Implementing Grassroots Logic Programs with Multiagent Transition Systems and AI

Ehud Shapiro
London School of Economics
Weizmann Institute of Science
LOPSTR+PPDP 2026, Indianapolis

1st ICLP 1982

Ehud Y. Shapiro:
Alternation and the Computational Complexity of Logic Programs. ICLP 1982: 154-163

Spoke about concurrent logic programming

42nd ICLP 2026

16:00-16:15 **GLP: A Grassroots, Multiagent, Concurrent, Logic Programming Language** (abstract) 15 min

Ehud Shapiro[1]

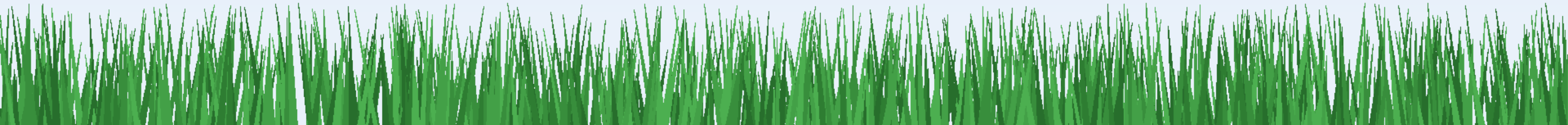
[1] *London School of Economics*

Spoke again about concurrent logic programming

Today's paper is about **implementing GLP**

Why a new Concurrent Logic Programming Language?

- We want to build something, with AI
- Existing languages are not good enough



Implementing Grassroots Logic Programs

1. What do we want to build?
2. Why new languages?
3. What is Grassroots Logic Programs (GLP)?
4. Why is this language good for the task?
5. How is it being implemented (with AI)? [LOPSTR+PPDP'26]

All covered by papers on arXiv

Implementing Grassroots Logic Programs

1. **What do we want to build?**
2. Why new languages?
3. What is Grassroots Logic Programs (GLP)?
4. Why is this language good for the task?
5. How is it being implemented (with AI)?

All covered by papers on arXiv

We want to build *Grassroots Platforms*

Today: We are ***users*** of ***global platforms*** owned, operated and governed by others who extract the value we generate.

Vision: We will be ***participants*** in ***grassroots platforms*** owned, operated and governed by us, retaining the value we generate.

Grassroots Platforms we want to build

1. **Grassroots social networks**, on par with Facebook, X, WhatsApp,… (but *child-safe* and *AI-resilient*)
2. **Grassroots currencies**, on par with Bitcoin, Ethereum
3. **Grassroots cooperatives**, on par with ”sharing economy” Uber, Deliveroo, Airbnb,…
4. **Grassroots digital economy**, on par with Amazon Marketplace,…

Defining Grassroots Platforms

Formally: (a lot of math needed to formalize)

1. Can have **multiple instances**, independent of each other and any global resource but the net
2. Instances may **coalesce**, possibly into a single global instance

Practically: A grassroots platform is constituted from its participants' smartphones and the grassroots app each participant runs.

Implementing Grassroots Logic Programs

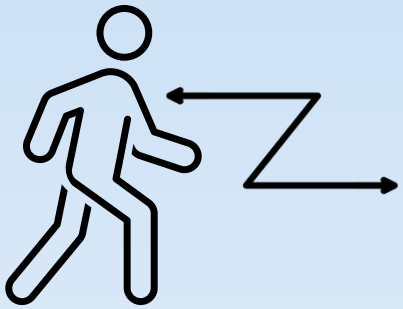
1. What do we want to build?
2. **Why new languages?**
3. What is Grassroots Logic Programs (GLP)?
4. Why is this language good for the task?
5. How is it being implemented (with AI)?

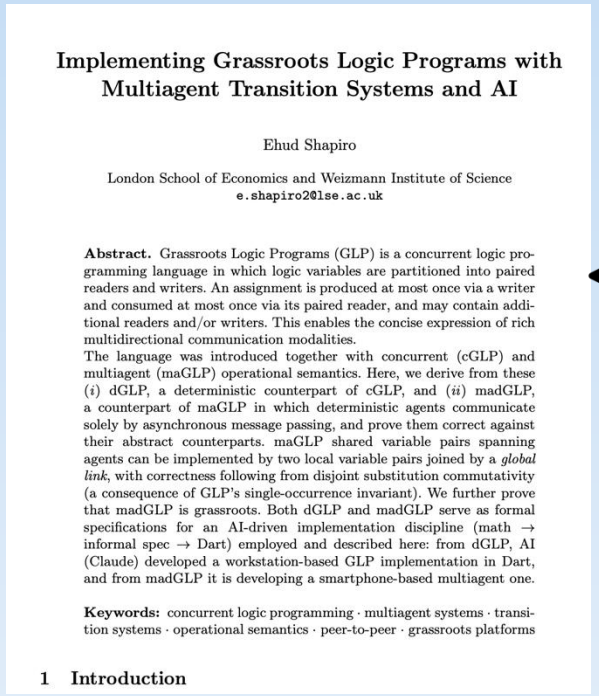
All covered by papers on arXiv

Why new languages?

- AI is the **best programmer** and will be the **last programmer**
- Hence we will build grassroots platforms with AI
- How can AI build what we want?
- Two-prong solution for AI coding:
  1. The **Human–Science–AI programming methodology**: Scientific papers are the interface between us humans and AI, and from which AI codes
  2. **Abstraction cascade programming:** Guiding AI via abstractions **(= languages**) and implementations among them

The Human–Science–AI programming methodology





Implementing Grassroots Logic Programs with Multiagent Transition Systems and AI

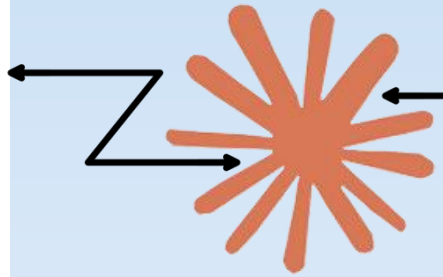
Ehud Shapiro

London School of Economics and Weizmann Institute of Science
e.shapiro2@lse.ac.uk

Abstract. Grassroots Logic Programs (GLP) is a concurrent logic programming language in which logic variables are partitioned into paired readers and writers. An assignment is produced at most once via a writer and consumed at most once via its paired reader, and may contain additional readers and/or writers. This enables the concise expression of rich multidirectional communication modalities.
The language was introduced together with concurrent (cGLP) and multiagent (maGLP) operational semantics. Here, we derive from these (*i*) dGLP, a deterministic counterpart of cGLP, and (*ii*) madGLP, a counterpart of maGLP in which deterministic agents communicate solely by asynchronous message passing, and prove them correct against their abstract counterparts. maGLP shared variable pairs spanning agents can be implemented by two local variable pairs joined by a *global link*, with correctness following from disjoint substitution commutativity (a consequence of GLP's single-occurrence invariant). We further prove that madGLP is grassroots. Both dGLP and madGLP serve as formal specifications for an AI-driven implementation discipline (math → informal spec → Dart) employed and described here: from dGLP, AI (Claude) developed a workstation-based GLP implementation in Dart, and from madGLP it is developing a smartphone-based multiagent one.

Keywords: concurrent logic programming · multiagent systems · transition systems · operational semantics · peer-to-peer · grassroots platforms

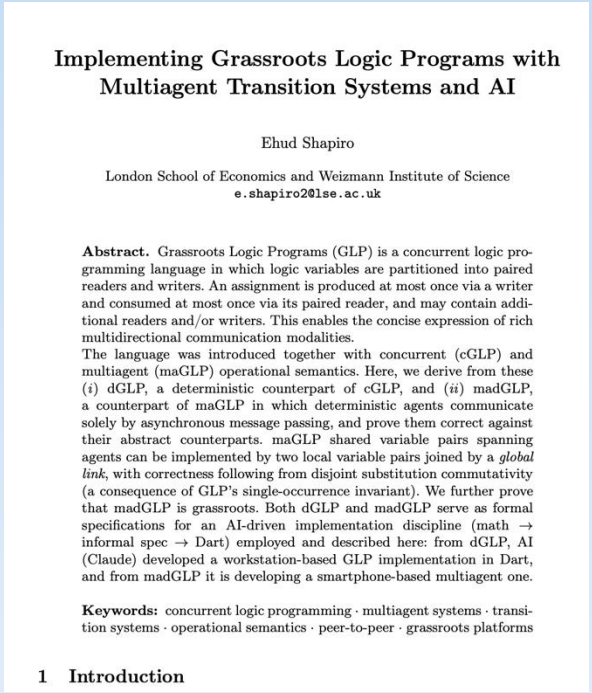
1 Introduction



```
agent(Id, [msg(person, Id1, introduce(P, Q))|UserIn], NetIn, Outs) :-
    Id? =?= Id1?, ground(P?), ground(Q?), ~(P? =?= Q?), new_channel(PQCh, QPCh) |
    send_friend(P?, msg(Id?, P?, intro(Q?, QPCh?)), Outs?, Outs1),
    send_friend(Q?, msg(Id?, Q?, intro(P?, PQCh?)), Outs1?, Outs2),
    agent(Id?, UserIn?, NetIn?, Outs2?).

%% User accepts intro — send ack, spawn peer-wait, inject result
agent(Id, [msg(person, Id1, accept_intro(Other, channel(Ch)))|UserIn], NetIn, Outs) :-
    Id? =?= Id1?, ground(Other?),
    send(ack(Id?), Ch?, Ch1) |
    intro_await_peer(Other?, Ch1?, Result),
    inject_intro_result(Result?, UserIn?, UserIn1),
    agent(Id?, UserIn1?, NetIn?, Outs?).

%% User rejects intro — send nack on channel, close it
agent(Id, [msg(person, Id1, reject_intro(_, channel(ch(_, [nack|[]])))|UserIn], NetIn, Outs) :-
    Id? =?= Id1? |
    agent(Id?, UserIn?, NetIn?, Outs?).

%% Intro result: peer acked — idempotent befriend commit (shared lib).
agent(Id, [intro_result(Other, Ch)|UserIn], NetIn, Outs) :-
    ground(Id?), ground(Other?) |
    befriend_commit(Id?, Other?, Ch?, Outs?, Outs1, NetIn?, NetIn1),
    agent(Id?, UserIn?, NetIn1?, Outs1?).

%% Intro result: peer nacked — notify rejected
agent(Id, [intro_rejected(Other)|UserIn], NetIn, Outs) :-
    ground(Id?) |
    send_user(msg(agent, person, rejected(Other?)), Outs?, Outs1),
    agent(Id?, UserIn?, NetIn?, Outs1?).

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

Human Scientific paper AI Program

The Human–Science–AI methodology



Implementing Grassroots Logic Programs with Multiagent Transition Systems and AI

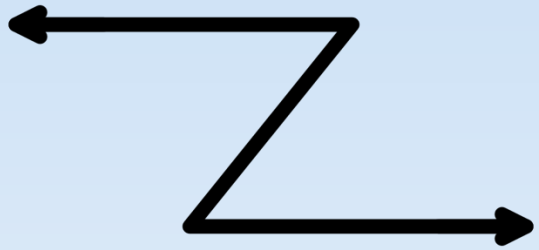
Ehud Shapiro

London School of Economics and Weizmann Institute of Science
e.shapiro2@lse.ac.uk

Abstract. Grassroots Logic Programs (GLP) is a concurrent logic programming language in which logic variables are partitioned into paired readers and writers. An assignment is produced at most once via a writer and consumed at most once via its paired reader, and may contain additional readers and/or writers. This enables the concise expression of rich multidirectional communication modalities.
The language was introduced together with concurrent (cGLP) and multiagent (maGLP) operational semantics. Here, we derive from these (*i*) dGLP, a deterministic counterpart of cGLP, and (*ii*) madGLP, a counterpart of maGLP in which deterministic agents communicate solely by asynchronous message passing, and prove them correct against their abstract counterparts. maGLP shared variable pairs spanning agents can be implemented by two local variable pairs joined by a *global link*, with correctness following from disjoint substitution commutativity (a consequence of GLP's single-occurrence invariant). We further prove that madGLP is grassroots. Both dGLP and madGLP serve as formal specifications for an AI-driven implementation discipline (math → informal spec → Dart) employed and described here: from dGLP, AI (Claude) developed a workstation-based GLP implementation in Dart, and from madGLP it is developing a smartphone-based multiagent one.

Keywords: concurrent logic programming · multiagent systems · transition systems · operational semantics · peer-to-peer · grassroots platforms

1 Introduction



```
agent(Id, [msg(person, Id1, introduce(P, Q))|UserIn], NetIn, Outs) :-
    Id? =?= Id1?, ground(P?), ground(Q?), ~(P? =?= Q?), new_channel(PQCh, QPCh) |
    send_friend(P?, msg(Id?, P?, intro(Q?, QPCh?)), Outs?, Outs1),
    send_friend(Q?, msg(Id?, Q?, intro(P?, PQCh?)), Outs1?, Outs2),
    agent(Id?, UserIn?, NetIn?, Outs2?).

%% User accepts intro — send ack, spawn peer-wait, inject result
agent(Id, [msg(person, Id1, accept_intro(Other, channel(Ch)))|UserIn], NetIn, Outs) :-
    Id? =?= Id1?, ground(Other?),
    send(ack(Id?), Ch?, Ch1) |
    intro_await_peer(Other?, Ch1?, Result),
    inject_intro_result(Result?, UserIn?, UserIn1),
    agent(Id?, UserIn1?, NetIn?, Outs?).

%% User rejects intro — send nack on channel, close it
agent(Id, [msg(person, Id1, reject_intro(_, channel(ch(_, [nack|[]])))|UserIn], NetIn, Outs) :-
    Id? =?= Id1? |
    agent(Id?, UserIn?, NetIn?, Outs?).

%% Intro result: peer acked — idempotent befriend commit (shared lib).
agent(Id, [intro_result(Other, Ch)|UserIn], NetIn, Outs) :-
    ground(Id?), ground(Other?) |
    befriend_commit(Id?, Other?, Ch?, Outs?, Outs1, NetIn?, NetIn1),
    agent(Id?, UserIn?, NetIn1?, Outs1?).

%% Intro result: peer nacked — notify rejected
agent(Id, [intro_rejected(Other)|UserIn], NetIn, Outs) :-
    ground(Id?) |
    send_user(msg(agent, person, rejected(Other?)), Outs?, Outs1),
    agent(Id?, UserIn?, NetIn?, Outs1?).

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

Problem: Paper-to-code could be a huge jump

Solution: Divide-and-conquer through ***abstraction cascades***

Abstraction cascade programming

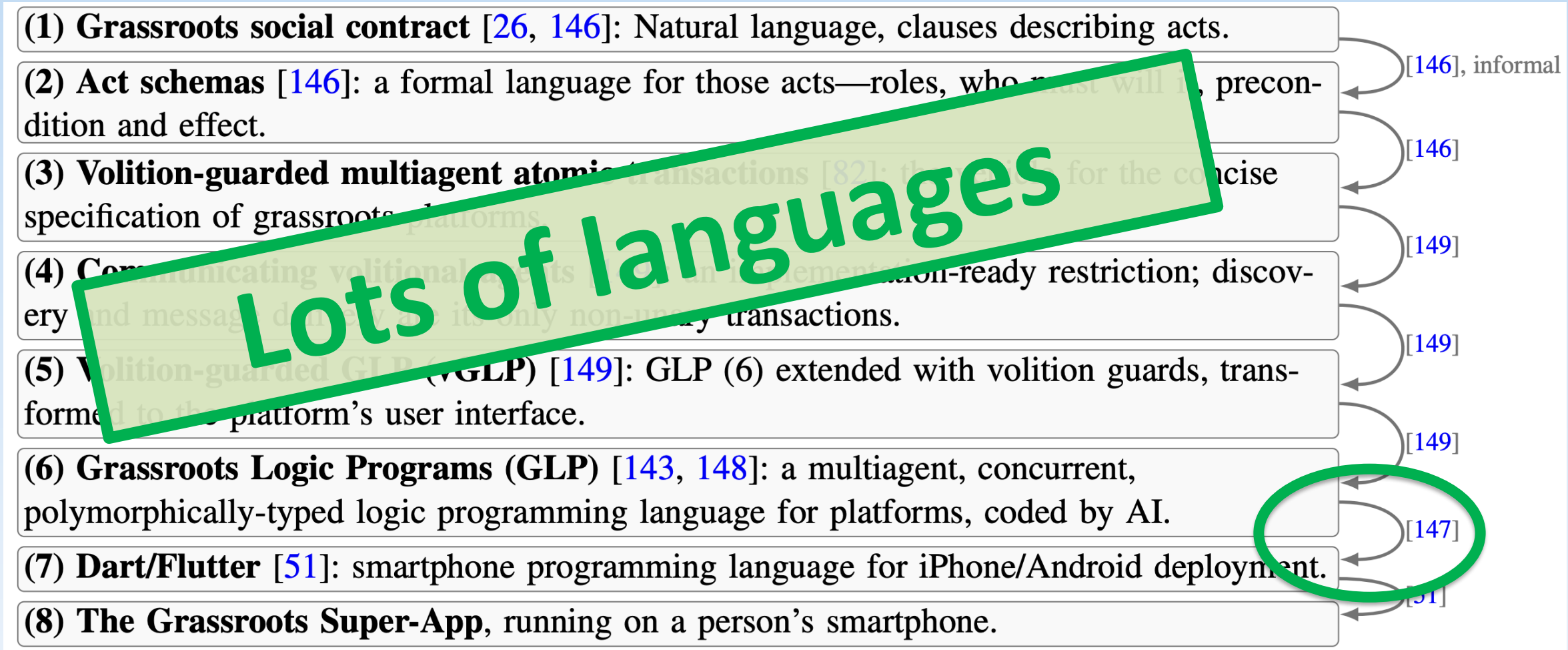


Figure 1: The abstraction cascade as currently envisioned, from a grassroots social contract to a running app. Each arrow represents an implementation with the reference defining it.

Implementing Grassroots Logic Programs

1. What do we want to build?
2. Why new languages?
3. **What is Grassroots Logic Programs (GLP)?**
4. Why is this language good for the task?
5. How is it being implemented (with AI)?

All covered by papers on arXiv

GLP Syntax

GLP syntax extends LP with:

1. **Writers and Readers:** each variable X (the writer) is paired with a reader X?, which receives the value assigned to X
2. **Single Occurrence (SO):** a variable occurs at most once in a goal or clause
3. **Single Reader Single Writer (SRSW):** a writer occurs in a clause iff its paired reader does, which together with SO implies SRSW.

GLP Operational Semantics

As in LP, a **GLP computation is deduction**. Also:

1. **Concurrency:** AND-parallelism
2. **Communication:** Instantiation is a message from **the single writer to the single reader** (and may carry further readers and writers)
3. **Synchronization:** Clauses tried in order, first successful reduction chosen. A reduction **suspends** if “unification” (in fact term matching) instantiates a reader

Fair Merge

```
Stream(X) ::= [] ;[X|Stream].

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

Goal: `merge([1,2,3|Xs?],[a,b|Ys?],Zs)`

The first clause swaps the inputs in the recursive call, ensuring fairness when both streams are available

GLP papers on arXiv

1. The language

12. arXiv:2510.15747 [pdf, ps, other] cs.PL cs.CR cs.DC cs.LO cs.MA

GLP: A Grassroots, Multiagent, Concurrent, Logic Programming Language for AI (Full Version)

2. Implementation

8. arXiv:2602.06934 [pdf, ps, other] cs.PL cs.AI cs.DC cs.LO cs.MA

Implementing Grassroots Logic Programs with Multiagent Transition Systems and AI (Full Version)

3. Type system

9. arXiv:2601.17957 [pdf, ps, other] cs.PL cs.DC cs.FL cs.LO cs.MA

Moded Types for Grassroots Logic Programs, by AI, for AI (Full Version)

4. UI

1. arXiv:2607.14138 [pdf, ps, other] cs.PL cs.AI cs.DC cs.HC cs.MA

Volition Elicitation: Operational Semantics for People and Their Machines

Implementing Grassroots Logic Programs

1. What do we want to build?
2. Why new languages?
3. What is Grassroots Logic Programs (GLP)?
4. **Why is this language good for the task?**
5. How is it being implemented (with AI)?

All covered by papers on arXiv

Friend-Mediated Introduction

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

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

Bob creates a channel pair and sends one half to each of Alice and Charlie; on mutual acceptance a direct connection arises — no cold-call

Friend-Mediated Introduction - Typed

```
Channel ::= ch(Stream?, Stream).

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

```
Name      ::= '_user' ; alice ; bob ; carol.
Op        ::= introduce(Name, Name) ; intro(Name, Channel).
Msg       ::= msg(Name, Name, Op).
MsgStream ::= [] ; [Msg | MsgStream].
Friends   ::= [] ; [(Name, Channel) | Friends].

procedure agent(Name?, MsgStream?, MsgStream?, Friends?).
```

Implementing Grassroots Logic Programs

1. What do we want to build?
2. Why new languages?
3. What is Grassroots Logic Programs (GLP)?
4. Why is this language good for the task?
5. **How is it being implemented (with AI)?**

All covered by papers on arXiv

Workstation Implementation of GLP with AI

1. **cGLP (paper):** standard nondeterministic interleaving-based operational semantics for GLP

Definition 3.11 (cGLP Transition System [33]). *Given a GLP program P, an **asynchronous resolvent** over P is a pair (G, σ) where $G \in \hat{\mathcal{G}}(P)$ and σ is a readers substitution.*

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

1. *$\mathcal{C}$ is the set of all asynchronous resolvents over P*
2. *$c_0 = (G_0, \emptyset)$*
3. *$\mathcal{T}$ is the set of all transitions $(G, \sigma) \rightarrow (G', \sigma')$ satisfying either:*
   (a) ***Reduce:** there exists a unit goal $A \in G$ such that $C \in P$ is the first clause for which the GLP reduction of A with C succeeds with result $(B, \hat{\sigma})$, $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$, and $\sigma' = \sigma \circ \hat{\sigma}$?.*
   (b) ***Communicate:** $\{X? := T\} \in \sigma$, $X? \in G$, $G' = G\{X? := T\}$, and $\sigma' = \sigma$.*
4. *$\sim$, the **cGLP transition equivalence**, relates $t_1 \sim t_2$ iff either both are Reduce transitions reducing the same goal (by identity, Definition 3.10) with the same clause, or both are Communicate transitions applying the counterpart of the same writer assignment to the same goal (by identity)*

Workstation Implementation of GLP with AI

1. **cGLP (paper):** standard nondeterministic interleaving-based operational semantics for GLP
2. **dGLP (paper):** Deterministic single-agent operational semantics for GLP

Workstation Implementation of GLP with AI

1. cG ving-
   ba
2. dC onal
   se

Definition 3.24 (dGLP Transition System). *The transition system* $dGLP(P) = (\mathcal{C}, c_0, \mathcal{T}, \mathrm{id})$ *over GLP program* P *and initial goal* G_0 *satisfying SO is defined by:*

1. $\mathcal{C}$ *is the set of all dGLP configurations over* P
2. $c_0 = (G_0, \emptyset, \emptyset)$
3. $\mathcal{T}$ *is the set of all transitions* $(Q, S, F) \to (Q', S', F')$ *where* $Q = A \cdot Q_r$ *(with* $\cdot$ *denoting the queue with head element* A *and remainder* Q_r*) and exactly one of the following holds:*
   (a) ***Reduce:*** *There exists a first clause* $C \in P$ *for which the GLP reduction of* A *with* C *(Definition 3.11) succeeds with* $(B, \hat{\sigma})$*. Let* $R = reactivate(S, \hat{\sigma}?)$*. Then:*
   $$Q' = (Q_r \cdot B \cdot R)\hat{\sigma}\hat{\sigma}?, \quad S' = S \setminus \{(G, W) : G \in R\}, \quad F' = F$$
   (b) ***Suspend:*** *No clause succeeds, but the GLP reduction of* A *with at least one clause* $C \in P$ *suspends. Let* $W = \bigcup_{C \in P} W_C$ *where* W_C *is the suspension set for clause* C*. Then:*
   $$Q' = Q_r, \quad S' = S \cup \{(A, W)\}, \quad F' = F$$
   (c) ***Fail:*** *No clause succeeds and no clause suspends (all clauses fail outright). Then:*
   $$Q' = Q_r, \quad S' = S, \quad F' = F \cup \{A\}$$
4. $\sim$ *is the identity relation: dGLP is deterministic, so each transition forms its own class*

Workstation Implementation of GLP with AI

1. **cGLP (paper):** standard nondeterministic interleaving-based operational semantics for GLP
2. **dGLP (paper):** Deterministic single-agent operational semantics for GLP
3. **Dart implementation (code):**
    1. GLP Virtual Machine, written in Dart (based on the FCP VM written in C in the 1980's by Avshalom Houri z"l)
    2. GLP to VM bytecode compiler, written in Dart

Smartphone Implementation of GLP with AI

1. **maGLP (paper):** multiagent operational semantics for GLP
2. **madGLP (paper):** agent-deterministic multiagent operational semantics for GLP
3. **Dart implementation (code):**
    1. GLP VM extended with shared logic variables and communication
    2. Same GLP to VM bytecode compiler, written in Dart

Smartphone Implementation of GLP with AI

1. **maGLP (paper):** multiagent operational semantics for GLP

Smartphone Implementation of GLP with AI

1. maGL GLP

Definition 5.1 (Multiagent GLP [52]). *Given agents $P \subset \Pi$ and GLP program M, the* ***maGLP transition system*** *over P and M is the transactions-based multiagent transition system (Definition 4.6) over P, local states being asynchronous resolvents (G_p, σ_p) over M, initial local state $s_0 = (\{\texttt{agent(ch(\_?,\_),ch(\_?,\_))}\}, \emptyset)$, the following transactions $c \to c'$, and the equivalence $\sim$ given below:*

1. ***Reduce*** *p: A unary transaction with participant p where $c_p \to c'_p$ is a cGLP Reduce transition (Definition 3.14).*
2. ***Communicate*** *p* ***to*** *q: A transaction with participants $p, q \in P$ where $\{X? := T\} \in \sigma_p$, $X?$ occurs in G_q, $c'_p = (G_p, \sigma'_p)$, $\sigma'_p = \sigma_p \setminus \{X? := T\}$, and $c'_q = (G_q\{X? := T\}, \sigma_q)$.*
3. ***Cold-call*** *p* ***to*** *q: A binary transaction with participants $p \neq q \in P$ where the network output stream in c_p has a new message $\texttt{msg}(q, X)$, c'_p is the result of advancing the network output stream in c_p, and c'_q is the result of adding $\texttt{msg}(q, X)$ to the network input stream in c_q.*
4. *$\sim$, the* ***maGLP transaction equivalence*** *on the transactions above, relates $t_1 \sim t_2$ iff both are Reduce transactions at the same agent p reducing the same goal (by identity, Definition 3.13) with the same clause, both are Communicate transactions from the same p to the same q applying the counterpart of the same writer assignment to the same goal (by identity), or both are Cold-call transactions from the same p to the same q delivering the same message.*

Smartphone Implementation of GLP with AI

1. **maGLP (paper):** multiagent operational semantics for GLP
2. **madGLP (paper):** agent-deterministic multiagent operational semantics for GLP

Smartphone Implementation of GLP with AI

1. maG
2. mac
   ope

Definition 6.17 (madGLP Transition System). *The* madGLP transition system *over agents $P \subset \Pi$ and a GLP program is the multiagent transition system madGLP $= (C, c_0, T, \sim)$ where C is the set of madGLP configurations, c_0 is the initial configuration with empty resolvents except for the initial* **agent** *goal and index-0 serializer entry, T is the set of Reduce, Send, and Receive transactions defined below, and two transitions are $\sim$-equivalent when they perform the same Reduce (same agent, goal by identity, clause), the same Send (same agent, queued message), or the same Receive (same agent, incoming message).*

Entries in the global writers table W_p route incoming assignments to local writers: (X, q) at index i binds X to assignments arriving via $\_w(p,i)$ from q, and (X_q, p, i) binds X_q to assignments arriving via $\_r(p,i)$ from p. Index 0 holds the serializer entry $(N_p, *)$, which is updated (not removed) on each cold-call so that multiple senders can append to the network input stream.

Definition 6.18 (madGLP Reduce Transaction). *The* unary *Reduce transaction for agent p where $A_p = A \cdot A_r$ transitions $s_p \to s'_p$ with exactly one of the following holding:*

1. ***Reduce:*** *The GLP reduction of A with the first applicable clause C succeeds with $(B, \hat{\sigma})$. Let $R =$ reactivate$(S_p, \hat{\sigma}?)$. Then:*
$$A'_p = (A_r \cdot B \cdot R)\hat{\sigma}\hat{\sigma}?, \quad S'_p = S_p \setminus \{(G, W) : G \in R\}, \quad F'_p = F_p$$
2. ***Suspend:*** *No clause succeeds, but the GLP reduction of A with at least one clause suspends. Let $W = \bigcup_C W_C$ where W_C is the suspension set for clause C. Then:*
$$A'_p = A_r, \quad S'_p = S_p \cup \{(A, W)\}, \quad F'_p = F_p$$
3. ***Fail:*** *No clause succeeds and no clause suspends. Then:*
$$A'_p = A_r, \quad S'_p = S_p, \quad F'_p = F_p \cup \{A\}$$

Definition 6.19 (madGLP Send Transaction). *The* unary *Send transaction for agent p is enabled when $(m, q) \in M_p$ is unsent and not held (§6.7). It places m in the communication channel to q; a reader-name value remains in M_p, marked* pending, *until acknowledged (§6.5), and every other message is removed from M_p when sent.*

Definition 6.20 (madGLP Receive Transaction). *The* unary *Receive transaction processes a message m from the communication channel, with exactly one of the following holding:*

1. ***Serializer*** *($m = (\_w(q,0) := [T^{\uparrow} \mid \_w(q,0)])$): The receiving agent is q, with entry $(N_q, *)$ at index 0 in W_q. Let N'_q be a fresh local writer, $\hat{\sigma} = \{N_q := [T^{\downarrow}_q \mid N'_q?]\}$, and $R =$ reactivate$(S_q, \hat{\sigma}?)$. Then:*
$$A'_q = (A_q \cdot R)\hat{\sigma}\hat{\sigma}?, \quad S'_q = S_q \setminus \{(G, W) : G \in R\}, \quad F'_q = F_q$$
The entry at index 0 in W'_q becomes $(N'_q, *)$.
2. ***Writer*** *($m = (\_w(p,i) := T^{\uparrow})$, $i > 0$): The receiving agent is p, with entry (X, q) at index i in W_p; the sender need not be q, the name having*

been forwarded (Definition 5.8); an assignment from an agent other than q is held and applied on authorisation (§6.7). Let $\hat{\sigma} = \{X := T^{\downarrow}_p\}$ and $R =$ reactivate$(S_p, \hat{\sigma}?)$. Then:
$$A'_p = (A_p \cdot R)\hat{\sigma}\hat{\sigma}?, \quad S'_p = S_p \setminus \{(G, W) : G \in R\}, \quad F'_p = F_p$$
The entry at index i is removed from W'_p.

3. ***Reader*** *($m = (\_r(p,i) := T^{\uparrow})$): The receiving agent q is the one that localized $\_r(p,i)$, with entry (X_q, p, i) in W_q. Let $\hat{\sigma} = \{X_q := T^{\downarrow}_q\}$ and $R =$ reactivate$(S_q, \hat{\sigma}?)$. Then:*
$$A'_q = (A_q \cdot R)\hat{\sigma}\hat{\sigma}?, \quad S'_q = S_q \setminus \{(G, W) : G \in R\}, \quad F'_q = F_q$$
The entry matching (p,i) is removed from W'_q, and $(\mathbf{ack}(\_r(p,i)), s)$ is added to M'_q, where s is the agent that sent m.
4. ***Request*** *($m = \mathbf{req}(\_r(p,i))$): The receiving agent is p, the anchor. The request is held (§6.7); on authorisation p records the requester as the holder of the link $\_r(p,i)$, and if a pending value addressed by $\_r(p,i)$ is in M_p, it is re-addressed to the requester and becomes unsent. A request for a closed link is dropped.*
5. ***Acknowledgement*** *($m = \mathbf{ack}(\_r(p,i))$): The receiving agent is the sender of the acknowledged value; let s be the acknowledging agent and V the pending value addressed by $\_r(p,i)$. The receiver records s as the holder of every link whose reader name anchored at the receiver occurs in V, re-addressing any pending value of such a link to s; it then removes V from its M. An acknowledgement for a closed link is dropped.*

Smartphone Implementation of GLP with AI

1. **maGLP (paper):** multiagent operational semantics for GLP
2. **madGLP (paper):** agent-deterministic multiagent operational semantics for GLP
3. **Dart implementation (code):**
    1. GLP VM extended with shared logic variables and communication
    2. Same GLP to VM bytecode compiler, written in Dart

Abstraction cascade programming

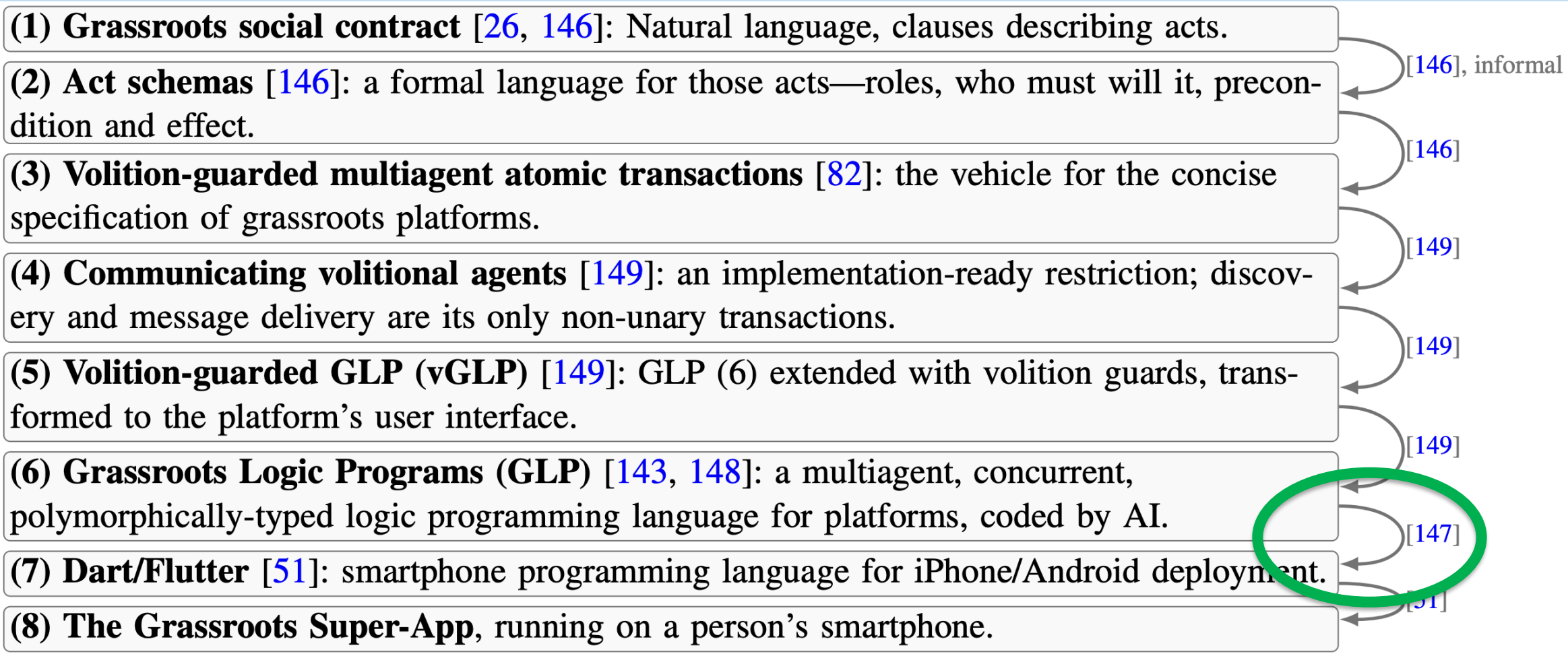


Figure 1: The abstraction cascade as currently envisioned, from a grassroots social contract to a running app. Each arrow represents an implementation with the reference defining it.

Defining Grassroots Platforms

Comparatively: *How many agents need to be removed to make communication impossible?*

1. ***Centralised*** – 1 (the server, e.g. Facebook)
2. ***Decentralised*** – finite >1 (hardwired boot nodes, e.g. Bitcoin)
3. ***Federated*** – infinite, but not all (all servers, e.g. Mastodon)
4. ***Grassroots*** – all but one (any two agents may communicate)