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

Ehud Shapiro

London School of Economics and Weizmann Institute of Science

Abstract. Grassroots platforms are distributed applications run by cryptographically-identified people on their networked personal devices, where multiple disjoint platform instances emerge independently and coalesce when they interoperate. Their foundation is the grassroots social graph, upon which grassroots social networks, grassroots cryptocurrencies, and grassroots democratic federations can be built.
Grassroots platforms have yet to be implemented, the key challenge being faulty and malicious participants: without secure programming support, correct participants cannot reliably identify each other, establish secure communication, or verify each other's code integrity.
We present Grassroots Logic Programs (GLP), a secure, multiagent, concurrent, logic programming language for implementing grassroots platforms. GLP extends logic programs with paired single-reader/single-writer (SRSW) logic variables, providing secure communication channels among cryptographically-identified people through encrypted, signed and attested messages, which enable identity and code integrity verification. We present GLP progressively: logic programs, concurrent GLP, multiagent GLP, augmenting it with cryptographic security, and providing smartphone implementation-ready specifications. We prove safety properties including that GLP computations are deductions, SRSW preservation, acyclicity, and monotonicity. We prove multiagent GLP is grassroots and that GLP streams achieve blockchain security properties. We present a grassroots social graph protocol establishing authenticated peer-to-peer connections and demonstrate secure grassroots social networking applications.

1 Introduction

Grassroots platforms. Grassroots platforms [62] are distributed applications in which multiple disjoint platform instances emerge independently and coalesce when they interoperate. They are run by people on their networked personal devices (today—smartphones), who are identified cryptographically [49], communicate only with authenticated friends, and can participate in multiple instances of multiple grassroots platforms simultaneously. The grassroots social graph [63] is both a platform in its own right and the infrastructure layer for all other grassroots platforms. In it, nodes represent people, edges—authenticated friendships, and connected components arise spontaneously and interconnect through

befriending. The social graph provides grassroots platforms with communication along graph edges, encrypted for ensuring privacy, signed for authenticity and attested for integrity. Upon this foundation, grassroots social networks [63], grassroots cryptocurrencies [64], and grassroots democratic federations [68] are built.

Programming grassroots platforms. A key challenge in implementing grassroots platforms is overcoming faulty and malicious participants [32]. Without secure language support, correct participants cannot reliably identify each other, establish secure communication channels, or verify each other's code integrity [52,13]. While grassroots platforms have been formally specified and their properties proven [62,63,64,68,65], they are so far mathematical constructions without an actual implementation. 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. Grassroots Logic Programs aim to close the gap between the mathematical specifications and actual implementation of grassroots platforms.

A grassroots programming language. Our goal is to design a high-level, secure, multiagent, concurrent programming language suitable for the implementation of grassroots platforms. To do so, the language should address:

1. Mutual authentication [8] enabling people to identify each other and verify each other's code identity and integrity
2. Grassroots social graph formation through both cold calls (for bootstrap and connecting disconnected components) and friend-mediated introductions
3. Secure communication among friends
4. Multiagent operational semantics [61] with proven security, safety and liveness
5. Useful abstractions for distributed multiagent programming in general, and metaprogramming support in particular, to enable the development of programming tools and runtime support for the language within the language.

Grassroots Logic Programs. We present Grassroots Logic Programs (GLP), a secure, multiagent, concurrent, logic programming language designed for implementing grassroots platforms. GLP extends logic programs [37,71] with paired single-reader/single-writer variables (akin to futures and promises [16,5]), 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.

Through signed attestations at the language level, participants verify each other's identity and code integrity when befriending and communicating. These mechanisms enable both cold calls (for bootstrap and connecting disconnected components) and friend-mediated introductions (the preferred trust propagation method).

We present Grassroots Logic Programs and prove their properties in five steps, injecting illustrative programming examples along the way:

1. **Logic Programs:** Define a transition system-based operational semantics for logic programs (LP) [37,71], in which a conjunctive goal (resolvent) is transformed by nondeterministic goal/clause reductions.
2. **Concurrent GLP:** Extend LP with reader/writer pairs, which must satisfy the Single-Reader/Single-Writer requirement; extend unification to suspend upon an attempt to bind a reader; extend configurations to include pending assignments to readers; extend transitions to include the application of an assignment from a writer to its paired reader, and thus provide nondeterministic interleaving-based asynchronous operational semantics for concurrent GLP. Prove safety properties [2], including that GLP computations are deductions [31,37]. Provide GLP with deterministic 'workstation implementation-ready' transition system (Appendix G), based on which a workstation implementation of GLP can be developed to support GLP program development.
3. **Multiagent, Concurrent GLP:** Employ multiagent transition systems [61] with atomic transactions [65] to define the operational semantics of multiagent concurrent GLP, in which goal reductions are local and assignments of shared logic variables are realized as writer-to-reader messages among agents, and prove it to be grassroots [62].
4. **Secure, Multiagent, Concurrent GLP:** Augment agents with self-chosen keypairs and augment cross-agent communication that is encrypted, signed and attested, resulting in secure, multiagent, concurrent GLP. Prove its security as a distributed systems [14] and that GLP streams enjoy the security properties of blockchains [44].
5. **Implementation-Ready Specification:** Replace nondeterministic goal selection with deterministic scheduling, and replace abstract push-based shared-variable communication with pull-based message-passing using dynamic shared-variable tables, geared for smartphone deployment.

The remainder of this paper is organized as follows. Section 2 recalls logic programs. Section 3 extends them to concurrent GLP. Section 4 presents basic GLP programming techniques. Section 5 defines multiagent GLP and proves it grassroots. Section 6 implements the grassroots social graph cold-call and friend-mediated introduction protocols. Section 7 adds cryptographic security and attestations and presents security properties, including blockchain security properties of streams. Section 8 discusses smartphone implementation. Section 9 reviews related work, and Section 10 concludes. The appendixes provide A LP syntax, B proofs, C social-graph protocol properties, D grassroots social networking, E guard and system predicates, F additional programming and metaprogramming examples, and G single-workstation and H networked-smartphones implementation-ready specifications of GLP.

2 Logic Programs

Here we introduce transition systems, providing the formal framework for the operational semantics of both Logic Programs and Grassroots Logic Programs.

We recall standard Logic Programs (LP): syntax, most-general unifier (mgu), operational semantics via nondeterministic goal/clause reduction, compositional semantics, and a proof that LP computations are deductions.

2.1 Transition Systems

We use $\subset$ to denote the strict subset relation, $\subseteq$ when equality is also possible, and $a \neq b \in S$ as a shorthand for $a \neq b \wedge a \in S \wedge b \in S$. The following definition uses 'configuration' rather than the more standard 'state' to avoid confusion with the 'local state' of agents in a multiagent transition system, Definition 12.

Definition 1 (Transition System). *A **transition system** is a tuple $TS = (C, c0, T)$ where:*

- *C is an arbitrary set of **configurations***
- *$c0 \in C$ is a designated **initial configuration***
- *$T \subseteq C \times C$ is a **transition relation**. A transition $(c, c') \in T$ is also written as $c \rightarrow c' \in T$.*

*A transition $c \rightarrow c' \in T$ is **enabled** from configuration c. A configuration c is **terminal** if no transitions are enabled from c. A **computation** is a (finite or infinite) sequence of configurations where for each two consecutive configurations (c, c') in the sequence, $c \rightarrow c' \in T$. A **run** is a computation starting from $c0$, which is **complete** if it is infinite or ends in a terminal configuration.*

2.2 Logic Programs Syntax

The syntax of Logic Programs follows the standard in logic programming and Prolog, and is formally defined in Appendix **??**. We note that V denotes the set of all variables and $\mathcal{T}$ the set of all terms. We recall the quintessential logic program for list concatenation as an example:

Example 1 (Append).

```
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
append([], Ys, Ys).
```

Logically, a logic program clause `A:- B` is a universally-quantified implication in which B implies A, and a program is a conjunction of its clauses. By convention, we use plural variable names like `Xs` to denote a list of `X`'s,

2.3 Logic Programs Operational Semantics

Definition 2 (Substitution, Instance, Unifier, Most-General Unifier). *A **substitution** σ is an idempotent function $\sigma : V \rightarrow \mathcal{T}$, namely a mapping from variables to terms applied to a fixed point. By convention, $\sigma(x) = x\sigma$.*

- *Given a substitution σ, $V_\sigma := \{X \in V \mid X\sigma \neq X\}$.*

- *Given a term $T \in \mathcal{T}$ and a substitution σ, $T\sigma$ is the term obtained from T by replacing every variable $X \in T$ by the term $X\sigma$.*
- *The partial order on terms $\preceq \subset \mathcal{T} \times \mathcal{T}$ is defined by $T \preceq T'$, or T' is an **instance** of T, if there is a substitution σ for which $T\sigma = T'$. If $T \neq T\sigma$ we say that σ **instantiates** T.*
- *For substitutions σ and σ', $\sigma \preceq \sigma'$ if for every $T \in \mathcal{T}$, $T\sigma \preceq T\sigma'$, σ is **as general as** σ' if $\sigma \preceq \sigma'$.*
- *A substitution σ is a **unifier** of two terms $T, T' \in \mathcal{T}$ if $T\sigma = T'\sigma$; it is a **most-general unifier (mgu)** of T, T' if in addition it is as general as any other unifier of T and T'.*

Namely, a most general unifier is frugal in not instantiating variables more than necessary.

Remark 1 (Substitution as Assignment Set). We view a substitution σ equivalently as a set of assignments $\{X_1 := T_1, \ldots, X_n := T_n\}$ where $X_i\sigma = T_i$ and $T_i = T_i\sigma = T$. Thus the singleton substitution mapping X to T is $\{X := T\}$, its application $T\sigma$ may be written $T\{X := T\}$, the empty substitution is $\emptyset$, and composition of commutative substitutions corresponds to set union.

Definition 3 (Renaming, Rename apart). *A **renaming** is a substitution $\sigma : V \mapsto V$ that maps variables to variables. A renaming σ renames T' **apart from** T if $T'\sigma$ and T have no variable in common.*

We assume a fixed renaming-apart function, so that the result of renaming T' apart from T is well defined. Next we define the operational semantics of Logic Programs via a transition system.

Definition 4 (LP Goal/Clause Reduction). *Given LP goal A and clause C, with H :- B being the result of renaming C apart from A, the **LP reduction** of A with C **succeeds with result** (B, σ) if A and H have an mgu σ, else **fails**.*

Definition 5 (Logic Programs Transition System). *A transition system $LP = (C, c0, T)$ is a **Logic Programs transition system** for a logic program M and initial goal $G_0 \in \mathcal{G}(M)$ if $C = \mathcal{G}(M)$, $c0 = G_0$, and T is the set of all transitions $G \to G' \in \mathcal{G}(M)^2$ such that for some atom $A \in G$ and clause $C \in M$ the LP reduction of A with C succeeds with result (B, σ), and $G' = (G \backslash \{A\} \cup B)\sigma$.*

We write $G \xrightarrow{\sigma} G'$ when we want to make the substitution of a reduction explicit. As a tribute to resolution theorem proving [50]—the intellectual ancestor of logic programming—a configuration of LP is also referred to as a *resolvent*.

Logic Programs have two forms of nondeterminism: the choice of $A \in G$, called *and-nondeterminism*, and then choice of $C \in M$, called *or-nondeterminism*. Thus, as an abstract model of computation, LP are closely-related to *Alternating Turing Machines*, a generalization of Nondeterministic Turing Machines [57].

Definition 6 (Proper Run and Outcome). *A run $\rho : G_0 \xrightarrow{\sigma_1} G_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} G_n$ of LP is **proper** if for any $1 \leq i < n$, a variable that occurs in G_{i+1} but not in G_i also does not occur in any G_j, $j < i$. If proper, the **outcome** of ρ is $(G_0$:- $G_n)\sigma$ where $\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$.*

It so happens that the set of all outcomes of all proper runs of a logic program constitutes its fully-abstract compositional semantics [20]. Next we prove the key safety property of LP:

Proposition 1 (LP Computation is Deduction). *The outcome* $(G_0 \text{ :- } G_n)\sigma$ *of a proper run* $\rho : G_0 \xrightarrow{\sigma_1} G_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} G_n$ *of LP, where* $\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$*, is a logical consequence of* M*.*

3 Grassroots Logic Programs

We present Grassroots Logic Programs (GLP) as an extension of Logic Programs: The syntax is extended with reader variables $X?$, where X and $X?$ form a reader/writer pair, and with the Single-Reader/Single-Writer syntactic restriction on clauses. For example, here is the quintessential concurrent logic program for merging two streams (incrementally constructed, potentially unbounded lists), written in GLP. Its first two arguments are the input streams to be merged, the third is the merged output stream:

Program 1: GLP Fair Stream Merger

```
merge([X|Xs],Ys,[X?|Zs?]) :- merge(Ys?,Xs?,Zs). % output from first stream
merge(Xs,[Y|Ys],[Y?|Zs?]) :- merge(Xs?,Ys?,Zs). % output from second stream
merge([],[],[]).                                 % terminate on empty streams
```

Note that in each clause, each reader or writer occurs at most once.

The operational semantics of GLP extends that of LP as follows:

1. **Synchronisation**: Unification may only instantiate writers, so in addition to succeed/fail, unification may suspend if it requires instantiating readers.
2. **Communication**: When a unifying writer substitution binds a writer X to a term T, the message $X? := T$ encoding its paired reader assignment is created and added to the configuration. Its application happens asynchronously, realizing a message T from the single occurrence of X to the single occurrence of $X?$.
3. **Deterministic clause selection**: The first applicable clause is chosen, not nondeterministically as in LP. This provides for the fairness of `merge` presented above: As long as the two input streams are available the output dovetails the two inputs, due to switching their order in the recursive call of the first clause; as long as only one stream is available, its elements are copied to the output; and when both streams are unavailable the goal suspends.

The remainder of this section presents GLP syntax, nondeterministic operational semantics, and safety properties. A deterministic 'workstation implementation-ready' transition-system specification for GLP is presented in Appendix G.

3.1 GLP Syntax

Reader/Writer pairs. GLP extends Logic Programs with paired reader/writer variables, where a *writer* X is a single-assignment variable (promise) and its

paired reader $X?$ provides read-only access to the (future) value of X. We denote by V the set of all writers, $V?$ the set of all readers and, $\mathcal{V} = V \cup V?$ the set of all variables, where for each writer $X \in V$ there exists a paired reader $X? \in V?$. We view ? as an identity suffix operator on non-writers, namely $(X?)? = X?$ for $X? \in V?$ and $T? = T$ for $T \notin \mathcal{V}$. We use $\mathcal{A}_?$ and $\mathcal{G}_?$ to denote the set of all atoms and goals, respectively, over $\mathcal{V}$ (i.e., goals that may contain both readers and writers), and for a GLP program M, $\mathcal{A}_?(M)$ and $\mathcal{G}_?(M)$ to denote the subsets of $\mathcal{A}_?$ and $\mathcal{G}_?$, respectively, restricted to the vocabulary of M.

Single-Reader/Single Writer (SRSW). The fundamental requirement in GLP is *single-writer*: any writer may occur at most once in any state of a computation, ensuring there can be no conflict when writing on a logic variable. We extend it to the *single-reader/single-writer (SRSW) requirement* that any reader also occurs at most once. The reason is that with multiple instances of a reader, instantiating the writer to a term containing another writer would give all instances of the paired reader access to that writer, violating the single-writer requirement. The SRSW requirement is realized by two complementary concepts:

1. *SRSW syntactic restriction on clauses*: In each clause each variable (reader or writer) occurs at most once
2. *SRSW invariant*: Given a resolvent that satisfies the SRSW requirement, applying to it a goal reduction with a clause that satisfies the SRSW syntactic restriction produces a new resolvent that also satisfies the SRSW requirement.

This SRSW syntactic restriction excludes programs like the equality definition $X = X$ as it has two occurrences of the writer X. At the same time it eliminates the need for distributed atomic unification [30]—replacing it with efficient point-to-point communication of a single assignment from the single occurrence of a writer to the single occurrence of its paired reader.

3.2 GLP Operational semantics

Definition 7 (Writer and Reader Substitution, Reader Counterpart, Suspension Set, Writer MGU).

A substitution σ is a ***writer substitution*** *if σ:*

1. *only binds writers: $V_\sigma \subset V$*
2. *does not bind writers to writers: if $X \neq X\sigma$ for $X \in V$ then $X\sigma \notin V$*
3. *does not form cycles through readers: $X?$ does not occur in $X\sigma$ for any $X \in V_\sigma$*

A substitution σ is a ***reader substitution*** *if $V_\sigma \subset V?$. If σ is a writer substitution then its* ***reader counterpart*** *is the reader substitution $\sigma?$ defined by $X?\sigma? = X\sigma$ for every $X \in V_\sigma$.*

The ***suspension set*** *of a (regular) substitution σ is $W_\sigma := \{X? \in V? : X?\sigma \notin \mathcal{V}\}$.*

The ***writer unification*** *of two terms:*

1. ***succeeds with*** *σ if they have a writer mgu σ.*
2. *else* ***suspends on*** *W_σ if they have a (regular) mgu σ*

3. else ***fails***

Remark 2. If a writer mgu exists it is unique, rather than unique up to renaming, since it does not include writer-to-writer assignments. If writer-to-writer assignments were allowed then, by the single-writer restriction, the assignment would leave their two paired readers *abandoned*, namely without a writer that can provide them with a value. The occurs check condition for the reader counterpart ensures that no writer is bound to a term containing its paired reader, preventing the formation of circular terms, as proven in Proposition 4.

Renaming (Definition 3) is extended to respect variable pairing:

Definition 8 (GLP Renaming). *Two GLP terms T, T' have a variable in common if for some writer $X \in V$, either X or $X?$ occur in T and either X or $X?$ occur in T'. A* ***GLP renaming*** *is a renaming substitution $\sigma : \mathcal{V} \mapsto \mathcal{V}$ such that for each $X \in V$: $X\sigma \in V$ and $X?\sigma = (X\sigma)?$.*

Definition 9 (GLP Goal/Clause Reduction). *Given GLP goal A and clause C, with H :- B being the result of the GLP renaming of C apart from A, the* ***GLP reduction*** *of A with C* ***succeeds with result*** *(B, σ),* ***suspends on*** *W, or* ***fails****, respectively, depending on the result of the writer unification of A and H.*

The GLP operational semantics is defined via the following transition system, which employs the notions defined above to extend LP (Definition 5). It abstracts-away goal suspension and failure; these are used in the implementation-ready specifications (Appendixes G and H) for explicit goal scheduling, suspension and activation.

Definition 10 (GLP Transition System). *Given a GLP program M, an* ***asynchronous resolvent*** *over M is a pair (G, σ) where $G \in \mathcal{G}_?(M)$ and σ is a reader substitution. A transition system $GLP = (\mathcal{C}, c0, \mathcal{T})$ is a* ***GLP transition system*** *over M and initial goal $G_0 \in \mathcal{G}_?(M)$ satisfying SRSW if:*

1. $\mathcal{C}$ is the set of all asynchronous resolvents over M
2. $c0 = (G_0, \emptyset)$
3. $\mathcal{T}$ is the set of all transitions $(G, \sigma) \rightarrow (G', \sigma')$ satisfying:
   (a) ***Reduce:*** *there exists an atom $A \in G$ such that $C \in M$ is the first clause for which the GLP reduction of A with C succeeds with result $(B, \hat{\sigma})$, $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$, and $\sigma' = \sigma \circ \hat{\sigma}$?*
   (b) ***Communicate:*** *$\hat{\sigma} = \{X? := T\} \in \sigma$, $G' = G\hat{\sigma}$, and $\sigma' = \sigma \setminus \hat{\sigma}$*

The monotonicity of GLP goal/clause reduction (Proposition 5) allows a simple *GLP fairness requirement*: A goal that can be reduced is eventually reduced.

Guards and system predicates. GLP also includes *guards*—predicates that test runtime conditions (e.g., `ground(X)` tests if `X` contains no variables) without modifying state, appearing after clause heads separated by `|`—and *system predicates* that provide access to the GLP runtime state and operating system and hardware capabilities (variable state and name, arithmetic evaluation,

timestamps). Guards enable conditional clause selection. The `ground(X)` guard allows relaxing the single-reader constraint for `X?` for the clause it occurs in, as having multiple occurrences of `X?` instantiated to a ground term does not violate the fundamental single-writer requirement. Their specification appears in Appendix E.

3.3 GLP Safety

Here we prove that, like LP, GLP computations are deductions, but, unlike LP, a goal that can be reduced in a configuration can still be reduced in any subsequent configuration of the computation.

GLP computations are deductions. First we show that the extensions of GLP over LP do not take it outside of the logic programming realm.

Definition 11 (Pure Logic Variant). *Given a GLP term or goal T, the **pure logic variant** $L(T)$ of T is defined by replacing every reader $X?$ in T with its paired writer X. Given a GLP computation r, its pure logic variant $L(r)$ is the result of replacing every configuration (G, σ) in r by $L(G)$, removing duplications and labelling the remaining transitions by the mgu of their respective reduction.*

Note that duplications as above result from Communicate transitions.

Proposition 2 (GLP Computations are Deductions). *For any finite GLP run r, let $L(r) = G0 \xrightarrow{\sigma_1} G_1 \xrightarrow{\sigma_2} \ldots G_n$, with $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_n$, then $(G_: - G_n)\sigma$ is a logical consequence of $L(M)$.*

Next, we establish essential safety properties for GLP that distinguish it from standard LP. The key is monotonicity—once a goal becomes reducible in GLP, it remains reducible.

SRSW.

Proposition 3 (SRSW Invariant). *If the initial goal G_0 in a GLP run satisfies SRSW, then every goal in the run satisfies SRSW.*

Acyclicity. The occurs check in readers prevents the formation of circular terms.

Proposition 4 (Acyclicity). *If the initial goal G_0 in a GLP run contains no circular terms, then no goal in the run contains a circular term.*

Monotonicity. Unlike LP where variable instantiation can cause a previously reducible goal to fail, GLP exhibits monotonicity. In a run, if a goal A can be reduced at some point, it remains reducible at all future points in that run, where "future" implies that readers in A (and only readers) have been further instantiated by other goal reductions.

Lemma 1 (Reader-Instance). *In any GLP run $G_0 \to G_1 \to \cdots$, if $G_i \to G_{i+1}$ via reduction with substitution σ? does not reduce $A \in G_i$, then $A\tau \in G_{i+1}$ where τ instantiates only readers.*

Proposition 5 (Monotonicity). *In any GLP run $G_0 \to G_1 \to \cdots$, if atom $A \in G_i$ can reduce with clause C, then for any $j > i$, either A has been reduced by step j, or there exists $A' \in G_j$ where $A' = A\tau$ for some reader substitution τ, and A' can reduce with C.*

4 Programming Examples

We present some basic GLP programming techniques through examples. Additional techniques appear in Appendix F.

Program 2: Concurrent Monitor

```
monitor(Reqs) :- monitor(Reqs?,0).

monitor([add(N)|Reqs],Sum) :-
    Sum1 := Sum? + N?, monitor(Reqs?,Sum1?).
monitor([subtract(N)|Reqs],Sum) :-
    Sum1 := Sum? - N?, monitor(Reqs?,Sum1?).
monitor([value(V)|Reqs],Sum) :-
    ground(Sum?) | V = Sum?, monitor(Reqs?,Sum?).
monitor([],_).
```

An example initial goal is:

```
client1(Xs), client2(Ys), merge(Xs?,Ys?,Zs), monitor(Zs?).
```

The monitor demonstrates a stateful service handling requests from multiple concurrent clients, serialized through stream merging (Program 3) whilst maintaining state through the `Sum` parameter in tail-recursive calls. The `value(V)` request demonstrates incomplete messages—upon receipt the monitor binds the response variable `V` to the current sum.

A fixed number of clients can be served by a fixed binary merge tree. A dynamically-changing set of clients can be served by the following dynamic stream merger, where an existing client can onboard a new client with a request stream `Ws` by sending down its own request stream the request `merge(Ws?)`, creating a dynamic merge tree as follows.

Program 3: Dynamic Stream Merger

```
merger(Ws,Xs,Out?) :- merge(Ws?,Xs?,Out).

merge([merge(Ws)|Xs],Ys,Zs?) :-
   merger(Ws?,Xs?,Xs1), merge(Xs1?,Ys?,Zs).
merge(Xs,[merge(Ws)|Ys],Zs?) :-
   merger(Ws?,Ys?,Ys1), merge(Xs?,Ys1?,Zs).
merge([X|Xs],Ys,[X?|Zs?]) :-
    X =\= merge(_) | merge(Ys?,Xs?,Zs).
merge(Xs,[Y|Ys],[Y?|Zs?]) :-
    Y =\= merge(_) | merge(Xs?,Ys?,Zs).
merge([],[],[]).
```

The resulting merge tree can be highly imbalanced; standard optimization techniques can be applied [66,67].

Broadcasting to multiple concurrent consumers uses the `ground` guard to enable input replication without violating the single-writer constraint:

Program 4: Concurrent Stream Distribution

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

When `X` is ground, multiple occurrences in the clause body do not violate SRSW. Additional programming examples appear in Appendix F.

5 Multiagent Grassroots Logic Programs

We first extend the notion of transition systems to multiagent transition systems, then use them to extend GLP to multiagent GLP, and finally recall the definition of grassroots protocols [62] and prove that multiagent GLP is grassroots.

5.1 Multiagent transition systems and atomic transactions

We assume a potentially infinite set of *agents* Π (think of all the agents that are yet to be born), 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 extend the notion of transition systems (Definition 1) to be multiagent [62,65]:

Informally, a multiagent configuration c over P and a set of local states S can be thought of as an array indexed by agents in P, with $c_p \in S$, the local state of p in c, being the array element in c indexed by p.

Definition 12 (Multiagent Transition System, Degree). *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 $C := S^P$, $c0 := \{s0\}^P$, and $T \subseteq C^2$ being a set of **multiagent transitions** over P and S. For $c \in C$ and $p \in P$, let c_p denote the p-indexed element of c, define TS to be of **degree** k (unary, binary, k-ary) if k is the minimal number such that for every transition $c \to c' \in T$, at most k agents $p \in P$ change their local state, $c_p \neq c'_p$.*

Definition 13 (Transaction, Closure). *Let $P \subset \Pi$, S a set of local states, and $C := S^P$. A **transaction** $t = (c \to c')$ over local states S with **participants** $Q \subset \Pi$ is but a multiagent transition over S and Q, with $t_p := (c_p \to c'_p)$ for any $p \in Q$. For every $P \subset \Pi$ s.t. $Q \subseteq P$, the P-**closure of** t, $t{\uparrow}P$, is the set of transitions over P and S defined by:*

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

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

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. A set of transactions R over S, each with participants $Q \subseteq P$, defines a multiagent transition system as follows:

Definition 14 (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, a* ***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 $Q \subseteq P$. Reference [65] provided transactions-based specification for social networks, grassroots cryptocurrencies, and grassroots federations. Here we do that for multiagent GLP.

5.2 Multiagent GLP

We extend GLP to be multiagent by letting agents' local states to be asynchronous resolvents, have unary Reduce transitions in which agents reduce a local goal and add reader assignments to its pending assignments; and binary Communicate transitions between agents p and q in case p has a pending $X? := T$ and $X?$ occurs in the resolvent of q.

A key difference between GLP and multiagent GLP is in the initial state. In a multiagent transition systems all agents must have the same initial state $s0$. This precludes setting up an initial configuration/goal in which agents share logic variables, as this would imply different initial states for different agents.

We resolve this conundrum 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.

Additional magic is needed to bootstrap communication between agents, so that agents that wish to communicate can have a shared variable to do so with. To address that we assume that the network connecting agents can transfer messages from the network output stream of one agent to the network input stream of another, as specified by the following GLP program template, assuming the network process holds in position p the paired channel of the network channel of p, for every $p \in P$. A full 3-way switch is shown as Program F.5 in Appendix F.

Program 5: Network switch, representative clause

```
% clause for forwarding a message from p to q:
network(...,(p,Chp),...,(q,Chq),.... :-
    receive(Chp?,msg(q,X),Chp1),
```

```
send(Chq?,X?,Chq1) |
network(...,(p,Chp1?),...,(q,Chq1?),....)
```

The Network transaction defined below causes the multiagent GLP system to behave as if agents' network channels were paired to such a `network` process that routes messages between them: Messages sent to agent q via agent p's network output stream appear on agent q's network input stream, realizing communication as specified by the `network` program. However, the network is not another GLP agent; the purpose of the `network` program is solely to provide behavioural specification for the network.

To avoid notational clutter, the Network binary transition below refers to the operation of `network` verbally. It is activated when agent p binds its network output stream tail to a list cell with head $\texttt{msg}(q, X)$, as specified by the `network`.

We leave the specification of 'user' open; assuming people have free will, their behaviour cannot be specified in GLP:) However, users testing or simulating a multiagent GLP program with specific social behaviours can of course be programmed in GLP.

Definition 15 (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 $\texttt{msg}(q, X)$, 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 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. Still, Network and Communicate are essentially identical: in both cases an assignment to a writer in p results in its application to a reader in q.

To show that maGLP computations are deductions, L is augmented so that the resolvent is the union of all local resolvents, the initial goal includes also a `network` goal with $|P|$ channels paired correctly to each agent's initial network channels as in Program **??**, and the module M is augmented with the GLP definition of `network`.

Proposition 6 (Safety Properties of maGLP). *The safety properties established for GLP in Section 3 extend directly to maGLP:*

1. ***SRSW Invariant** (cf. Proposition 3): If the initial goals of all agents in a maGLP run satisfy the SRSW requirement, then every goal in every agent's resolvent throughout the run satisfies the SRSW requirement.*

2. ***Acyclicity*** *(cf. Proposition 4): 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 5): If atom 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.*

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 transitions preserve the properties through binary assignment transfer.

5.3 Multiagent Grassroots Logic Programs are Grassroots

Overview. Here we prove that multiagent Grassroots Logic Programs are indeed *grassroots* [62]. To do so, we recall necessary mathematical foundations:

1. **Protocols:** The notion of grassroots applies to protocols: A *protocol* $\mathcal{F}$ is an infinite family of multiagent transition systems, $\mathcal{F}(P)$ for each set of agents $P \subset \Pi$.
2. **Grassroots:** Informally, proving that a protocol $\mathcal{F}$ is grassroots requires proving for that for any two sets of agents $P \subset P' \subset \Pi$:
   (a) **Oblivious:** Any behaviours available to agents P according to $\mathcal{F}(P)$ are also available to them when they operate within P', namely in $\mathcal{F}(P')$
   (b) **Interactive:** There are behaviours available to agents P operating within $P' \supset P$ not available when they operate on their own in $\mathcal{F}(P)$

We proceed with the definitions.

Definition 16 (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 a designated initial state $s0 \in \mathcal{S}$ and satisfies $P \subset P' \subset \Pi \implies S(P) \subset S(P')$.*

Definition 17 (Protocol). *A **protocol** $\mathcal{F}$ over a local-states function S is a family of multiagent transition systems that has exactly one mts $\mathcal{F}(P) = (C(P), c_0(P), T(P))$ for every $P \subset \Pi$, where $c_p \in S(P)$ and $c0(P)_p = s0$ for every $c \in C(P)$ and $p \in P$.*

Note that maGLP over M and S is a protocol, parameterized by P. Next we recall the notion of a grassroots protocol.

Definition 18 (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 $q \in P' \setminus P$.

We use the notions of projection and closure (Definition 13) to define when a protocol is grassroots:

Definition 19 (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.*

For protocols defined via atomic transactions, such as maGLP, we get the oblivious property "for free", following from the closure construction: transactions defined over $Q \subseteq P$ extend to P by having non-participants remain stationary, ensuring that behaviours available to Q-agents are preserved when operating within the larger set P.

Proposition 7 ([65]). *A transactions-based protocol is oblivious.*

The interactive property requires that agents in P can always potentially interact with agents in $P' \setminus P$, leaving "alien traces" in their local states that could not have been produced by P operating alone. In maGLP this is achieved by the Network transition, in which agent $q \in P' \setminus P$ sends a message with a shared logic variable to agent $p \in P$.

Theorem 1. *maGLP is grassroots.*

6 The Grassroots Social Graph

This section demonstrates how GLP, specified by the multiagent transition systems maGLP, can realize the foundational grassroots platform, the grassroots social graph: the Network transaction enables cold-call connections between disconnected agents, whilst the Communicate transaction provides secure message transfer through established friend channels. Friend-mediated introductions for expanding the network through existing trust relationships are presented in Appendix C.

The grassroots social graph serves as the infrastructure layer for all other grassroots platforms. It enables people to establish authenticated friendships through cryptographically-identified connections. Grassroots platforms built upon this foundation—including grassroots social networks, grassroots cryptocurrencies, and grassroots democratic federations—employ the social graph to establish their platform-specific communication network.

6.1 Protocol Architecture

Each agent maintains its social graph neighbourhood as a friends list containing named bidirectional channels to connected peers. The protocol processes three types of events: connection requests initiated by the agent's user, offers received from other agents through the network, and responses to the agent's own connection attempts. The architecture unifies all communication through a single

merged input stream, with the friends list serving as both the social graph state and the routing table for outgoing messages.

The protocol achieves non-blocking asynchronous operation through GLP's synchronization mechanisms, enabling agents to handle multiple concurrent connection attempts, process friend messages, and respond to user commands simultaneously without deadlock or starvation (see Appendix C.1 for details).

6.2 Initialization and Message Routing

Each agent begins with the goal `agent(Id, ChUser, ChNet)` where `Id` is the agent's unique identifier, `ChUser` provides bidirectional communication with the user interface, and `ChNet` connects to the network for initial message routing. The initialization phase establishes the unified message processing architecture:

Program 6: Social Graph Initialization

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

The initialization extracts the input and output streams from the user and network channels, merges the input streams into a unified stream `In`, and stores the output streams in the initial friends list with special identifiers "user" and "net". This design treats the user interface and network as special cases of friends, enabling uniform message sending through the `lookup_send` procedure regardless of destination type.

6.3 Cold Call Protocol

The cold call mechanism enables agents to establish friendship without prior shared variables. When agent p wishes to befriend agent q, the protocol proceeds through four phases: user p initiation, p to q offer transmission, user q consultation, and if the response is positive then $p - q$ channel establishment.

Program 7: Social Graph Cold-Call Befriending Protocol

```
% Process user request to connect (self-introduction)
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?, Fs1),
    inject(Resp?, msg(Target, Id, response(Resp)), In?, In1),
    social_graph(Id, In1?, Fs1?).

% Process received self-introduction
social_graph(Id, [msg(From, Id, intro(From, From, Resp))|In], Fs) :-
    ground(Id), attestation(intro(From, From, Resp), att(From, _)) |
    lookup_send(user, msg(agent, user, befriend(From?, Resp)), Fs?, Fs1),
```

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

% Process user decision on received introduction
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 sent 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?).

% Application message handling
social_graph(Id, [msg(From, To, Content)|In], Fs) :-
    ground(Id), otherwise |
    % Forward to application layer
    social_graph(Id, In?, Fs?).

inject(X,Y,Ys,[Y?|Ys?]) :- known(X) | true.
inject(X,Y,[Y1|Ys],[Y1?|Ys1?]) :- unknown(X) | inject(X?,Y?,Ys?,Ys1).
```

The first clause handles user-initiated connections by sending an offer containing an unbound response variable through the network. The `inject` procedure defers insertion of the response message into the input stream until the response variable becomes bound, while allowing the stream to continue flowing. The second clause receives offers from other agents and forwards them to the user interface for approval, including the response variable that the user's decision will bind. The third clause processes user decisions, calling `bind_response` to handle acceptance or rejection. The fourth clause handles responses to the agent's own offers.

When `X` is known, `inject` inserts the message at the output stream and terminates. Until then, it passes input stream messages to its output. This ensures the protocol remains responsive while awaiting responses to its own connection attempts.

6.4 Channel Establishment and Response Handling

The protocol's response handling demonstrates sophisticated use of GLP's concurrent programming capabilities. When an offer is accepted, both agents must establish symmetric channel configurations and merge the new friend's input stream into their main processing loop:

Program 8: Response Processing

```
% Bind response based on user decision
```

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

% Handle response (for both received and sent offers)
handle_response(accept(ch(FIn, FOut)), From, Fs, [(From, FOut)|Fs], In, In1) :-
    tag_stream(From?, FIn?, Tagged),
    merge(In?, Tagged?, In1).
handle_response(no, _, Fs, Fs, In, In).
```

When accepting an offer, `bind_response` creates a new channel pair using `new_channel`, which produces two channels with crossed input/output streams. The acceptor retains one channel and sends the other through the response variable, ensuring both agents receive complementary channel endpoints. The `handle_response` procedure, called for both accepted sent offers and accepted received offers, adds the friend's output stream to the friends list and merges the tagged input stream into the main message flow. The stream tagging preserves sender identity after merging, enabling the agent to determine message origin.

6.5 Friend-Mediated Introductions

Beyond cold calls, the social graph protocol enables friend-mediated introductions, leveraging existing trust relationships to establish new connections. When agent r is friends with both p and q, it can introduce them to each other, creating a direct communication channel between them. The protocol proceeds through five phases: (1) the introducer creates paired channels and sends introduction messages, (2) recipients initiate attestation exchange through the new channel, (3) attestation requests are verified and responded to, (4) verified introductions prompt user consultation, and (5) user acceptance establishes the connection.

Program 9: Friend-Mediated Introduction Protocol

```
% Friend introduces two others
social_graph(Id, [msg(user, Id, introduce(P, Q))|In], Fs) :-
    ground(Id), ground(P), ground(Q),
    new_channel(ch(PQIn, PQOut), ch(QPIn, QPOut)) |
    lookup_send(P, msg(Id, P, intro(Q?, ch(QPIn?, PQOut?))), Fs?, Fs1),
    lookup_send(Q, msg(Id, Q, intro(P?, ch(PQIn?, QPOut?))), Fs1?, Fs2),
    social_graph(Id, In?, Fs2?).

% Process introduction - initiate attestation exchange
social_graph(Id, [msg(From, Id, intro(Other, ch(In, Out)))|In], Fs) :-
    ground(Id), attestation(intro(Other, ch(In, Out)), att(From, _)) |
    Out = [attest_req(Id?, AttResp)|Out1?],
    inject(AttResp?, msg(Other, Id, verified_intro(From?, Other?, ch(In?, Out1?))),
           In?, In1),
```

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

% Process attestation request and send verification
social_graph(Id, [msg(From, Id, attest_req(From, AttResp))|In], Fs) :-
    ground(Id), attestation(attest_req(From, AttResp), att(From, Module)) |
    AttResp = verified(Id?, Module?),
    social_graph(Id, In?, Fs?).

% Attestation verified - now ask user
social_graph(Id, [msg(Other, Id, verified_intro(Introducer, Other, Ch))|In], Fs) :-
    ground(Id),
    attestation(verified_intro(Introducer, Other, Ch), att(Other, Module)) |
    lookup_send(user, msg(agent, user,
                befriend_verified(Introducer?, Other?, Module?, Ch?)), Fs?, Fs1),
    social_graph(Id, In?, Fs1?).

% User accepts verified introduction
social_graph(Id, [msg(user, Id, decision(yes, Other, ch(In, Out)))|In], Fs) :-
    ground(Id) |
    tag_stream(Other?, In?, Tagged),
    merge(In?, Tagged?, In1),
    social_graph(Id, In1?, [(Other?, Out?)|Fs?]).
```

Friend-mediated introductions provide stronger trust assurance than cold calls through double verification. The introducer r creates a fresh channel pair connecting p and q, sending each party one of the paired channels, along with the identity of the other party. Recipients first verify through the signature and attestation that the introduction genuinely originates from their mutual friend r running verified code. Before accepting the connection, the introduced parties p and q exchange signed and attested messages through the new channel, allowing each to verify the other's identity through signatures and code compatibility through attestations.

This double verification mechanism addresses two distinct security requirements. The introducer's signature and attestation prevent forgery—the signature proves the introduction came from r while the attestation confirms it was produced by legitimate social graph code. The signatures and attestations exchanged between introduced parties ensure they are indeed who the introducer claims, with signatures providing cryptographic proof of identity and attestations ensuring code compatibility.

Unlike cold calls which require external identity verification, friend-mediated introductions provide both the introducer's social vouching and direct cryptographic verification from the introduced party through their signatures. The mutual friend serves as a trusted intermediary who facilitates the connection, while the exchange of signed and attested messages between parties ensures the connection's authenticity independent of the introducer.

7 Securing Multiagent Grassroots Logic Programs

7.1 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 C) 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 8 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, assiging 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 implementations should employ standard cryptographic optimizations [39]: 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 [17] or ECDH [23], 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.

7.2 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.

7.3 Security of the Social Graph Protocol

Authenticated Connection Establishment. Cold call offers carry attestation $(\texttt{msg}(q, \texttt{offer}(\texttt{Resp})))_{M,p,q}$ 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.

7.4 Blockchain Security of GLP Streams

Authenticated GLP streams achieve blockchain security properties [44,21] 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 4 guarantees no circular terms. The occurs check prevents any writer from being bound to a term containing its paired 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 [21]. 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 F.4 in Appendix F), 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 F.8), they form a blocklace [1]—a DAG where blocks reference multiple predecessors—employed by modern consensus protocols including Cordial Miners [29], Morpheus [33], and

Constitutional Consensus [28]. The resulting structure provides eventual consistency equivalent to Byzantine fault-tolerant CRDTs [69] 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.

8 Implementation

The implementation of Grassroots Logic Programs on smartphones requires cross-platform mobile deployment, garbage-collected memory management, lightweight concurrency, cryptographic operations, and TEE attestation access. The Dart programming language [15], deployed via Flutter [18], satisfies these requirements. Flutter compiles to native iOS and Android applications from a single codebase, while Dart's event loop with microtask scheduling maps naturally to GLP's operational semantics. Flutter plugins provide access to Google Play Integrity [22] and Apple App Attest [3], enabling TEE-based peer verification. Server infrastructure supports initial attestation and NAT traversal via STUN [51], TURN [38], and ICE [46], but core GLP execution remains peer-to-peer on smartphones. While React Native [40] and Kotlin Multiplatform [27] are popular alternatives, they lack either Dart's concurrency model [15] or Flutter's unified cross-platform deployment with TEE access [18], both essential for implementing GLP's multiagent semantics with attestation.

Secure implementation on smartphones. On current smartphones, secure multiagent GLP is realized through Trusted Execution Environments (TEEs) with hardware providers (e.g., ARM TrustZone [47]) as trust anchors, combined with OS-level attestation services (Google Play Integrity [22], Apple App Attest [3]) with OS providers as trust anchors. This infrastructure authenticates and attests to the integrity of the sender and prevents tampering while ensuring confidentiality.

Architecture. The Dart implementation maps the formal 'implementation-ready' multiagent GLP semantics (detailed in Appendix **??**) to concrete smartphone operations. Each agent maintains its resolvent as Dart microtasks with three goal categories: active (queued for reduction), suspended (awaiting variable assignments), and failed (permanently blocked). A shared variable table tracks creator-holder relationships for distributed variables, enabling message routing without consensus protocols.

The implementation preserves GLP's three core transactions. **Reduce** performs goal/clause reduction within Dart microtasks, generating assignments for remote readers that enter the message queue M_p. **Communicate** delivers these assignments across agents via encrypted, signed, and attested messages routed through variable creators, with the Dart event loop processing received messages and updating the variable table V_p. **Network** handles initial channel establishment for cold calls using platform-specific APIs (WebRTC for peer-to-peer,

HTTPS for NAT traversal). The single-reader/single-writer invariant eliminates distributed unification and is enforced through exclusive variable table tracking, while creator-mediated routing ensures messages reach their destinations despite variable migration. Variable abandonment detection runs as a periodic microtask, scanning for unreachable variables and generating appropriate abandonment messages.

Security. Security enforcement occurs at message boundaries as specified in Section 7.1. While the formal specification in Appendix H requires attestation, signature and encryption for every message, practical implementations employ the standard cryptographic optimizations described in Section 7.1—including intermittent attestation verification and session-key-based channels—to reduce computational overhead while maintaining security guarantees.

9 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.

Distributed actor and process languages. Actor-based languages (Erlang/OTP [4], Akka [35], Pony [11]) and active object languages [7,6] provide message-passing concurrency and fault isolation. However, their security models operate at the transport layer (TLS in Akka Remote [35], Erlang's cookie-based authentication [4]) rather than integrating cryptographic identity and code attestation into language primitives. Orleans [41] assumes trusted runtime environments, lacking the attestation mechanisms required for grassroots platforms where participants must verify code integrity without central coordination.

Capability security. E [42] 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 [74] 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 [25] specify communication protocols statically, with implementations in Links [12,36], Rust [26], Scala [54], and Go [9]. 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 [48] provides first-class synchronous channels and events. The Join Calculus [19] offers pattern-based synchronization through join patterns. GLP's SRSW variables provide

asynchronous communication through reader/writer pairs with the monotonicity property (Proposition 5) 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 [43] and Move [72] provide deterministic execution and asset safety but assume blockchain infrastructure for identity and consensus. While Scilla [55] 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 blockchain security properties (Section 7.4) through the language-level SRSW invariant and attestations, without requiring global consensus.

Authorization languages. OPA/Rego [45] and Cedar [24] 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.

Concurrent logic programming. Concurrent logic programming languages [60] extend logic programming with shared variables for process synchronization. Concurrent Prolog [59] introduced reader/writer variables, while PARLOG [10] and GHC [73] used mode declarations. Unlike these, GLP enforces Single-Reader/Single-Writer (SRSW) restriction where each variable occurs at most once, establishing secure point-to-point channels through exclusive reader/writer pairs. This enables authenticated messaging while eliminating distributed atomic unification [30]. GLP's homoiconic nature inherits logic programming's metaprogramming capabilities [53,34,58], essential for platform tooling development.

10 Conclusion

We have presented secure, multiagent, concurrent GLP, argued for its utility for implementing grassroots platforms, and provided workstation and smartphone implementation-ready specifications for it. The next step is to implement it.

References

1. Paulo Sérgio Almeida and Ehud Shapiro. The blocklace: A byzantine-repelling and universal conflict-free replicated data type. *arXiv preprint arXiv:2402.08068*, 2024.
2. Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.
3. Apple Developer Documentation. Implementing App Attest. `https://developer.apple.com/documentation/devicecheck/implementing_app_attest`, 2024. `https://developer.apple.com/documentation/devicecheck/implementing_app_attest`.

4. Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2nd edition, 2013.
5. Keyvan Azadbakht, Frank S de Boer, Nikolaos Bezirgiannis, and Erik de Vink. A formal actor-based model for streaming the future. *Science of Computer Programming*, 186:102341, 2020.
6. Frank de Boer, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Eduard Kamburjan. *Active Object Languages: Current Research Trends*, volume 14360. Springer, 2024.
7. Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, et al. A survey of active object languages. *ACM Computing Surveys (CSUR)*, 50(5):1–39, 2017.
8. Colin Boyd and Anish Mathuria. *Protocols for authentication and key establishment*. Springer, 2003.
9. David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go. *Proceedings of the ACM on Programming Languages*, 3(POPL):29:1–29:30, 2019.
10. Keith Clark and Steve Gregory. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):1–49, 1986.
11. Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12, 2015.
12. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.
13. Victor Costan and Srinivas Devadas. Intel sgx explained. In *Cryptology ePrint Archive*, 2016.
14. George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley, Boston, MA, 5th edition, 2011.
15. Dart Team. Isolates - concurrency in dart. `https://dart.dev/guides/concurrency/isolates`, 2023. Accessed July 2025.
16. Tamino Dauth and Martin Sulzmann. Futures and promises in haskell and scala. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 68–74, 2019.
17. Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.
18. Flutter Team. Flutter - build apps for any screen. `https://flutter.dev`, 2024. Accessed July 2025.
19. Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. *Proceedings of POPL'96*, pages 372–385, 1996.
20. Haim Gaifman and Ehud Shapiro. Fully abstract compositional semantics for logic programs. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 134–142, 1989.
21. Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.
22. Google Play Developer Documentation. Play Integrity API. `https://developer.android.com/google/play/integrity`, 2024. `https://developer.android.com/google/play/integrity`.

23. Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
24. Craig Hicks, Anuj Datta, Kesha He, John Kasampalis, Neha Khanna, Madison Lampson, William Lee, Shubham Mehta, Anand Rengarajan, Emina Thakkar, Radu Vanciu, and Aaron Warden. 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)*, 2023.
25. Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, pages 509–523. Springer, 1993.
26. Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, pages 13–22. ACM, 2015.
27. JetBrains. Kotlin multiplatform - share code across platforms. `https://kotlinlang.org/docs/multiplatform.html`, 2024. Accessed October 2025.
28. Idit Keidar, Andrew Lewis-Pye, and Ehud Shapiro. Constitutional consensus, 2025.
29. Idit Keidar, Oded Naor, and Ehud Shapiro. Cordial miners: A family of simple and efficient consensus protocols for every eventuality. In *37th International Symposium on Distributed Computing (DISC 2023)*. LIPICS, 2023.
30. Alon Kleinman, Yoram Moses, and Ehud Shapiro. Distributed variable server for atomic unification. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 59–74, 1990.
31. Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–574, 1974.
32. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
33. Andrew Lewis-Pye and Ehud Shapiro. Morpheus consensus: Excelling on trails and autobahns. *arXiv preprint arXiv:2502.08465*, 2025.
34. Yossi Lichtenstein and Ehud Shapiro. Concurrent algorithmic debugging. *ACM SIGPLAN Notices*, 24(1):248–260, 1988.
35. Lightbend Inc. Akka: Build concurrent, distributed, and resilient message-driven applications. `https://akka.io`, 2022. Accessed October 2025.
36. Sam Lindley and J Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools*, pages 265–286, 2017.
37. John W Lloyd. *Foundations of logic programming*. Springer, 1987.
38. R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to STUN. `https://datatracker.ietf.org/doc/html/rfc5766`, 2010. RFC 5766, `https://datatracker.ietf.org/doc/html/rfc5766`.
39. Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 1996.
40. Meta Platforms. React native - build native mobile apps using javascript and react. `https://reactnative.dev`, 2024. Accessed October 2025.
41. Microsoft. Orleans: Cloud native application framework. `https://dotnet.github.io/orleans`, 2022. Accessed October 2025.
42. Mark S Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.
43. Mayukh Mukhopadhyay. *Ethereum Smart Contract Development: Build blockchain-based decentralized applications using solidity*. Packt Publishing Ltd, 2018.

44. Satoshi Nakamoto and A Bitcoin. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 4, 2008.
45. Open Policy Agent Contributors. Open policy agent. `https://www.openpolicyagent.org`, 2021. Cloud Native Computing Foundation.
46. M. Petit-Huguenin, A. Keranen, and C. Holmberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. `https://datatracker.ietf.org/doc/html/rfc8445`, 2018. RFC 8445, `https://datatracker.ietf.org/doc/html/rfc8445`.
47. Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys*, 51(6):1–36, 2019.
48. John H Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.
49. Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
50. John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.
51. J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). `https://datatracker.ietf.org/doc/html/rfc5389`, 2008. RFC 5389, `https://datatracker.ietf.org/doc/html/rfc5389`.
52. Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.
53. Shmuel Safra and Ehud Shapiro. Meta interpreters for real. In *Concurrent Prolog: Collected Papers*, pages 166–179. MIT Press, 1988.
54. Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, pages 21:1–21:28. Schloss Dagstuhl, 2016.
55. Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 366–381. ACM, 2019.
56. Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
57. Ehud Shapiro. Alternation and the computational complexity of logic programs. *The Journal of Logic Programming*, 1(1):19–33, 1984.
58. Ehud Shapiro. Systems programming in concurrent prolog. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 93–105, 1984.
59. Ehud Shapiro. *Concurrent Prolog: collected papers (Vols. I and II)*. MIT press, 1987.
60. Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR)*, 21(3):413–510, 1989.
61. Ehud Shapiro. Multiagent transition systems: Protocol-stack mathematics for distributed computing. *arXiv preprint arXiv:2112.13650*, 2021.
62. Ehud Shapiro. Grassroots distributed systems: Concept, examples, implementation and applications (brief announcement), 2023.
63. Ehud Shapiro. Grassroots social networking: Serverless, permissionless protocols for twitter/linkedin/whatsapp. In *OASIS '23*. Association for Computing Machinery, 2023.
64. Ehud Shapiro. Grassroots currencies: Foundations for grassroots digital economies. *arXiv preprint arXiv:2202.05619*, 2024.

65. Ehud Shapiro. Grassroots platforms with atomic transactions: Social networks, cryptocurrencies, and democratic federations, 2025.
66. Ehud Shapiro and Colin Mierowsky. Fair, biased, and self-balancing merge operators: Their specification and implementation in concurrent prolog. *New Generation Computing*, 2(3):221–240, 1984.
67. Ehud Shapiro and Shmuel Safra. Multiway merge with constant delay in concurrent prolog. *New Generation Computing*, 4(2):211–216, 1986.
68. Ehud Shapiro and Nimrod Talmon. Grassroots federation: Fair governance of large-scale, decentralized, sovereign digital communities. *arXiv preprint arXiv:2505.02208*, 2025.
69. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 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*, pages 386–400. Springer, 2011.
70. William Silverman, Michael Hirsch, Avshalom Houri, and Ehud Shapiro. The logix system user manual version 1.21. In *Concurrent Prolog: Collected Papers*, pages 46–77. 1988.
71. Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT press, 1994.
72. The Diem Association. Move: A language with programmable resources. `https://github.com/move-language/move`, 2022. Accessed October 2025.
73. Kazunori Ueda. Guarded horn clauses. In *Logic Programming '85*, volume 221 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 1986.
74. Philip Wadler. Linear types can change the world. *Programming concepts and methods*, 2:347–359, 1990.

A Logic Programs Syntax

Definition 20 (Logic Programs Syntax). *The syntax of Logic Programs is defined thus:*

- *A* ***variable*** *is an alphanumeric string beginning with uppercase letter, e.g.* `X, X1, Xs`*. We use V to denote the set of all variables.*
- *A* ***constant*** *is a string beginning with a lowercase letter, e.g. a, a1, and foo, as well as any quoted string, e.g.* `","` *and* `"X"`*.*
- *A* ***number*** *is a numeric string, which may include a decimal point, e.g. 0, 1, 103.65.*
- *A* ***logic term****, or* ***term*** *for short is a variable in V, a constant, a number, as well as a* ***composite term*** *of the form $f(T_1, T_2, \ldots, T_n)$, $n \geq 1$, where f is a constant and each T_i is a term, $i \in [n]$, referred to as a* ***subterm*** *of T.*
- *A term T* ***occurs*** *in term T', denoted $T \in T'$, if $T = T'$ or if T' is an n-ary term $f(T_1, T_2, \ldots, T_n)$ for some constant f and T occurs in T_i for some $i \in [n]$. A term is* ***ground*** *if it contains no variables, namely $X \notin T$ for any $X \in V$. We let $\mathcal{T}$ denote the set of all terms.*
- ***Lists:*** *By convention the constant* `[]` *(read "nil") represents an empty list, the binary term* `[X|Xs]` *represents a (linked) list with the first element* `X` *and (a link to the) rest* `Xs`*, the term* `[X]` *is a shorthand for* `[X|[]]` *and the term* `[X1,X2,...Xn]` *is a shorthand for the nested term* `[X1|[X2|...[Xn|[]]...]]`*.*

- *An* ***atom*** *is a constant or a composite term.*
- *A* ***goal*** *is a term of the form $a_1, a_2, \ldots a_n$, $n \geq 0$, where each a_i is an atom, $i \in [n]$. Such a goal is* ***empty*** *if $n = 0$, in which case it may also be written as* `true`*,* ***atomic*** *if $n = 1$, and* ***conjunctive*** *if $n \geq 2$. A conjunctive goal can be written equivalently as $(a_1, (a_2, (\ldots a_n) \ldots))$, where (a, b) is a shorthand for* `","`(a, b)*. As goal order is immaterial here, a conjunctive goal is identified with a multiset of its atoms and an atomic goal with its singleton. Let $\mathcal{A}$ denote the set of all atoms and $\mathcal{G}$ the set of all goals.*
- *A* ***clause*** *is a term of the form* `A :- B` *(read 'A if B'), where A is an atom, referred to as the clause's* ***head****, and B is a (possibly empty) goal, referred to as the clause's* ***body****. If B is empty then the clause is called* ***unit*** *and can be written simply as A. The underscore symbol* `_` *is a* don't-care variable *that stands for a variable occurring only once, which can be bound to any value that subsequently cannot be unified.*
- *A* ***logic program*** *is a finite sequences of "."-separated clauses. As a convention, clauses for the same predicate (name and arity) are grouped together and are referred to as the* ***procedure*** *for that predicate. Given logic program M, let $\mathcal{A}(M)$ and $\mathcal{G}(M)$ be the subsets of $\mathcal{A}$ and $\mathcal{G}$, respectively, that include only the vocabulary (constant, function, and predicate symbols) of M.*

B Proofs

Proposition 1 (LP Computation is Deduction). *The outcome $(G_0 \text{ :- } G_n)\sigma$ of a proper run $\rho : G_0 \xrightarrow{\sigma_1} G_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} G_n$ of LP, where $\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$, is a logical consequence of M.*

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

Lemma 1 (Reader-Instance). *In any GLP run $G_0 \to G_1 \to \cdots$, if $G_i \to G_{i+1}$ via reduction with substitution $\sigma?$ does not reduce $A \in G_i$, then $A\tau \in G_{i+1}$ where τ instantiates only readers.*

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

By Definition 10, the Reduce transition specifies that $G_{i+1} = (G_i \backslash \{A'\} \cup B)\sigma$, 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 $\sigma?$. □

Proposition 2 (GLP Computations are Deductions). *For any finite GLP run r, let $L(r) = G0 \xrightarrow{\sigma_1} G_1 \xrightarrow{\sigma_2} \ldots G_n$, with $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_n$, then $(G_: - G_n)\sigma$ is a logical consequence of $L(M)$.*

Proof. Follows from the correspondence between GLP reductions and LP reductions on pure logic variants, combined with Proposition 1. □

Proposition 3 (SRSW Invariant). *If the initial goal G_0 in a GLP run satisfies SRSW, then every goal in the run satisfies SRSW.*

Proof. By induction on run length. The base case holds by assumption. For the inductive step, consider $G_i \to G_{i+1}$ via reduction with clause C renamed apart. The renamed clause has fresh variables satisfying the SRSW syntactic constraint.

The reduction replaces atom A with body B and applies $\sigma?$. Since $\sigma?$ replaces variables with terms (eliminating variable occurrences rather than duplicating them), and B has fresh variables distinct from G_i, the SRSW invariant is preserved in G_{i+1}. □

Proposition 4 (Acyclicity). *If the initial goal G_0 in a GLP run contains no circular terms, then no goal in the run contains a circular term.*

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 $G_i \rightarrow G_{i+1}$ via reduction of atom A with clause C. Let $(H\texttt{:-}B)$ 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. □

Proposition 5 (Monotonicity). *In any GLP run $G_0 \rightarrow G_1 \rightarrow \cdots$, if atom $A \in G_i$ can reduce with clause C, then for any $j > i$, either A has been reduced by step j, or there exists $A' \in G_j$ where $A' = A\tau$ for some reader substitution τ, and A' can reduce with C.*

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 $k + 1$. □

Theorem 1. *maGLP is grassroots.*

Proof. We prove that maGLP is oblivious and interactive.

1. **maGLP is Oblivious:** Follows directly from Proposition 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. □

C Grassroots Social Graph Protocol Properties

C.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 6.3 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.

C.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.

D 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.

D.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 10: 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 authenticated friend
    handle_friend(From?, FIn?, FOut?, DMIn?, DMOut?, Fs?, Fs1, In?, In1).

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.

D.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 11: 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.

D.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 12: 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))|In], Fs) :-
    social_graph(Id, In?, [((Founder,Name), group(member, Streams?))|Fs?]).
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 13: 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 F.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.

D.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.

E Guards and System Predicates

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

Guard predicates. Guards provide read-only access to the runtime state of GLP computation. A guard appears after the clause head, separated by `|`, and must be satisfied for the clause to be selected. The following guards are fundamental for concurrent GLP programming:

- `ground(X)` succeeds if `X` contains no variables. With this guard, the clause body may contain multiple occurrences of `X?` without violating the single-writer requirement, enabling safe replication of ground terms to multiple concurrent consumers.
- `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` succeed if X and Y are identical
- `X=\=Y` succeeds if the unification of X and Y fails.

Defined guard predicates. To support abstract data types and cleaner code organization, GLP provides for user-defined guards, defined unit clauses `p(T1,...,Tn)`. The call `p(S1,...,Sn)` in the guard is folded to the equalities `T1=S1,...,Tn=Sn`

for each unit goal. This mechanism is demonstrated in the channel abstractions below.

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.

F Additional Programming Techniques

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

F.1 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 14: 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 15: 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.

F.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 16: 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.

F.3 Stream Observation

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

Program 17: 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).
```

F.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 18: 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.

F.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 19: 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?)).
```

F.6 Implementation Correctness Properties

Proposition 8 (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 31) 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 9 (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 25), 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 10 (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 25) 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.

F.7 Replication of Non-Ground Terms

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

Program 20: Non-Ground Term Replicator

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

```
    Xs? = [X|Xs1] |
    replicate(X?, Y1,..., Yn),
    replicate(Xs1?, Ys1,..., Ysn).
```

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

F.8 Interlaced Streams as Distributed Blocklace

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

Program 21: 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.

F.9 Metainterpreters

Program development is essentially a single-agent endeavour: The programmer trying to write and debug a GLP program. As in Concurrent Prolog, a key strength of GLP is metainterpretation: The ability to write GLP interpreters with various functions in GLP. This allows writing a GLP program development environment and a GLP operating system within GLP itself [71,53,56,34,70], as well as writing a GLP operating system in GLP [58]. 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 22: 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?,Ys?,Zs)).
```

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

Fail-safe metainterpreter. The operational semantics of Logic Programs and Grassroots Logic Programs 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 23: 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 24: GLP metainterpreter with run control

```
run(true,_).  % halt
run((A,B),Cs) :- distribute(Cs?,Cs1,Cs2), run(A?,Cs1?), run(B?,Cs1). % fork
run(A,[suspend|Cs]) :- suspended_run(A,Cs?).   % suspend
run(A,Cs) :- known(A) |        % reduce
        distribute(Cs?,Cs1,Cs2), reduce(A?,B,Cs1?), run(B?,Xs,Cs2?).

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 F.9 and F.9 would allow the programmer to abort the run as soon as a goal fails. But we wish to introduce additional capabilities before integrating them all.

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

Program 25: 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 26: 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 27: 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 [58], 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 28: 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?) %  reduce
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 [71] 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).

G Workstation Implementation-Ready Transition System for GLP

This section specifies a workstation (single-agent) implementation-ready transition system for GLP with deterministic execution.

Definition 21 (irGLP Configuration). *An* irGLP configuration *over program M is a triple $R = (Q, S, F)$ where:*

- $Q \in \mathcal{A}^*$ *is a sequence of active goals*
- $S \subseteq \mathcal{A} \times 2^{V?}$ *contains suspended goals with their suspension sets*
- $F \subseteq \mathcal{A}$ *contains failed goals*

The irGLP reduction extends GLP reduction by activating goals that were suspended on variables instantiated by the reduction, and explicitly failing goals that do not succeed or suspend.

Definition 22 (irGLP Goal/Queue Reduction). *Given configuration (Q, S, F) with $Q = A \cdot Q'$ and clause $C \in M$, the* irGLP reduction *of A with C:*

- ***succeeds with*** *$(B, \hat{\sigma}, R)$ if the GLP reduction of A with C succeeds with $(B, \hat{\sigma})$ and $R = \{G : (G, W) \in S \wedge X? \in W \wedge X?\hat{\sigma}? \neq X?\}$*
- ***suspends with*** *W_C if GLP reduction of A with C suspends on readers W_C*
- ***fails*** *otherwise*

Definition 23 (Implementation-Ready GLP Transition System). *The transition system $irGLP = (\mathcal{C}, c_0, \mathcal{T})$ over M and initial goal G_0 has configurations $\mathcal{C}$ being all irGLP configurations over M, with initial configuration $c_0 = (G_0, \emptyset, \emptyset)$, and transitions $\mathcal{T}$ being all transitions $(Q, S, F) \rightarrow (Q', S', F')$ where $Q = A \cdot Q_r$ and:*

1. ***Reduce:*** *If GLP reduction of A with first applicable clause $C \in M$ succeeds with $(B, \hat{\sigma}, R)$:*
   - ***Activate:*** *$S' = S \setminus \{(G, W) : G \in R\}$, $F' = F$*

- ***Schedule:*** $Q' = (Q_r \cdot B \cdot R)\hat{\sigma}\hat{\sigma}?$

2. ***Suspend:*** *Else if* $W = \bigcup_{C \in M} W_C \neq \emptyset$ *then* $Q' = Q_r$, $S' = S \cup \{(A, W)\}$, $F' = F$
3. ***Fail:*** *Else,* $Q' = Q_r$, $S' = S$, $F' = F \cup \{A\}$.

A key restriction compared to the GLP operational semantics is the immediate application of reader substitutions during reduction rather than through asynchronous communication. This simplification is appropriate for workstation execution where all variables are local.

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 5). 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.

Definition 24 (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 25*
- c_0 *is the initial configuration where for each* $p \in P$*:*
  - $R_p = ([\texttt{agent}(p, \texttt{ch}(?,) , \texttt{ch}(?,))], \emptyset, \emptyset)$
  - $V_p = \emptyset$
  - $M_p = \emptyset$
- T *is the union of all transitions generated by:*
  - *Unary Reduce transactions for each* $p \in P$ *(Definition 31)*
  - *Binary Communicate transactions for each* $(p, q) \in P \times P, p \neq q$ *(Definition 32)*
  - *Binary Network transactions for each* $(p, q) \in P \times P, p \neq q$ *(Definition 33)*

H.1 Local States

Definition 25 (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 $(Y, q, s) \in V_p$:*
   - ***Writer:*** *$Y \in V$, $s \in \mathcal{T}$ 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, 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 26 (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 27 (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 28 (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*.*

Definition 29 (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.2 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 30 (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 31 (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 V'_p: for each $X? \in W$ where $(X?, q, \bot) \in V'_p$, update to $(X?, q, q)$*
   - *Update M'_p: add $(X? := T, r)$ for each $\{X? := T\} \in \hat{\sigma}?$ where $(X?, p, r) \in V'_p, r \neq \bot$*
   - *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 32 (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*** *$m = (X? := T)$ where $X?$ is local to q:*
   - *Let $R =$ 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* $(X?, \cdot, \cdot)$ *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*** $m = request(X?, p)$*:*
   - *If* $p = \bot$ *then call abandon*$(X?)$ *for agent* q *(modifies* V'_q *and* M'_q*)*
   - *Else if* $(X?, q, \bot) \in V'_q$ *then 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 33 (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* $X' := export(X)$ *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.3 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:

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

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

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 32 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.

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

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.