authentication [2]) rather than integrating cryptographic identity and code attestation into language primitives. Orleans [33] assumes trusted runtime environments, lacking the attestation mechanisms required for grassroots platforms where participants must verify code integrity without central coordination.

Capability security. E [35] provides capability-based security through unforgeable object references with automatic encryption. While ensuring object uniqueness and access control, E does not address verifying real-world identity or protocol implementation attestation—distinct requirements for grassroots platforms.

Linear types and session types. Linear types [63] ensure single-use of resources, similar to GLP's single-writer constraint. However, GLP's SRSW mechanism provides bidirectional pairing—each writer has exactly one reader—enabling authenticated channels without type-level tracking. Session types [19] specify communication protocols statically, with implementations in Links [10, 30], Rust [20], Scala [44], and Go [7]. While these verify protocol conformance at compile time, GLP's reader/writer synchronization enforces protocol dynamically through suspension and resumption, and runtime attestation enables participants to verify protocol compatibility when establishing connections between independently-deployed agents.

Concurrent coordination languages. Concurrent ML [40] provides first-class synchronous channels and events. The Join Calculus [14] offers pattern-based synchronization through join patterns. GLP's SRSW variables provide asynchronous communication through reader/writer pairs with the monotonicity property (Proposition 2.12) ensuring suspended goals remain reducible once readers are instantiated. However, neither provides mechanisms for cryptographic identity verification or authenticated channel establishment required for grassroots platforms.

Blockchain programming languages. Smart contract languages like Solidity [36] and Move [58] provide deterministic execution and asset safety but assume blockchain infrastructure for identity and consensus. While Scilla [45] separates computation from communication similar to GLP's message-passing model, it targets on-chain state transitions rather than peer-to-peer authenticated channels. GLP achieves security properties through the language-level SRSW invariant and attestations, without requiring global consensus.

Authorization languages. OPA/Rego [38] and Cedar [18] provide declarative policy specification but are specialized for policy evaluation. They consume authentication tokens as inputs but do not integrate attestation as first-class primitives for verifying remote code execution.

Grassroots platforms. The mathematical foundations for grassroots platforms were established in [50], with atomic transactions introduced in [53]. Grassroots social networks [51], cryptocurrencies [26, 52], and federations [16, 54] have been formally specified and proven grassroots. GLP provides the programming language to implement these specifications.

6 Conclusion

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

(1) **Mathematical Foundations:** We recalled the framework of multiagent transition systems with atomic transactions and the key theorem that protocols over interactive transactions are grassroots.
(2) **GLP Definition:** We defined GLP as an extension of logic programs with reader/writer variable pairs satisfying the Single-Reader/Single-Writer (SRSW) restriction, providing asynchronous communication through term matching rather than unification.
(3) **Grassroots Proof:** We proved that multiagent GLP (maGLP) is grassroots by showing it is a transactions-based protocol with interactive transactions (via the Network transaction for cold calls).
(4) **Application Criterion:** We established that any GLP application using cold calls inherits the grassroots property from maGLP.
(5) **Grassroots Social Graph:** We demonstrated the utility of GLP by implementing the grassroots social graph protocol and proving it grassroots.

The grassroots social graph serves as the infrastructure layer for other grassroots platforms. Future work includes implementing grassroots social networks, grassroots cryptocurrencies, and grassroots federations in GLP, as well as developing the Dart/Flutter implementation for smartphone deployment.

References

[1] Almeida, P. S. and Shapiro, E. 2024. The blocklace: A byzantine-repelling and universal conflict-free replicated data type. *arXiv preprint arXiv:2402.08068*.

[2] Armstrong, J. 2013. *Programming Erlang: Software for a Concurrent World*, 2nd ed. Pragmatic Bookshelf.

[3] Azadbakht, K., de Boer, F. S., Bezirgiannis, N., and de Vink, E. 2020. A formal actor-based model for streaming the future. *Science of Computer Programming 186*, 102341.

[4] Boer, F. d., Damiani, F., Hähnle, R., Johnsen, E. B., and Kamburjan, E. 2024. *Active Object Languages: Current Research Trends*. Vol. 14360. Springer.

[5] Boer, F. D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C. C., Johnsen, E. B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., et al. 2017. A survey of active object languages. *ACM Computing Surveys (CSUR) 50*, 5, 1–39.

[6] Buterin, V. 2018. Governance, part 2: Plutocracy is still bad. Available at https://vitalik.eth.limo/general/2018/03/28/plutocracy.html.

[7] Castro-Perez, D., Hu, R., Jongmans, S.-S., Ng, N., and Yoshida, N. 2019. Distributed programming using role-parametric session types in go. *Proceedings of the ACM on Programming Languages 3*, POPL, 29:1–29:30.

[8] Clark, K. and Gregory, S. 1986. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems (TOPLAS) 8*, 1, 1–49.

[9] Clebsch, S., Drossopoulou, S., Blessing, S., and McNeil, A. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*. ACM, 1–12.

[10] Cooper, E., Lindley, S., Wadler, P., and Yallop, J. 2007. Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*. Springer, 266–296.

[11] Costan, V. and Devadas, S. 2016. Intel sgx explained. In *Cryptology ePrint Archive*.

[12] Dauth, T. and Sulzmann, M. 2019. Futures and promises in haskell and scala. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 68–74.

[13] Diffie, W. and Hellman, M. 1976. New directions in cryptography. *IEEE transactions on Information Theory 22*, 6, 644–654.

[14] Fournet, C. and Gonthier, G. 1996. The reflexive cham and the join-calculus. *Proceedings of POPL'96*, 372–385.

[15] Garay, J., Kiayias, A., and Leonardos, N. 2015. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 281–310.

[16] Halpern, D., Procaccia, A. D., Shapiro, E., and Talmon, N. 2024. Federated assemblies.

[17] Hankerson, D., Menezes, A. J., and Vanstone, S. 2004. *Guide to Elliptic Curve Cryptography*. Springer.

[18] Hicks, C., Datta, A., He, K., Kasampalis, J., Khanna, N., Lampson, M., Lee, W., Mehta, S., Rengarajan, A., Thakkar, E., Vanciu, R., and Warden, A. 2023. Cedar: A new language for expressive, fast, safe, and analyzable authorization. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[19] Honda, K. 1993. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR)*. Lecture Notes in Computer Science, vol. 715. Springer, 509–523.

[20] Jespersen, T. B. L., Munksgaard, P., and Larsen, K. F. 2015. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming (WGP)*. ACM, 13–22.

[21] Keidar, I., Lewis-Pye, A., and Shapiro, E. 2025. Constitutional consensus.

[22] Keidar, I., Naor, O., and Shapiro, E. 2023. Cordial miners: A family of simple and efficient consensus protocols for every eventuality. In *37th International Symposium on Distributed Computing (DISC 2023)*. LIPICS.

[23] Kowalski, R. 1974. Predicate logic as programming language. In *IFIP congress*. Vol. 74. 569–574.

[24] Lamport, L., Shostak, R., and Pease, M. 1982. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3, 382–401.

[25] Levi, G. and Palamidessi, C. 1985. The declarative semantics of read-only annotations in logic programming. *Proceedings of the 1985 Symposium on Logic Programming*, 212–220.

[26] Lewis-Pye, A., Naor, O., and Shapiro, E. 2023. Grassroots flash: A payment system for grassroots cryptocurrencies. *arXiv preprint arXiv:2309.13191*.

[27] Lewis-Pye, A. and Shapiro, E. 2025. Morpheus consensus: Excelling on trails and autobahns. *arXiv preprint arXiv:2502.08465*.

[28] Lichtenstein, Y. and Shapiro, E. 1988. Concurrent algorithmic debugging. *ACM SIGPLAN Notices 24*, 1, 248–260.

[29] Lightbend Inc. 2022. Akka: Build concurrent, distributed, and resilient message-driven applications. https://akka.io. Accessed October 2025.

[30] Lindley, S. and Morris, J. G. 2017. Lightweight functional session types. *Behavioural Types: From Theory to Tools*, 265–286. Chapter in edited volume.

[31] Lloyd, J. W. 1987. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag.

[32] Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. 1996. *Handbook of Applied Cryptography*. CRC press.

[33] Microsoft. 2022. Orleans: Cloud native application framework. https://dotnet.github.io/orleans. Accessed October 2025.

[34] 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.

[35] Miller, M. S. 2006. Robust composition: Towards a unified approach to access control and concurrency control. Ph.D. thesis, Johns Hopkins University.

[36] Mukhopadhyay, M. 2018. *Ethereum Smart Contract Development: Build blockchain-based decentralized applications using solidity*. Packt Publishing Ltd.

[37] Nakamoto, S. and Bitcoin, A. 2008. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf 4*.

[38] Open Policy Agent Contributors. 2021. Open policy agent. https://www.openpolicyagent.org. Cloud Native Computing Foundation.

[39] Raman, A., Joglekar, S., Cristofaro, E. D., Sastry, N., and Tyson, G. 2019. Challenges in the decentralised web: The mastodon case. In *Proceedings of the internet measurement conference*. 217–229.

[40] Reppy, J. H. 1999. *Concurrent Programming in ML*. Cambridge University Press.

[41] Rivest, R. L., Shamir, A., and Adleman, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2, 120–126.

[42] Sabt, M., Achemlal, M., and Bouabdallah, A. 2015. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. IEEE, 57–64.

[43] Safra, S. and Shapiro, E. 1988. Meta interpreters for real. In *Concurrent Prolog: Collected Papers*. MIT Press, 166–179.

[44] Scalas, A. and Yoshida, N. 2016. Lightweight session programming in scala. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl, 21:1–21:28.

[45] Sergey, I., Kumar, A., and Hobor, A. 2019. Scilla: a smart contract intermediate-level language. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 366–381.

[46] Shapiro, E. 1982. *Algorithmic Program Debugging*. MIT Press.

[47] Shapiro, E. 1983. A subset of concurrent prolog and its interpreter. *ICOT Technical Report, TR-003*.

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

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

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

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

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

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

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

[55] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.

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

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

[58] The Diem Association. 2022. Move: A language with programmable resources. https://github.com/move-language/move. Accessed October 2025.

[59] Ueda, K. 1986. Guarded horn clauses. In *Logic Programming '85*. Lecture Notes in Computer Science, vol. 221. Springer, 168–179.

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

[61] Ueda, K. 2001. Resource-passing concurrent programming. *Proceedings of TACS*

2001, 95–126.
[62] Ueda, K. and Morita, M. 1995. I/o mode analysis in concurrent logic programming. In *Proceedings of the International Symposium on Theory and Practice of Parallel Programming*. Springer, 356–368.
[63] Wadler, P. 1990. Linear types can change the world. *Programming concepts and methods 2*, 347–359.
[64] Zuboff, S. 2019. *The age of surveillance capitalism: The fight for a human future at the new frontier of power.* Public Affairs, US.

A Proofs

*

Proof. We prove by induction on the length of the run that each step preserves logical consequence.

Base case. For $n = 0$, we have $G_0 = G_0$ with empty substitution ϵ. The outcome $(G_0 :\text{-} G_0)$ is a tautology, hence a logical consequence of any program.

Inductive step. Assume the proposition holds for runs of length k. Consider a proper run of length $k + 1$:

$$\rho : G_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_k} G_k \xrightarrow{\sigma_{k+1}} G_{k+1}$$

By the inductive hypothesis, $(G_0 :\text{-} G_k)\sigma'$ is a logical consequence of M, where $\sigma' = \sigma_1 \circ \cdots \circ \sigma_k$.

For the transition $G_k \xrightarrow{\sigma_{k+1}} G_{k+1}$:

- There exists atom $A \in G_k$ and clause $(H :\text{-} B) \in M$ renamed apart
- σ_{k+1} is the mgu of A and H
- $G_{k+1} = (G_k \setminus \{A\} \cup B)\sigma_{k+1}$

Since $(H :\text{-} B)$ is a clause in M and σ_{k+1} unifies A with H, we know that:

- The instance $(H :\text{-} B)\sigma_{k+1}$ is a logical consequence of M (by instantiation of a program clause)
- Since $A\sigma_{k+1} = H\sigma_{k+1}$ (by the mgu property), we can replace A with B under substitution σ_{k+1}
- Therefore, the implication $(G_k :\text{-} G_{k+1})$ is a logical consequence of M when we consider that G_{k+1} was obtained by replacing A in G_k with B and applying σ_{k+1}

By the transitivity of logical consequence, if $(G_0 :\text{-} G_k)\sigma'$ is a logical consequence of M and $(G_k :\text{-} G_{k+1})$ follows from M under the additional substitution σ_{k+1}, then $(G_0 :\text{-} G_{k+1})(\sigma' \circ \sigma_{k+1})$ is a logical consequence of M.

Since $\sigma = \sigma' \circ \sigma_{k+1} = \sigma_1 \circ \cdots \circ \sigma_{k+1}$, we conclude that the outcome $(G_0 :\text{-} G_{k+1})\sigma$ is a logical consequence of M. □ □

*

Proof. Consider the transition $G_i \to G_{i+1}$ via reduction of some atom $A' \in G_i$ with clause C. Let $(H :\text{-} B)$ be the renaming of C apart from A', with writer mgu σ and reader counterpart $\sigma?$.

By Definition **??**, the Reduce transition specifies that $G_{i+1} = (G_i \setminus \{A'\} \cup B)\sigma$, and the configuration's reader substitution is updated with $\sigma?$.

For any atom $A \in G_{i+1}$ that also appeared in G_i, we have:

(1) $A \neq A'$ (A was not the reduced atom). Then $A \in G_i \setminus \{A'\}$. The reduction applies σ to all atoms in the resolvent. Since A was in G_i and the clause was renamed apart from the entire goal (including A), any writers in A are distinct from V_σ. Therefore σ does not instantiate variables in A. Only the reader counterpart $\sigma?$ can affect A. Since $\sigma?$ is a reader substitution with $V_{\sigma?} \subset V?$, we have A in G_{i+1} equals $A'\tau$ where $A' \in G_i$ and $\tau = \sigma?$ instantiates only readers.
(2) $A = A'$ (A was the reduced atom). This case cannot occur since A' is removed from the resolvent during reduction and thus cannot appear in G_{i+1}.

Therefore, any atom persisting from G_i to G_{i+1} is instantiated only by the reader substitution $\sigma?$. □ □

*

Proof. Follows from the correspondence between GLP reductions and LP reductions on pure logic variants, combined with Proposition **??**. □ □

*

Proof. By induction on run length. The base case holds by assumption: G_0 satisfies SO.

For the inductive step, assume G_i satisfies SO and consider $G_i \to G_{i+1}$ via reduction of atom $A \in G_i$ with clause C satisfying the SRSW syntactic restriction. Let $(H :\text{-} B)$ be the renaming of C apart from G_i, with writer mgu σ.

Since the clause satisfies SRSW, it satisfies SO: every variable in C occurs exactly once. After renaming apart, $(H :\text{-} B)$ also satisfies SO with fresh variables.

Since G_i satisfies SO and $A \in G_i$, every variable in A occurs exactly once in G_i. The writer mgu σ maps variables in A to subterms of H (and vice versa). By SO of both A and H, each such variable occurs exactly once on each side.

The new goal $G_{i+1} = (G_i \setminus \{A\} \cup B)\sigma$ is formed by:

(1) Removing A from G_i (eliminating all variable occurrences in A)
(2) Adding B (introducing fresh variables, each occurring once by SO of B)
(3) Applying σ (substituting variables with terms, not duplicating occurrences)

Since removal and substitution do not duplicate variable occurrences, and B's fresh variables each occur once, G_{i+1} satisfies SO. □ □

*

Proof. By induction on run length. For the base case, G_0 contains no circular terms by assumption. For the inductive step, assume G_i contains no circular terms and consider the transition $G_i \to G_{i+1}$ via reduction of atom A with clause C. Let $(H :\text{-} B)$ be the renaming of C apart from A, with writer mgu σ and reader counterpart $\sigma?$. The reader counterpart exists only if for all $X \in V_\sigma$, $X? \notin X\sigma$ (occurs check). This ensures no writer is bound to a term containing its paired reader. Since $G_{i+1} = (G_i \setminus \{A\} \cup B)\sigma?$, and the occurs check prevents circular assignments, G_{i+1} contains no circular terms. □ □

*

Proof. By induction on $j - i$. For the base case ($j = i$), the atom $A \in G_i$ can reduce with C by assumption. For the inductive step, assume the property holds for $j = k$ and consider $j = k + 1$.

If A was reduced at some step between i and k, then case (1) holds. Otherwise, by the inductive hypothesis, there exists $A' \in G_k$ where $A' = A\tau$ for some reader substitution τ, and A' can reduce with C.

Consider the transition $G_k \rightarrow G_{k+1}$. If the reduction involves A', then case (1) holds for $j = k + 1$. If the reduction involves a different atom $B \in G_k$, then A' persists in G_{k+1}, possibly further instantiated. Specifically, the reduction applies substitution $\sigma?$ where $\sigma?$ instantiates only readers (by definition of reader counterpart). Thus there exists $A'' \in G_{k+1}$ where $A'' = A'\sigma? = A(\tau \circ \sigma?)$, and $\tau \circ \sigma?$ is a reader substitution.

Since A' could reduce with C (renamed apart) via some writer mgu at step k, and $\sigma?$ only instantiates readers, the unification of A'' with the head of C (appropriately renamed) still succeeds: reader instantiation preserves unifiability and cannot introduce new writer instantiation requirements. Therefore A'' can reduce with C at step $k + 1$. □ □

*

Proof. We prove that maGLP is oblivious and interactive.

(1) **maGLP is Oblivious:** Follows directly from Proposition 4.7.
(2) **maGLP is Interactive:** We have to show that in any configuration c of a run of maGLP over P, if this configuration is in fact configuration over $P' \supset P$, then members of P have a behaviour not available to them if this was a run over P. The answer, of course, is that in such a case any agent $q \in P' \setminus P$ can send a network message to some agent $p \in P$, resulting in the local state of p having an 'alien trace'—a variable produced by an agent not in P—a behaviour not available to P on their own.

We conclude that maGLP is grassroots. □ □

B Grassroots Social Graph Protocol Properties

B.1 Non-blocking Operation Through Variable Synchronization

The social graph protocol achieves non-blocking operation through careful use of unbound variables and the `inject` procedure. When initiating connections, agents send offers containing unbound response variables and continue processing other messages while awaiting responses. Similarly, when receiving offers, agents query their users for approval without blocking the main protocol loop.

The `inject` procedure in Program **??** implements deferred message insertion: when `X` is unbound, `inject` passes input stream messages to its output whilst waiting for `X` to become bound. Once `X` is known, it inserts the message and terminates. This ensures the protocol remains responsive while awaiting responses to connection attempts, preventing any single pending operation from blocking the entire message processing loop.

B.2 Protocol Properties

The social graph protocol exhibits several essential properties for grassroots platforms. Non-blocking operation ensures that agents remain responsive during connection establishment, with no single operation capable of indefinitely blocking message processing. Symmetric channel establishment guarantees that successful connections result in bidirectional communication with identical capabilities for both parties. The unified message processing through stream merging provides fair handling of messages from all sources, preventing starvation of any input source.

The protocol's use of unbound variables for response coordination elegantly solves the distributed consensus problem for connection establishment. Both agents must explicitly agree to connect—the offerer by initiating and the receiver by accepting—with the shared response variable serving as the synchronization mechanism. This design ensures that connections only form through mutual consent while avoiding complex state machines or timeout mechanisms.

The friends list serves multiple roles simultaneously: it represents the agent's local view of the social graph, provides the routing table for message sending, and maintains the state needed for friend-mediated introductions. This unified structure simplifies reasoning about the protocol while enabling efficient implementation. The incremental construction of the social graph through individual connections allows multiple disconnected components to form independently and later merge through cross-component connections, embodying the grassroots principle of spontaneous emergence without central coordination.

C Social Networking Applications

Building upon the authenticated social graph, this section demonstrates how GLP enables secure social networking applications. The established friend channels and attestation mechanisms provide verifiable content authorship and provenance guarantees impossible in centralised platforms.

C.1 Direct Messaging

Direct messaging establishes dedicated conversation channels between friends, separate from the protocol control channels. When accepting friendship, the acceptor creates a messaging channel and includes it in the acceptance response:

Program 1: Direct Messaging Channel Establishment

```
% Modified establishment for direct messaging
% Secure version - verifies DM channel attestation
establish(yes, From, Resp, Fs, Fs1, In, In1) :-
    new_channel(ch(FIn, FOut), FCh),
    new_channel(ch(DMIn, DMOut), DMCh),
    Resp = accept(FCh, DMCh),
   attestation(DMCh, att(From, _)) |  % Verify DM channel from authen
   handle_friend(From?, FIn?, FOut?, DMIn?, DMOut?, Fs?, Fs1, In?, In

handle_friend(From, FIn, FOut, DMIn, DMOut, Fs,
        [(From, FOut), (dm(From), DMOut)|Fs], In, In1) :-
    tag_stream(From?, FIn?, Tagged),
    merge(In?, Tagged?, In1),
    forward_to_app(dm_channel(From?, DMIn?)).
```

The protocol maintains separation between control and messaging channels. The friend channel handles protocol messages whilst the direct messaging channel carries conversation data. Each message through the DM channel carries attestation, ensuring non-repudiation and authenticity of the conversation history.

C.2 Feed Distribution with Verified Authorship

Content feeds leverage the ground guard's relaxation of SRSW constraints to broadcast to multiple followers whilst maintaining cryptographic proof of authorship:

Program 2: Authenticated Feed Distribution

```
% Post distribution with attestation preservation
post(Content, Followers, Followers1) :-
    ground(Content), current_time(Time) |
    create_post(Content?, Time?, Post),
    broadcast(Post?, Followers?, Followers1).

broadcast(_, [], []).
broadcast(Post, [(Name,Out)|Fs], [(Name,[Post|Out1?])|Fs1]) :-
    broadcast(Post?, Fs?, Fs1).

% Defined guard for attestation preservation
preserve_attestation(Post, Author, forward(Author?, Post)).

% Forward with attestation verification
forward(Post, Followers, Followers1) :-
    ground(Post), attestation(Post, att(Author, _)),
    preserve_attestation(Post?, Author?, Forward) |
    broadcast(Forward?, Followers?, Followers1).
```

Each post carries the creator's attestation $(Post)_{M,p,q}$. When forwarding, the original attestation is preserved whilst adding the forwarder's attestation, creating a cryptographically verifiable provenance chain. Recipients can verify both the original author and the complete forwarding path, preventing misattribution and enabling accountability for content distribution.

C.3 Group Communication

Groups in GLP follow a founder-administered model where users create groups with selected friends. The founder invites authenticated friends who decide whether to join. Group messages use interlaced streams, creating natural causal ordering without consensus.

Group Formation. Users initiate groups with a name and friend list. The globally unique group identifier is (founder, name), preventing naming conflicts:

Program 3: Group Formation Protocol

```
% User creates group with friend list
social_graph(Id, [msg(user, Id, create_group(Name, Friends))|In], Fs) :-
    create_group_streams([Id|Friends]?, Streams),
   send_invitations(Friends?, Id?, Name?, Streams?, Fs?, Fs1),
   social_graph(Id, In?, [((Id,Name), group(admin, Streams?))|Fs1?]).

% Send invitations through friend channels
send_invitations([], _, _, _, Fs, Fs).
send_invitations([Friend|Friends], Founder, Name, Streams, Fs, Fs1) :-
    lookup(Friend, Fs?, Ch),
    Ch = [inv(Founder?, Name?, Streams?)|Ch1?],
   send_invitations(Friends?, Founder?, Name?, Streams?, [(Friend,Ch1?)|Fs2?], Fs1).

% Receive invitation from friend
social_graph(Id, [msg(From, Id, inv(Founder, Name, Streams))|In], Fs) :-
   attestation(inv(Founder, Name, Streams), att(From, _)) |
    lookup_send(user, msg(agent, user,
          join_group(From?, Founder?, Name?)), Fs?, Fs1),
    social_graph(Id, In?, Fs1?).

% User decision on invitation
social_graph(Id, [msg(user, Id, join(yes, Founder, Name, Streams))|I
   social_graph(Id, In?, [((Founder,Name), group(member, Streams?))
social_graph(Id, [msg(user, Id, join(no, _, _, _))|In], Fs) :-
    social_graph(Id, In?, Fs?).
```

The founder creates interlaced stream structures for all members and sends invitations through authenticated friend channels. Recipients verify the invitation's attestation before consulting their user. Accepted groups are stored with key (Founder, Name), ensuring uniqueness whilst clarifying ownership.

Group Messaging via Interlaced Streams. Group members maintain independent message streams whilst observing others' messages, creating causal ordering through the interlaced streams mechanism:

Program 4: Group Messaging

```
% Member participates in group
group_member(Id, (Founder, Name), Streams) :-
    lookup((Founder,Name), Fs?, group(Role, Streams)),
    compose_messages(Id?, Name?, Messages),
    find_my_stream(Id?, Streams?, MyStream),
    interlace(Messages?, MyStream?, [], Streams?).

compose_messages(Id, Name, [Msg|Msgs]) :-
    await_user_input(Id?, Name?, Input),
    format_message(Input?, Id?, Msg),
    compose_messages(Id?, Name?, Msgs?).
compose_messages(_, _, []).

format_message(reply(Text), Id, msg(Id, reply, Text)).
format_message(post(Text), Id, msg(Id, post, Text)).
```

Members post independently without control tokens. The interlaced streams mechanism (Program E.8) ensures each member's block references all observed messages. When member p replies to message m, the reply appears in p's stream only after p has observed m, creating natural causality where replies follow what they reply to whilst independent messages remain unordered.

Security derives from authenticated friend channels—all group communication occurs through channels established via the social graph protocol, with automatic attestation on every message. Byzantine agents outside the group cannot inject messages as they lack authenticated channels to members. The interlaced structure provides causal consistency equivalent to consensus protocols whilst eliminating their overhead, demonstrating how authenticated channels combined with GLP's concurrent programming primitives enable efficient group communication without centralisation or Byzantine agreement.

C.4 Content Authenticity and Provenance

Content authenticity in GLP derives from the attestation mechanism applied recursively through forwarding operations. When agent p creates post P, it carries attestation $(P)_{M,p,*}$. When agent q forwards this post, the forward operation wraps the entire attested

post: 'forward(p, P)', which receives attestation $(forward(p, P))_{M,q,*}$. Recipients can verify both q's forwarding attestation and p's original creation attestation, with the nesting depth revealing the complete forwarding chain.

This mechanism addresses three vulnerabilities in conventional social networks. First, impersonation becomes cryptographically impossible—agents cannot forge attestations for other agents' keys. Second, misattribution is prevented—the original author's attestation remains embedded regardless of forwarding depth. Third, conversation manipulation is detectable—group messages through interlaced streams create a tamper-evident partial order where altered histories fail attestation verification. These properties emerge from the language-level integration of attestations with GLP's communication primitives, requiring no external trust infrastructure or consensus protocols.

D 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 following builtin guards are fundamental for concurrent GLP programming:

- `ground(X)` succeeds if `X` is ground (contains no variables).
- `known(X)` succeeds if `X` is not a variable, though it may not be ground.
- `writer(X)` and `reader(X)` succeed if `X` is an uninstantiated writer or reader respectively. Note that `reader(X)` is non-monotonic.
- `otherwise` succeeds if all previous clauses for this procedure failed.
- `X =?= Y` 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 `X < Y`, `X > Y`, `X =< Y`, `X >= Y`, `X =:= Y`, `X =\= Y` succeed if both arguments are ground numbers satisfying the relation.

When a guard's success condition implies that a variable is ground, the clause body may contain multiple occurrences of that variable's reader without violating the single-writer requirement, enabling safe replication of ground terms to multiple concurrent consumers.

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 `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`.

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

- `evaluate(Expr?,Result)` evaluates ground arithmetic expressions.
- `current_time(T)` provides system timestamps for temporal coordination.
- `variable_name(X,Name)` returns a unique identifier for variable `X` and its pair.

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.

E 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.

E.1 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 5: 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 6: Stream-Channel Relay

```
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.

E.2 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 7: Stream Tagging

```
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.

E.3 Stream Observation

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

Program 8: Concurrent Observer

```
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).
```

E.4 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 9: Cooperative Producers

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

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.

E.5 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 10: 3-Way Network Switch

```
% 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?)).
```

E.6 Implementation Correctness Properties

Proposition E.1 (Goal State Integrity)**.** *For any configuration (R_p, V_p, M_p) where $R_p = (A_p, S_p, F_p)$ in an IRmaGLP run, every goal of agent p appears in exactly one of A_p, S_p, or F_p. Furthermore, F_p is monotonically increasing: once a goal enters F_p, it remains there.*

Proof. By induction on transition steps. Initially all goals are in A_p. The Reduce transaction (Definition H.9) moves goals between sets atomically: from A_p to S_p on suspension, from S_p to A_p on reactivation, and to F_p on failure. No transition removes goals from F_p. □

Proposition E.2 (SRSW Preservation in Implementation)**.** *If the initial configuration of IRmaGLP satisfies SRSW, then for any reachable configuration and any variable Y, at most one agent holds Y locally (in their resolvent) and at most one agent holds Y' locally.*

Proof. The variable table V_p tracks all non-local variable references. When agent p exports a variable Y through the export helper (Definition H.3), Y is added to V_p marking it as created by p but referenced externally. The Communicate and Network transactions maintain exclusivity by transferring variables between agents rather than duplicating them. The export helper's relay mechanism for requested readers preserves the single-reader property through fresh variable pairs. □

Proposition E.3 (Suspension Correctness)**.** *If goal G is suspended on reader set W at agent p, then G transitions to active exactly when either: (1) some $X? \in W$ receives a value through a Communicate transaction, or (2) some $X? \in W$ is abandoned.*

Proof. The reactivate helper (Definition H.3) is called precisely when assignments arrive or abandonment occurs. It removes (G, W) from S_p if $X? \in W$, adding G to the tail of A_p. No other operation modifies suspended goals. □

E.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 11: 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.

E.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 12: 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?]))

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.

E.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 [28, 43, 46, 56, 57], as well as writing a GLP operating system in GLP [48]. 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 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 13: GLP plain metainterpreter

```
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?,
```

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 14: GLP fail-safe metainterpreter

```
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 15: GLP metainterpreter with run control

```
run(true,_).  % halt
run((A,B),Cs) :- distribute(Cs?,Cs1,Cs2), run(A?,Cs1?), run(B?,Cs1).
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?).

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

The metainterpreter suspends reductions as soon as the control stream is bound to `[suspend|Cs?]`, upon which the run can be resumed or aborted by binding `Cs` accordingly. Combining Programs E.9 and E.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 16: GLP termination-detecting metainterpreter

```
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 bound to `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 17: GLP metainterpreter with run control and snapshot collection

```
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 bound to `[suspend,abort]`, the reader `R?` will be bound to 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 18: GLP a tracing metainterpreter

```
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 [48], 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 19: GLP metainterpreter with runtime control

```
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?) % r
run(A,[abort|Cs],[A?|R?],R). % abort and dump
run(A,[suspend|Cs],L?,R) :- suspended_run(A?,Cs?,L,R?). % suspend

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 [57] 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).

F Securing Multiagent GLP

F.1 Secure Multiagent GLP

Here we assume that each agent $p \in \Pi$ has a self-chosen keypair, unique with high probability, and identify p with its public key. Agents learn public keys through two mechanisms: existing social channels (exchanging keys in person, via email, phone numbers, or other trusted communication methods outside the protocol)

and friend-mediated introductions within the protocol itself. In cold calls, agents initiate connections only with those whose public keys they have verified through external channels. Friend-mediated introductions (Appendix B) provide an additional trust propagation mechanism, where mutual friends vouch for the cryptographic identities of introduced parties, enabling the social graph to expand through existing trust relationships.

In addition to the standard cryptographic assumptions on the security of encryption and signatures, we assume that the underlying GLP execution mechanism can produce *attestations*: A proof that a network message $\texttt{msg}(q, X)$ or a substitution message $\{X? := T\}$ was produced by module M as a result of a correct goal/clause reduction. For such a message E, we denote by E_M the message together with its attestation, and by $E_{M,p}$ such an attestation further signed by agent p's private key. Furthermore, we assume that when such a signed attestation is sent to agent q, it is encrypted with q's public key, denoted $E_{M,p,q}$. In summary, each message $\texttt{msg}(q, X)$ or assignment to X produced by agent p using module M is sent to the intended recipient q or the holder q of X? attested by M, signed by p and encrypted for q. (See Section **??** for smartphone-specific implementation of these security mechanisms.)

Programs require the ability to inspect attestations on received messages and identify their own module for protocol decisions. GLP provides guard predicates for security operations:

- `attestation(X, Info)` succeeds if `X` carries an attestation, assigning to `Info` a term `att(Agent, Module)` containing the attesting agent's public key and module identifier. For locally-produced terms, `Agent` binds to the distinguished constant `self`.
- `module(M)` binds `M` to the identifier of the currently executing module. Agents use this guard to determine their own module identity when evaluating compatibility with other agents' attested modules. Module identifiers include version information enabling compatibility verification between different protocol versions.

These guards enable programs to make protocol decisions based on attestation properties and module compatibility without accessing the underlying cryptographic mechanisms directly. The social graph protocol uses these to verify cold call origins and enforce module compatibility, whilst social networking applications extract and preserve provenance chains when forwarding content.

While the formal specification requires attestation, signature and encryption for every message, a practical implementation should employ standard cryptographic optimizations [32]: Attestation can be required only on initial contact and then verified intermittently rather than for every message, reducing computational overhead while maintaining security guarantees. Public keys exchanged during initial attestation can be used to establish secure agent-to-agent communication channels using ephemeral session keys through protocols such as Diffie-Hellman key exchange [13] or ECDH [17], providing perfect forward secrecy while reducing the cost of encryption operations. These optimizations are transparent to the GLP program level, where the security properties continue to hold as specified.

F.2 Program-Independent Security Properties

The cryptographic mechanisms of secure maGLP guarantee three fundamental properties for all executions, regardless of the specific GLP program:

(1) **Integrity:** Any entity $E_{M,p,q}$ transmitted from agent p to agent q either arrives unmodified or is rejected upon signature verification failure. Tampering with E invalidates p's signature, which cannot be forged without p's private key.
(2) **Confidentiality:** The content of $E_{M,p,q}$ remains inaccessible to all agents except q, as decryption requires q's private key. Combined with the SRSW invariant ensuring exclusive reader/writer pairing, this prevents both direct cryptographic attacks and indirect access through shared variables.
(3) **Non-repudiation:** Agent p cannot deny sending any entity successfully verified as $E_{M,p,q}$, as the valid signature constitutes cryptographic proof of authorship that only p could have created.

These properties provide the cryptographic foundation for secure maGLP communication. Authentication and trust propagation properties depend on program-specific behaviour and are analysed for particular protocols such as the grassroots social graph.

F.3 Security of the Social Graph Protocol

Authenticated Connection Establishment. Cold call offers carry attestation $(\texttt{msg}(q, \texttt{offer(Resp)}))_{M,p,q}$ proving agent p executes module M. Acceptance returns $(\texttt{Resp:=accept(FCh)})_{M,q,p}$, establishing mutual authentication. The signature mechanism proves control of private keys and attestation verifies code execution, but neither establishes real-world identity—this requires external verification through existing social channels. Attestations include module identifiers, enabling compatibility verification between protocol versions.

Trust Propagation. Friend-mediated introductions strengthen identity assurance. When p introduces friends q and r, recipients verify the introduction originates from p through attestation. The established channel provides ongoing mutual attestation. The introducer vouches for cryptographic-to-social identity mappings, combining cryptographic proof with social trust.

Attack Prevention. The protocol prevents three attack categories through integrated cryptographic and language-level mechanisms. Sybil attacks are mitigated through the requirement that agents know each other's public keys through external social verification before connecting—an adversary cannot create meaningful fake identities without corresponding social relationships. Man-in-the-middle attacks fail because messages are encrypted for specific recipients and the SRSW invariant ensures exclusive reader/writer channels that cannot be intercepted. Impersonation attempts are detected through signature verification on every message, with invalid signatures causing silent drops. These mechanisms combine to ensure that successful communication occurs only between authenticated parties running verified code.

F.4 Blockchain Security of GLP Streams

Authenticated GLP streams achieve blockchain security properties [15, 37] through language-level guarantees:

(1) **Immutability:** Once a stream element `[X|Xs]` is created with `X` bound to value `T`, the single-assignment semantics of logic variables prevents any subsequent assignment of `X`. This provides immutability without cryptographic hashing.
(2) **Unforkability:** The SRSW invariant ensures each writer `Xs` has exactly one occurrence. Attempting to create two continuations `Xs=[Y|Ys]` and `Xs=[Z|Zs]` would require two occurrences of writer `Xs`, violating SRSW. This prevents forks at the language level.
(3) **Non-repudiation:** Stream extensions communicated between agents carry attestations $(\texttt{Xs:=[Y|Ys]})_{M,p,q}$. The signature by agent p provides cryptographic proof of authorship that p cannot deny.
(4) **Acyclicity:** Proposition **??** guarantees no circular terms. The occurs check prevents any writer from being bound to a term containing its dual reader, ensuring strict temporal ordering of stream elements.

Cooperative Extension. These properties establish that authenticated GLP streams provide blockchain security guarantees through logical foundations rather than proof-of-work or proof-of-stake mechanisms. Traditional blockchains employ competitive consensus where multiple parties race to extend the chain [15]. GLP's single-writer constraint makes competitive extension impossible—only the agent holding the tail writer can extend a stream. This enables cooperative protocols through explicit handover (Program E.4 in Appendix E), supporting round-robin production or priority-based scheduling without consensus overhead.

Interlaced Streams are a Blocklace. When multiple agents maintain interlaced streams that reference each other (Program E.8), they form a blocklace [1]—a DAG where blocks reference multiple predecessors—employed by modern consensus protocols including Cordial Miners [22], Morpheus [27], and Constitutional Consensus [21]. The resulting structure provides eventual consistency equivalent to Byzantine fault-tolerant CRDTs [55] while maintaining blockchain integrity guarantees.

In secure multiagent GLP, mutual attestations ensure all participants execute verified code, allowing consensus protocols to handle only network and fail-stop failures rather than Byzantine behaviour, significantly reducing complexity while maintaining safety.

G Implementation-Ready GLP (irGLP)

This appendix specifies irGLP, an implementation-ready transition system for single-agent (workstation) GLP execution. While the GLP transition system (Definition 2.7) is nondeterministic in goal selection and abstracts away suspension and failure, irGLP provides:

(1) Deterministic FIFO scheduling of active goals
(2) Explicit tracking of suspended goals with their suspension sets
(3) Explicit tracking of failed goals
(4) Automatic reactivation of suspended goals when blocking readers are instantiated

The key simplification compared to multiagent GLP is that all variables are local, so reader assignments can be applied immediately rather than through asynchronous communication.

Definition G.1 (irGLP Configuration)**.** An **irGLP configuration** over program P is a triple (Q, S, F) where:

- $Q \in \mathcal{A}_?^*$ is a sequence (queue) of **active goals**
- $S \subseteq \mathcal{A}_? \times 2^{\mathcal{V}?}$ contains **suspended goals**, each paired with its suspension set of blocking readers
- $F \subseteq \mathcal{A}_?$ contains **failed goals**

Definition G.2 (Reactivation Set)**.** Given a suspended goal set S and a readers substitution $\sigma?$, the **reactivation set** is:

$$\text{reactivate}(S, \sigma?) = \{G : (G, W) \in S \wedge \exists X? \in W.\ X?\sigma? \neq X?\}$$

These are goals suspended on readers that have been instantiated by $\sigma?$.

Definition G.3 (irGLP Transition System)**.** The transition system $\text{irGLP}(P) = (\mathcal{C}, c_0, \mathcal{T})$ over GLP program P and initial goal G_0 satisfying SO is defined by:

(1) $\mathcal{C}$ is the set of all irGLP configurations over P
(2) $c_0 = (G_0, \emptyset, \emptyset)$
(3) $\mathcal{T}$ is the set of all transitions $(Q, S, F) \to (Q', S', F')$ where $Q = A \cdot Q_r$ (head goal A, remainder Q_r) and exactly one of the following holds:
   (a) **Reduce:** There exists a first clause $C \in P$ for which the GLP reduction of A with C (Definition **??**) succeeds with $(B, \hat{\sigma})$. Let $R = \text{reactivate}(S, \hat{\sigma}?)$. Then:

$$Q' = (Q_r \cdot B \cdot R)\hat{\sigma}\hat{\sigma}?, \quad S' = S \setminus \{(G, W) : G \in R\}, \quad F' = F$$

   (b) **Suspend:** No clause succeeds, but the GLP reduction of A with at least one clause $C \in P$ suspends. Let $W = \bigcup_{C \in P} W_C$ where W_C is the suspension set for clause C. Then:

$$Q' = Q_r, \quad S' = S \cup \{(A, W)\}, \quad F' = F$$

   (c) **Fail:** No clause succeeds and no clause suspends (all clauses fail outright). Then:

$$Q' = Q_r, \quad S' = S, \quad F' = F \cup \{A\}$$

Remark G.4 (Deterministic Scheduling). The irGLP transition system is deterministic: given any non-terminal configuration, exactly one transition is enabled. Goal selection follows FIFO order from the active queue Q, and clause selection uses the first applicable clause as in GLP. The only source of nondeterminism in GLP—and-nondeterminism in goal selection—is resolved by the queue discipline.

Remark G.5 (Immediate Reader Application). In irGLP, reader substitutions $\hat{\sigma}?$ are applied immediately during reduction rather than through separate Communicate transitions as in the GLP transition system. This is sound for single-agent execution where all variables are local. The substitution $\hat{\sigma}\hat{\sigma}?$ in the Reduce transition applies both the writer assignments (to clause body variables) and their reader counterparts (to goals containing the corresponding readers).

Proposition G.6 (irGLP Implements GLP)**.** *For any irGLP run, projecting away the S and F components and interleaving Communicate transitions yields a valid GLP run. Conversely, any GLP run with FIFO goal selection and first-clause selection corresponds to an irGLP run.*

H Smartphone Implementation-ready Multiagent Transition System for GLP

This section combines the implementation-ready structure of irGLP (Section G) with the multiagent framework of maGLP (Section **??**). While irGLP provides deterministic scheduling and suspension management for single agents, and maGLP defines cross-agent communication through shared variables, irmaGLP specifies the concrete data structures and message-passing mechanisms suitable for multiagent smartphone implementation.

A variable X is *local* to agent p if X occurs in p's resolvent. Non-local variables require coordination through variable tables and explicit message passing, replacing maGLP's abstract shared-variable communication with concrete routing mechanisms.

The fundamental invariant: assignments produced by Reduce transactions are immediately applied if the reader is local, otherwise they become messages routed through the variable tables.

H.1 maGLP: The Specification

For completeness, we recall the definition of multiagent GLP from Section **??**, which irmaGLP implements.

Definition H.1 (Multiagent GLP)**.** The **maGLP transition system** over agents $P \subset \Pi$ and GLP module M is the multiagent transition system over multiagent asynchronous resolvents over M induced by the following transactions $c \to c'$:

(1) **Reduce** p**:** $c_p \to c'_p$ is a GLP Reduce transition, $\forall p \in P$
(2) **Communicate** p **to** q**:** $c_p = (G_p, \sigma_p)$, $c_q = (G_q, \sigma_q)$, $\{X? := T\} \in \sigma_p$, $X?$ occurs in G_q, $c'_p = (G_p, \sigma_p \setminus \{X? := T\})$, and $c'_q = (G_q\{X? := T\}, \sigma_q)$, $\forall p, q \in P$ (including $p = q$)
(3) **Network** p **to** q**:** The network output stream in c_p has a new message $\mathtt{msg}(q, X)$, 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 Reduce is unary while Communicate and Network are binary. Communicate operates between agents sharing logic variables, while Network operates through network streams. The implementation-ready variant below makes explicit the data structures and message routing needed to realize these abstract transactions on smartphones.

H.2 irmaGLP: The Implementation

Definition H.2 (Implementation-Ready maGLP Transition System)**.** The implementation-ready maGLP transition system over agents $P \subset \Pi$ and GLP module M is the multiagent transition system IRmaGLP $= (C, c_0, T)$ where:

- C is the set of all configurations where for each $p \in P$, the local state c_p is an implementation-ready resolvent as in Definition H.3
- c_0 is the initial configuration where for each $p \in P$:
  - $R_p = ([\mathtt{agent}(p, \mathtt{ch}(?,), \mathtt{ch}(?,))], \emptyset, \emptyset)$
  - $V_p = \emptyset$
  - $M_p = \emptyset$
- T is the union of all transitions generated by:
  - Unary Reduce transactions for each $p \in P$ (Definition H.9)
  - Binary Communicate transactions for each $(p, q) \in P \times P, p \neq q$ (Definition H.10)
  - Binary Network transactions for each $(p, q) \in P \times P, p \neq q$ (Definition H.11)

H.3 Local States

Definition H.3 (Implementation-Ready maGLP Local State)**.** The local state of agent $p \in \Pi$ is an **implementation-ready resolvent** $s_p = (R_p, V_p, M_p)$ where:

(1) $R_p = (A_p, S_p, F_p)$ separates the resolvent goals into three types:
   - **Active:** $A_p \in \mathcal{A}^*$
   - **Suspended:** $S_p \subseteq \mathcal{A} \times 2^{V?}$
   - **Failed:** $F_p \subseteq \mathcal{A}$
(2) $V_p \subseteq \mathcal{V} \times \Pi \times (\mathcal{T} \cup \Pi \cup \{\bot\})$ maintains shared variable state as a set of triples where each $(Y, q, s) \in V_p$:
   - **Created Writer:** $Y \in V$, $q = p$, $s \in \mathcal{T}$ is the value of Y, else $s = \bot$
   - **Imported Writer:** $Y \in V$, $q \neq p$, $s \in \mathcal{T}$ is the value of Y, else $s = \bot$
   - **Created Reader:** $Y \in V?$, $q = p$, $s \in \Pi$ is the read-requesting agent, else $s = \bot$
   - **Imported Reader:** $Y \in V?$ (reader), $q \neq p$, $s = q$ indicates a read request has been sent from p to q, else $s = \bot$
(3) M_p is a set of pending messages as pairs (content, destination) where destination $q \in \Pi$:
   - assignments $(X? := T, q)$
   - read requests $(request(X?, p), q)$ where p requests $X?$ from q
   - abandonment notifications $(abandon(X), q)$

The resolvent R_p partitions goals into three categories. Active goals A_p contains a queue of goals to be reduced in FIFO order. Suspended goals S_p pairs each atom with the set of readers preventing its reduction—for $(A, W) \in S_p$, the set W contains all readers from the suspension sets across all clause attempts. When any reader $X? \in W$ receives a value or is abandoned, A moves to the tail of A_p. Failed goals F_p contains atoms for which every reduction attempt either failed outright or suspended only on abandoned variables.

The variable table V_p maintains shared variables where one element of each reader/writer pair is local to p while its counterpart is non-local. For writers (both created and imported), the table stores the creator and any assignment to enable response to read requests. For created readers, it records which agent has requested the value. For imported readers, it tracks whether a read request has been sent to the creator. This unified structure ensures variables referenced by non-local counterparts are not prematurely garbage collected and provides routing information for cross-agent communication.

The variable table V_p maintains an invariant: it contains exactly those variables whose paired counterparts are non-local. When p receives a term containing a variable from V_p, that variable becomes local and must be removed from V_p. When p exports a term, the export helper function updates V_p accordingly: variables created by p are added when first exported, while variables created by others are removed (except for requested readers which require relay variables).

Helper Routines for Implementation-Ready Transactions, agent p.

The abandon helper notifies other agents when variable Y becomes unreachable. For imported variables, it notifies the creator q. For created readers with a requester s, it notifies that requester. The paired variable Y' is sent in the message to indicate which part of the pair was abandoned.

Definition H.4 (routine abandon(Y))**.** • If $(Y, q, s) \in V_p$ where $q \neq p$: remove from V'_p and add $(abandon(Y'), q)$ to M'_p
- If $(Y, p, s) \in V_p$ and $s \neq \bot$: remove from V'_p and add $(abandon(Y'), s)$ to M'_p
- Otherwise: just remove $(Y, \cdot, \cdot)$ from V'_p if present

where $Y' = Y?$ if $Y \in V$, else $Y' = Y$ if $Y \in V?$ (the paired variable)

The request helper sends a read request for an imported reader that hasn't been requested yet. It updates the table entry from $(X?, q, \bot)$ to $(X?, q, q)$ to record that the request was sent, preventing duplicate requests.

Definition H.5 (routine request(X?))**.** If $(X?, q, \bot) \in V'_p$ and $q \neq p$ then:

- Update to $(X?, q, q)$ in V'_p
- Add $(request(X?, p), q)$ to M'_p

The export helper updates the variable table when term T is sent outside agent p. Variables created by p are added to V_p when first exported. Imported variables are typically removed since they're no longer local, except for requested readers which require special handling: a fresh relay pair $(Z, Z?)$ is created with a forwarding goal to maintain the request relationship while allowing the original reader to leave p's scope.

Definition H.6 (routine export(T) returns T')**.** Set $T' := T$

- For each variable Y occurring in T:
  - **Local:** If Y created by p and $(Y, p, \cdot) \notin V'_p$: add $(Y, p, \bot)$ to V'_p
  - **Non-local:** If Y created by $q \neq p$ then
    - **Writer or Non-requested Reader:** If $Y \in V$ or $(Y, q, \bot) \in V'_p$ then remove $(Y, q, \cdot)$ from V'_p
    - **Requested Reader:** If $(Y, q, q) \in V'_p$ then create fresh pair $(Z, Z?)$, replace Y with $Z?$ in T', add export_reader(Y, Z) to A'_p, add $(Z?, p, \bot)$ to V'_p

T' is the result of applying variable replacements (if any) to T.

The export_reader goal spawned by the export helper is a simple forwarding process that waits for the original reader to receive its value and then assigns that value to the relay writer:

```
export_reader(Y?, Z) :- Z = Y?.
```

This ensures that when the creator eventually binds the original variable, the value flows through the relay to the new recipient.

Definition H.7 (routine reactivate(X?) for agent p returns R)**.** • Let $R = \{G : (G, W) \in S'_p, X? \in W\}$
- $S'_p := S'_p \setminus \{(G, W) : G \in R\}$
- Return R

H.4 Transactions

Next, we describe the implementation-ready maGLP transactions one by one:

Abandoned variables. During goal reduction, variables may become abandoned when their paired counterparts disappear from the computation without being instantiated. This happens when a variable that occurs in the reduced atom is neither instantiated by the reduction nor occurring in the resulting body. The implementation should detect such abandonment to prevent indefinite suspension or shared-variable entries for variables that can never receive values. Abandoned variables allow garbage-collection in shared variable tables and cause dependent suspended goals to fail rather than wait indefinitely.

Definition H.8 (Variable Abandonment in Reduction)**.** When reducing atom A with clause C yielding body B and substitution $\hat{\sigma}$, a variable Y is *abandoned* if its paired variable Y' satisfies all three conditions: Y' occurs in A, Y' is not instantiated by $\hat{\sigma}$ or $\hat{\sigma}?$, and Y' does not occur in B.

Definition H.9 (Implementation-Ready Reduce Transaction)**.** The unary Reduce transaction for agent p transitions $(R_p, V_p, M_p) \to (R'_p, V'_p, M'_p)$ where $R_p = (A_p, S_p, F_p)$, $(R'_p, V'_p, M'_p) := (R_p, V_p, M_p)$ with $A_p = A \cdot A_r$ for head goal A:

(1) **Reduce:** If GLP reduction of A with first applicable clause $C \in M$ succeeds with $(B, \hat{\sigma})$:
- Let $R = \bigcup_{X? \in V_{\hat{\sigma}?}}$ reactivate$(X?)$ (modifies S'_p)
- $A'_p := (A_r \cdot B \cdot R)\hat{\sigma}\hat{\sigma}?$
- Update M'_p for created readers: add $(X? := T, r)$ for each $\{X? := T\} \in \hat{\sigma}?$ where $(X?, p, r) \in V'_p, r \neq \bot$
- Update M'_p for imported writers: add $(X? := T, q)$ for each $\{X := T\} \in \hat{\sigma}$ where $(X, q, \cdot) \in V'_p, q \neq p$
- Call abandon(Y) for each abandoned variable Y

(2) **Suspend:** Else if $W = \bigcup_{C \in M} W_C \neq \emptyset$:
- $A'_p := A_r$
- $S'_p := S'_p \cup \{(A, W)\}$
- Call request$(X?)$ for each $X? \in W$ (modifies V'_p and M'_p)

(3) **Fail:** Else:
- $A'_p := A_r$
- $F'_p := F'_p \cup \{A\}$
- Call abandon(Y) for each variable Y in A (modifies V'_p and M'_p)

Then $R'_p := (A'_p, S'_p, F'_p)$.

Definition H.10 (Implementation-Ready Communicate Transaction)**.** The binary Communicate transaction $(c_p, c_q) \to (c'_p, c'_q)$ where $p \neq q$ and $(m, q) \in M_p$. Set $(c'_p, c'_q) := (c_p, c_q)$, remove (m, q) from M'_p, and case:

(1) **Assignment** $m = (X? := T)$:
- If $(X?, q, r) \in V'_q$ where $r \in \Pi$ (created reader with pending request):
  - Forward assignment: add $(X? := T, r)$ to M'_q
  - Update entry: $(X?, q, r) \to (X?, q, T)$ in V'_q
- Else if $(X?, q, \bot) \in V'_q$ (created reader, no request yet):

  - Store value: update to $(X?, q, T)$ in V'_q
- Else if $(X?, r, s) \in V'_q$ where $r \neq q$ (imported reader):
  - Let $R = \text{reactivate}(X?)$ for agent q (modifies S'_q)
  - If $T \neq \bot$: $A'_q := (A_q \cdot R)\{X? := T\}$, and apply $\{X? := T\}$ to S'_q and F'_q
  - Else: $A'_q := A_q \cdot R$
  - Remove $(X?, r, s)$ from V'_q
  - For each variable Y in T not already local to q and created by r': add $(Y, r', \bot)$ to V'_q

(2) **Read Request** $m = \text{request}(X?, p)$:

- If $p = \bot$ then call $\text{abandon}(X?)$ for agent q (modifies V'_q and M'_q)
- Else if $(X?, q, T) \in V'_q$ where $T \in \mathcal{T}$ (value already stored): add $(X? := T, p)$ to M'_q
- Else if $(X?, q, \bot) \in V'_q$: update to $(X?, q, p)$ in V'_q
- Else if $(X, q, T) \in V'_q$ then add $(X? := T, p)$ to M'_q

Definition H.11 (Implementation-Ready Network Transaction)**.** The binary Network transaction $(c_p, c_q) \to (c'_p, c'_q)$ where $p \neq q$ and a new $\texttt{msg}(q, X)$ appears in p's network output stream. Set $(c'_p, c'_q) := (c_p, c_q)$:

- Let $X' := \text{export}(X)$ for agent p (modifies V'_p and M'_p)
- Add X' to q's network input stream
- For each variable Y in X' not already local to q and created by r: add $(Y, r, \bot)$ to V'_q

The scheduler operates deterministically by selecting the head of the active queue A_p. When any reader $X? \in W$ for a suspended goal $(A, W) \in S_p$ receives a value or is marked abandoned, the goal A is moved from S_p to A_p for re-evaluation. Goals in F_p remain terminal, preserving logical completeness while enabling runtime fault analysis.

H.5 Example Scenarios

The following scenarios illustrate how the variable table and message routing mechanisms work together to support cross-agent communication, including the friend-mediated introduction protocol.

H.5.1 Scenario 1: Direct Communication. Alice sends a message to her friend Bob through an established channel.

Initial State. Alice holds writer W for Bob's input channel; Bob holds paired reader $W?$.

- $V_{alice} = \{(W, alice, \bot)\}$ — created writer, reader is non-local
- $V_{bob} = \{(W?, alice, \bot)\}$ — imported reader from alice

Step 1: Bob suspends on $W?$**.** Bob's goal needs the value of $W?$, causing suspension.

- Bob calls $\text{request}(W?)$: updates $(W?, alice, \bot) \to (W?, alice, alice)$
- Bob sends $(\text{request}(W?, bob), alice)$ to Alice

Step 2: Alice receives request. Alice processes the read request.

- Alice updates $(W, alice, \bot) \to (W, alice, bob)$ — records Bob as requester

Step 3: Alice binds W**.** Alice's reduction binds $W := T$ for some term T.

- Alice sees $(W, alice, bob)$ with requester $bob \neq \bot$
- Alice sends $(W? := T, bob)$ to Bob

Step 4: Bob receives assignment. Bob processes the assignment for imported reader $W?$.

- Bob has $(W?, alice, alice)$ — imported reader (creator ≠ bob)
- Bob applies $\{W? := T\}$ to resolvent
- Bob removes $(W?, \cdot, \cdot)$ from V_{bob}
- Suspended goals reactivate

H.5.2 Scenario 2: Friend-Mediated Introduction. Bob introduces Alice to Charlie by creating a shared channel and distributing its endpoints. This scenario demonstrates imported writers and the creator-as-routing-hub pattern.

Initial State. Bob knows both Alice and Charlie; they don't know each other.

- $V_{alice} = V_{charlie} = V_{bob} = \emptyset$ (for the new channel)

Step 1: Bob creates channel. Bob executes `new_channel(ch(AC?, CA), ch(CA?, AC))` creating:

- Writer CA and reader $CA?$ (Alice → Charlie direction)
- Writer AC and reader $AC?$ (Charlie → Alice direction)

All four variables are initially local to Bob.

Step 2: Bob exports to Alice. Bob sends `ch(AC?, CA)` to Alice via export.

- Bob adds $(AC?, bob, \bot)$ and $(CA, bob, \bot)$ to V_{bob}
- Alice imports: adds $(AC?, bob, \bot)$ and $(CA, bob, \bot)$ to V_{alice}

Alice now holds: reader $AC?$ (to receive from Charlie) and writer CA (to send to Charlie).

Step 3: Bob exports to Charlie. Bob sends `ch(CA?, AC)` to Charlie via export.

- Bob adds $(CA?, bob, \bot)$ and $(AC, bob, \bot)$ to V_{bob}
- Charlie imports: adds $(CA?, bob, \bot)$ and $(AC, bob, \bot)$ to $V_{charlie}$

Charlie now holds: reader $CA?$ (to receive from Alice) and writer AC (to send to Alice).

Variable Table Summary After Introduction.

- $V_{bob} = \{(AC?, bob, \bot), (CA, bob, \bot), (CA?, bob, \bot), (AC, bob, \bot)\}$
- $V_{alice} = \{(AC?, bob, \bot), (CA, bob, \bot)\}$ — both imported from bob
- $V_{charlie} = \{(CA?, bob, \bot), (AC, bob, \bot)\}$ — both imported from bob

Bob is the creator of all channel variables and serves as the routing hub.

Step 4: Charlie requests $CA?$**.** Charlie's goal needs the value of $CA?$.

- Charlie calls $\text{request}(CA?)$: sends $(\text{request}(CA?, charlie), bob)$
- Charlie updates $(CA?, bob, \bot) \to (CA?, bob, bob)$

Step 5: Bob receives request. Bob processes Charlie's read request for $CA?$.

- Bob has $(CA?, bob, \bot)$ — created reader, no value yet

- Bob updates $(CA?, bob, \bot) \rightarrow (CA?, bob, charlie)$ — records Charlie as requester

Step 6: Alice binds CA. Alice sends a message by binding writer $CA := T$.

- Alice has $(CA, bob, \bot)$ — imported writer from bob
- Since $bob \neq alice$, Alice sends $(CA? := T, bob)$ to the creator

Step 7: Bob receives assignment, forwards to Charlie. Bob processes assignment for $CA?$.

- Bob has $(CA?, bob, charlie)$ — created reader with pending request
- Bob forwards: sends $(CA? := T, charlie)$
- Bob updates $(CA?, bob, charlie) \rightarrow (CA?, bob, T)$ — stores value

Step 8: Charlie receives value. Charlie processes assignment for imported reader $CA?$.

- Charlie has $(CA?, bob, bob)$ — imported reader (creator ≠ charlie)
- Charlie applies $\{CA? := T\}$ to resolvent
- Charlie removes $(CA?, \cdot, \cdot)$ from $V_{charlie}$
- Suspended goals reactivate

Key Insight: Creator as Routing Hub. The creator (Bob) maintains entries for all channel variables and routes values from writers to readers:

(1) Alice binds imported writer → notifies creator Bob
(2) Bob receives value, checks for pending request → forwards to Charlie
(3) Charlie receives value at imported reader

This pattern enables introduction without requiring Alice and Charlie to communicate directly until after the introduction completes.

H.5.3 Scenario 3: Request Arrives After Value. If Alice binds CA before Charlie requests $CA?$:

Step 6': Alice binds CA early.

- Alice sends $(CA? := T, bob)$ to creator

Step 7': Bob receives assignment, no request yet.

- Bob has $(CA?, bob, \bot)$ — no requester
- Bob stores value: updates $(CA?, bob, \bot) \rightarrow (CA?, bob, T)$

Step 4': Charlie requests $CA?$ later.

- Charlie sends $(\text{request}(CA?, charlie), bob)$

Step 5': Bob receives request, value already stored.

- Bob has $(CA?, bob, T)$ where $T \in \mathcal{T}$
- Bob immediately replies: sends $(CA? := T, charlie)$

The order of assignment and request does not matter; the creator stores whichever arrives first and completes the exchange when both are present.

H.6 Extensions for Secure Multiagent GLP

To extend the implementation-ready transition system to Secure maGLP, the following cryptographic mechanisms augment the definitions without modifying their structure:

H.6.1 Agent Identity and Cryptography. Each agent $p \in \Pi$ is augmented with:

- A self-chosen keypair (pk_p, sk_p) where the public key pk_p serves as the agent's identity
- The agent identifier p is synonymous with pk_p throughout the system
- We assume knowledge of other agents' public keys through social contacts

H.6.2 Message Authentication and Encryption. All messages in M_p are cryptographically protected. A message $(m, q) \in M_p$ becomes $(m_{M,p,q}, q)$ where the subscript notation indicates:

- M: Attestation by the GLP runtime proving m resulted from correct execution of module M
- p: Digital signature using agent p's private key sk_p
- q: Encryption using agent q's public key pk_q

H.6.3 Transaction Augmentations.

Reduce Transaction. When generating messages $(X? := T, r)$ for remote readers, the implementation creates $(X? := T)_{M,p,r}$ with attestation proving the assignment resulted from correct goal/clause reduction using module M.

Communicate Transaction. Before processing any received message $(m_{M,p,q}, q)$:

(1) Decrypt using q's private key sk_q
(2) Verify signature using p's public key pk_p
(3) Validate attestation for module M
(4) Discard the message if any verification fails
(5) Process according to Definition H.10 only if all verifications succeed

Network Transaction. Network messages $\texttt{msg}(q, X)$ are similarly protected as $(\texttt{msg}(q, X))_{M,p,q}$ ensuring authenticated channel establishment.

H.6.4 Module Verification.

- Each agent executes a verified GLP module M with a cryptographic hash identifier
- Attestations include the module hash, enabling recipients to verify code compatibility
- Guard predicates `attestation(X, att(Agent, Module))` and `module(M)` provide program-level access to verification results

H.6.5 Security Properties Achieved. These extensions ensure:

- **Integrity**: Messages cannot be modified without detection
- **Confidentiality**: Only intended recipients can decrypt messages
- **Non-repudiation**: Senders cannot deny authenticated messages
- **Authentication**: All inter-agent communication is mutually authenticated

The implementation-ready transition system with these cryptographic extensions realizes Secure maGLP while maintaining the same operational behaviour for correctly authenticated participants. Byzantine agents who fail verification are effectively excluded from the computation through message rejection.