Volition Elicitation: Operational Semantics for People and Their Machines (Full Version)

IDIT KEIDAR, Technion – Israel Institute of Technology, Israel
EHUD SHAPIRO, London School of Economics and Weizmann Institute of Science, UK and Israel

The most prevalent distributed systems today include people and their personal machines (smartphones), and computations in them are driven by people's volitions. Volition-guarded multiagent atomic transactions were proposed as an abstract specification language for such systems, in which a transaction has a precondition on the machine states of its participants and is guarded by the volitions of some of them.

Here we develop an abstraction cascade that derives smartphone apps from such specifications and prove it correct down to the base programming language, GLP. The cascade consists of: (*i*) communicating volitional agents (CVA), which implement every local volition-guarded specification, correctly, and completely for its strongly fair runs, by two-phase commit coordinated by the initiating agent; (*ii*) volition-guarded GLP (vGLP), an extension of GLP with volitional procedures, whose goals put an interactively-typed question to the person, into which that implementation compiles; and (*iii*) GLP, which implements vGLP by eliciting the person's answers through the widgets of their interactive types, correctly, and completely for complete widgets. We demonstrate the approach on three grassroots platforms — a social graph, a social network, and currencies: a compiler carries each contract down the cascade to a GLP program that runs with scripted persons, taking the words of its app from the contract too, and the apps of the three, so compiled, run on an iPhone simulator.

CCS Concepts: • **Theory of computation** → Operational semantics; • **Software and its engineering** → Constraint and logic languages; Concurrent programming languages; Distributed programming languages; • **Human-centered computing** → Human computer interaction (HCI); • **Computer systems organization** → Peer-to-peer architectures.

Additional Key Words and Phrases: Volition Elicitation, Operational Semantics, Concurrent Logic Programming, GLP, Volitional Transactions, Grassroots Platforms, User Interface

1 Introduction

Volition-guarded multiagent atomic transactions. The most prevalent distributed systems today include people and their personal machines (smartphones). In such systems, computations are driven by people's volitions: a payment when a person wishes to pay someone, befriending when two people wish to become friends, etc. Volition-Guarded Multiagent Atomic Transactions [28] were proposed as an abstract specification language for such systems. In them (*i*) each agent consists of a person and their machine; (*ii*) a multiagent atomic transaction has a precondition on the machine states of its participants and is guarded by the volitions of some of them. For example, the befriending of two people is guarded by both persons and unfriending by either; a payment is guarded by the payer, a swap by both parties. An illustrative example is child-safe online befriending [54]: it is guarded by the volitions of the two children and one parent of each, and its precondition is that the machines do not record the two children as friends and do record the two parents as friends (Appendix A.1).

Grassroots platforms. We focus on platforms that can have multiple, volitionally-forming instances that may coalesce by mutual consent, possibly forming a single global instance, referred to as *grassroots platforms* [28, 50, 52, 56]. Practically, a grassroots platform consists of people running a person-to-person smartphone app.

Example: minting and swapping grassroots coins. Grassroots coins [49] are units of debt that any person can issue (Section 7.3). The machine state of each agent p is its holdings c_p, a multiset of coins, and $¢_p^k$ is k coins issued by p. Two of the transactions of grassroots coins are:

(1) $\mathbf{Mint}_p(k)$: $c'_p := c_p \cup \mathfrak{c}^k_p$. Guarded by p.
(2) $\mathbf{Swap}_{p,q}(\mathfrak{c}^k_u, \mathfrak{c}^{k'}_v)$: $c'_p := c_p \setminus \mathfrak{c}^k_u \cup \mathfrak{c}^{k'}_v$ and $c'_q := c_q \setminus \mathfrak{c}^{k'}_v \cup \mathfrak{c}^k_u$, provided $u \neq v$ or $k \neq k'$, $\mathfrak{c}^k_u \subseteq c_p$ and $\mathfrak{c}^{k'}_v \subseteq c_q$. Guarded by $\{p, q\}$.

Minting changes the state of one agent, and takes place only if its person wills it. A swap changes the states of both parties at once, each giving coins and taking the other's, and takes place only if both persons will it. The currency agent of Section 5 implements both. A mint is a request of its person. A proposed swap is put to the other party as an *approval question*, which shows the proposal and has two buttons, yes and no.

Previous work. Previous work [28] presented volition-guarded specifications for the grassroots social graph and grassroots currencies [49], and contracts written as speech-act schemas for these two and for the grassroots social network [16, 51]. Volition-guarded transactions thus condition machine transitions on persons' volitions. Formally, such a transition can occur only if the will for it is part of the volitional state of the person, namely is in the person's head. Given today's technologies, the only way for a machine to know what's in a person's head is for the person to express it; this is the fundamental reason for a UI. One of the contributions of this paper is to rationalise the design of a UI that can elicit volitions from the person's head, when needed. Existing programming languages treat the person as a source of input — a stream of clicks [8, 10, 14, 25], an event [4, 20, 40, 60], the environment [23, 61, 63] — so input drives computation, but in none does a transaction wait on the wills of its guards, with an else-branch taken when a guard does not will it (Section 9).

The abstraction cascade. Here we develop an *abstraction cascade* that derives smartphone grassroots apps from such abstract specifications, as shown in Figure 1, and prove it correct down to the base programming language, GLP (Section 6.4). The passage from a legal text to a running platform goes through the cascade, from the legal text at the top [16] to an app on each person's own phone at the bottom. This paper supplies the arrows (3)→(6), each proved correct, starting from (3), volition-guarded multiagent atomic transactions [28]. Its layers (4)–(8) in Figure 1 are:

(4) Communicating volitional agents. Implementing a multiagent atomic transactions-based specification must go through unary transactions, carried out by each agent. In the case of such volition-guarded transactions, the corresponding implementation-ready model is Communicating Volitional Agents (CVA), defined in Section 3. Mathematically, CVA is but a restriction of the general model, in which the only non-unary transactions are agent discovery, agent introduction and message delivery. We prove that every CVA protocol is grassroots (Proposition 3.6). When implementing CVA on a smartphone, the basic requirement from the smartphone UI is one: to elicit, from the person operating the smartphone, the volitions the transactions require.

A volition-guarded transaction changes the machine states of its participants at once, while a communicating volitional agent changes only its own state and reaches another only by a message. Here we implement the first by the second, for every local volition-guarded specification (Section 3.2), correctly, and completely for its strongly fair runs. We use two-phase commit coordinated by the initiating agent [19]. Wound-wait [45] orders contending attempts, and an attempt that asks a person who does not will it closes by the else-branch of a volition-guarded if-then-else transaction [28] (Section 2.2), taken when the person does not will its then-branch. The protocol is free of deadlock and livelock. It is correct for runs in which an initiator knows its participants, no agent is kept busy without end, a transaction that can be taken in several ways can, from some point on, be taken in one of them, by the wills of the persons that way needs, and a machine that can propose or request a transaction again and again does so again and again. It is complete for

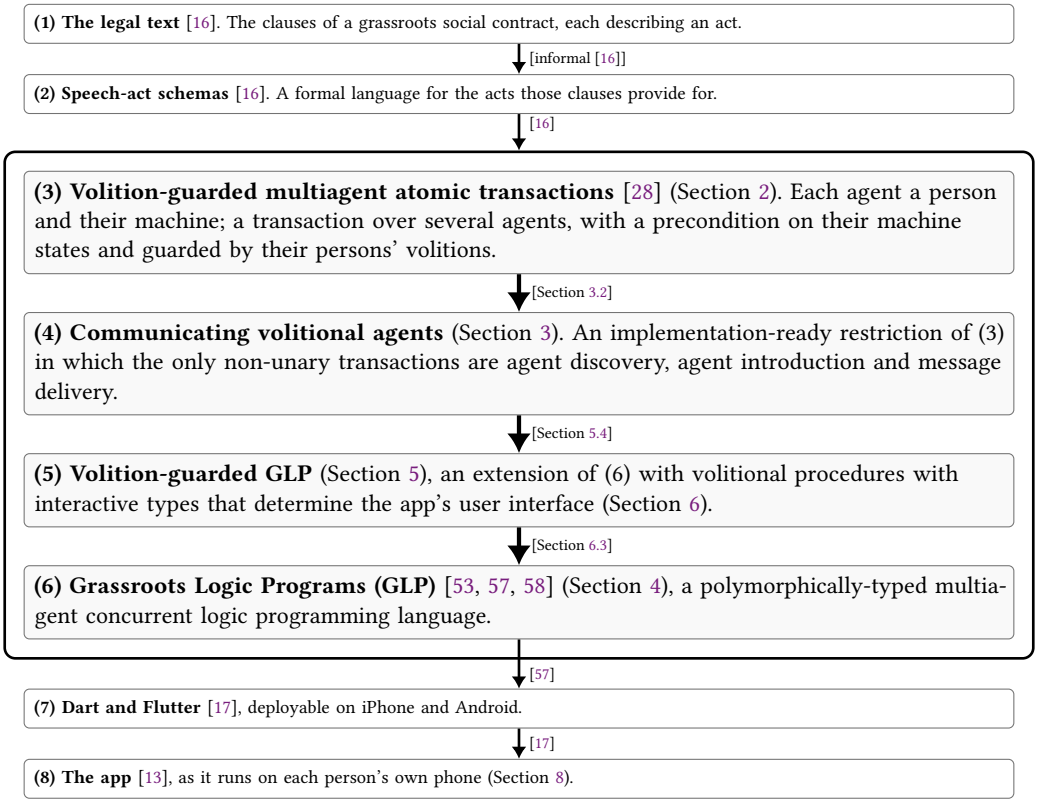


Fig. 1. The abstraction cascade, from a legal text to a running platform. The framed (3)–(6) are this paper's subject: (3) is [28], recalled in Section 2, (4) is Section 3, and the arrows (3)→(6) are this paper. Each arrow carries the reference in which its mapping is defined; the first, rendering a legal text as speech-act schemas, is the one step left to judgement.

the runs of the specification in which a transaction enabled again and again is taken again and again. The model is asynchronous and fault-free: it has no notion of time, and no agent crashes or misbehaves.

(5) Volition-guarded GLP. Volition-guarded GLP (vGLP) extends GLP with volitional procedures. A goal of a volitional procedure puts a question to the person, and a clause that reads the answer waits for it. A volitional procedure is declared `procedure (`T`)*`$p(T_1, \ldots, T_n)$: each goal $p(S_1, \ldots, S_n)$ of it receives an *interactive variable* of the *interactive type* T as an added argument, and each clause $(A)*p(S_1, \ldots, S_n)$ `:-` G `|` B matches it with its interactive term A. The interactive type says which positions the program writes, for the person to see, and which positions the person writes, as their answer.

Two fragments of the currency agent (Section 7) preview the language, `...` eliding arguments a clause carries through unchanged. The first is the agent, whose question is the stream of the person's requests, served by an ordinary procedure, with the clause by which the person mints coins. The second is a *responder* goal, spawned per arriving swap proposal, by which the person answers it:

```
Request ::= mint(Amount)
```

```
          ; swap(Friend, GiveCoin, GiveAmount,
                 WantCoin, WantAmount)
          ; pay(Friend, Amount)
          ; redeem(Friend, WantCoin).
procedure (Stream(Request)?)*agent(...).

%% Ask the person for their requests, once, and serve them.
(Reqs)*agent(Id, ..., Outs, Holdings, Serial) :-
    serve(Reqs?, Id?, ..., Outs?, Holdings?, Serial?).

%% Mint_p(k), guarded by p: c'_p = c_p + cent^k_p.
serve([mint(Amount)|Reqs], Id, ..., Outs, Holdings,
      Serial) :-
    ground(Id?), integer(Amount?), Amount? >= 1 |
    create_coins(Id?, Amount?, Serial?, New),
    Serial1 := Serial? + Amount?,
    append_coins(Holdings?, New?, Holdings1),
    show(minted(Amount?), Outs?, Outs1),
    after_change(Reqs?, Id?, ..., Outs1?, Holdings1?,
                 Serial1?).

%% Show the person the offer, and pass their answer on.
Offer ::= offer(From, Want, Offered).
YesNo ::= yes ; no.
Approval ::= approval(Offer, YesNo?).
procedure (Approval)*respond_swap(Offer?, SwapResponse,
    Stream(UserInMsg)).
(approval(offer(From?, Want?, Offered?), Answer))*
  respond_swap(offer(From, Want, Offered), Resp?,
               Ans?) :-
    ground(From?), ground(Want?), ground(Offered?) |
    decide_swap(Answer?, From?, Want?, Offered?, Resp,
                Ans).
```

The agent asks its question once, and the person answers it one request at a time. The mint clause waits for a `mint` request at the head of the stream, which supplies `Amount`. The request form stays on screen. The responder writes into its approval question the friend who proposes the swap, the coins they want and the coins they offer, and passes the person's yes or no to `decide_swap` (Section 5). The question has two buttons, and no is the decline.

The operational semantics of volition-guarded GLP. We then define the operational semantics of vGLP. The state of each agent has two parts: the questions its person holds, and the goals its machine is running, called its resolvent. A goal of a volitional procedure is first asked: the goal receives one end of its interactive variable, and the person the other. A question reaches the person with the program's output in it written whole, and a clause that reads the person's part reduces only after the person has answered. An answer cannot be withdrawn. The machine cannot compel the person, so in vGLP an answer carries no liveness obligation. We prove that on programs with no volitional procedures the semantics coincides with GLP's (conservativity), and that a question is assigned only by its person's answer (volitional soundness).

We compile the canonical CVA platform of every effective local specification into vGLP, one goal per agent (Section 5.4), and prove the compilation correct and its protocol grassroots.

(6) Grassroots Logic Programs and (7) Dart. For AI to derive smartphone-based grassroots platforms from such abstract specifications, Grassroots Logic Programs (GLP) [53, 57, 58], a high-level, polymorphically-typed, multiagent, concurrent logic programming language, was designed and implemented in Dart/Flutter, which is deployable on most smartphones (iPhone/Android). So far, GLP implementations of these platforms were derived in the same way from (volition-free) multiagent atomic transactions [56].

However, specifying the role of people in such platforms is out-of-reach for this formalism, and hence the step of designing and implementing the app's UI was an ad hoc, "manual" process.

To rectify that, we need formal foundations for the implementation of volition-guarded transactions, not only for attaining mathematical elegance and completeness, but, more pragmatically, for AI to methodically develop the volition-eliciting UI for abstractly-specified grassroots platforms. Moreover, we wish for a single smartphone app, running the GLP engine, to support multiple, dynamically-loaded grassroots platforms written in GLP, and for that we cannot have an ad-hoc UI for each new platform.

As the semantics of vGLP requires the person's answer before a reduction that reads it, a complete implementation of vGLP in GLP must elicit it: finding out "what's in the person's head" is the rationale for the UI. We derive the UI from the semantics: an open question is presented as the widget of its interactive type, which shows what the program writes for the person and forms the person's input into a term of the type. Widgets are associated with types by declarations, scoped as type declarations are, and every interactive type has a default widget built from its constructors. We prove that a construct offers only answers of its question's type, and every such answer where its widget is complete (elicitation completeness). The implementation compiles vGLP onto GLP, and we prove the implementation of vGLP by GLP correct, and complete for complete widgets.

(8) The app. We demonstrate the approach on the grassroots social graph, grassroots social network, and grassroots currencies, starting from their contracts, written as speech-act schemas, and going down the cascade: the CVA implementation of each platform is its canonical platform, and its vGLP program is the compilation of that platform. A compiler takes a contract written as speech-act schemas [16], whose roles are party roles or listed roles, to the vGLP compilation of its canonical platform in one pass, producing neither the volition-guarded transactions the contract specifies nor the platform. It compiles the social graph and the social network, GSG and GSN [16], and the currencies, GC [49], and the compiled programs run on GLP with scripted persons. The apps of the three, and of the base contract GKK they extend, so compiled, run as mini-apps of the grassroots super-app [13] on an iPhone simulator (Section 8). Beside its clauses and its schemas a contract carries the words its app shows, which the compiler takes into the platform, so the app is compiled from the contract.

Paper outline. Section 2 recalls volition-guarded multiagent atomic transactions, volition-guarded if-then-else transactions, transition systems and implementations; grassroots protocols are recalled in Appendix A. Section 3 defines CVA, proves every CVA protocol volitionally grassroots, and implements every local volition-guarded specification by CVA, by two-phase commit coordinated by the initiating agent. Section 4 recalls GLP: its syntax, its reduction and its multiagent semantics. Section 5 defines vGLP as its extension and the operational semantics vmaGLP, with volitional soundness, then compiles the canonical CVA platform of every effective local specification into vGLP, and proves the compilation correct. Section 6 derives the elicitation constructs from interactive types, presents widgets and their declarations, proves elicitation completeness, defines the compilation of vGLP onto GLP, and proves the implementation of vGLP by GLP correct, and complete for

complete widgets; it ends by composing the implementations of the cascade, from an effective local specification to GLP. Section 7 presents the three platforms as the contracts the compiler compiles, whose CVA implementations are their canonical platforms and whose vGLP programs are the compilations of those platforms. Section 8 shows the apps of the three platforms and of GKK, compiled down the cascade, on an iPhone simulator. Section 9 reviews related work, and Section 10 concludes.

2 Volition-Guarded Multiagent Atomic Transactions

This section recalls the mathematical foundations on which the paper is based [28, 48]. Each agent consists of a *person* and a *machine* (e.g., smartphone) operated by the person. A *machine transaction* specifies what machines do; a *volition guard* specifies which persons must be willing for it to occur.

We assume a potentially infinite set of *agents* Π, but consider only finite subsets of it, so we assume any particular set of agents $P \subset \Pi$ to be nonempty and finite. We use $\subset$ to denote the strict subset relation and $\subseteq$ when equality is also possible. As standard, we use S^P to denote the set of all total functions from P to S, and if $c \in S^P$ we use c_p to denote the value of c at $p \in P$.

2.1 Volition-Guarded Transactions

Definition 2.1 (Machine State, Configuration, Transaction, Volition-Guarded Multiagent Atomic Transaction). Given an arbitrary set S of *machine states*, with a designated *initial state* $s_0 \in S$, and agents $Q \subset \Pi$, a *machine configuration* over Q is a member of S^Q, and a *machine transaction* over *participants* Q is a pair $c \longrightarrow c' \in (S^Q)^2$ such that $c \neq c'$. Given such a machine transaction t, a *volition-guarded multiagent atomic transaction* over t — henceforth, *volition-guarded transaction* — is a pair (t, Q') where $Q' \subseteq Q$ are its *guards*.

Here a guard is a person: the guards of a transaction are the participants who must will it, and a transaction with no guards is *unguarded*. The condition on the machine states, that the participants be in the states c of $c \longrightarrow c'$, its *source*, is the transaction's *precondition*. Machine transactions are atomic and asynchronous [48]—they can be carried out by their participants at any time, regardless of the states of non-participants. Participants include both active agents (whose state changes) and stationary agents (whose state is a precondition but does not change). Volition-guarded transactions are machine transactions that can be carried out only if their guards $Q' \subseteq Q$ are willing. When we say a transaction is "guarded by $\{p, q\}$," both must be willing; when we say it is "guarded by either p or q," we mean there are two volition-guarded transactions over the same machine transaction, $(t, \{p\})$ and $(t, \{q\})$, so that either person's volition suffices. Distinct machine transactions can represent "the same action" in different configurations, and we identify them by an equivalence relation on machine transactions.

Definition 2.2 (Transaction Equivalence). Given a set of machine transactions R, a *transaction equivalence* is an equivalence relation $\sim$ on R such that $t \sim t'$ implies t and t' have the same participants, and the set $R/\sim$ of its classes is countable. We write $[t]$ for the equivalence class of t under $\sim$.

A person's volitional state is a set of equivalence classes of machine transactions they are willing their machine to participate in. The will names the equivalence class, not a label on a transition: one action is a set of transitions, one per configuration it may occur in, and the will is fulfilled by whichever of them is taken.

Definition 2.3 (Agent State and Configuration). Given agents P, states S with initial state s_0, a set of machine transactions T each over its own participants $Q \subseteq P$ and S, and equivalence $\sim$ on T, an *agent state* is a pair $(V, m) \in \mathcal{A} = (2^{T/\sim} \times S)$ where V is its *volitional state* and $m \in S$ its *machine*

state. The *initial agent state* is $(\emptyset, s_0)$. An *agent configuration* c over P, S, T, and $\sim$ is a member $c \in \mathcal{A}^P$ in which $c_p^v \subseteq (T/\!\sim)_p$ for every $p \in P$, where $(T/\!\sim)_p$ denotes the classes in $T/\!\sim$ in which p is a participant; we write c_p^v for the volitional state and c_p^m for the machine state of agent p in c.

Definition 2.4 (Volitional Transaction [28]). Given agents P, states S, machine transactions T over P and S, and equivalence $\sim$ on T:

(1) A *volition change of agent* $p \in P$ is a pair $c \to c'$ of agent configurations over $\{p\}$, S, T, and $\sim$ such that $c_p^v, {c'}_p^v \subseteq (T/\!\sim)_p$ and $c_p^v \neq {c'}_p^v$, and $c_p^m = {c'}_p^m$.
(2) A *volitional machine transaction* induced by a volition-guarded transaction (t, Q'), for some $t = (d \to d') \in T$ over $Q' \subseteq Q \subseteq P$, is a pair $c \to c'$ where $c \neq c'$ are agent configurations over P, S, T, and $\sim$ such that $[t] \in c_q^v$ for every $q \in Q'$; $c_p^m = d_p$ and ${c'}_p^m = d'_p$ for every $p \in Q$; $c_p^m = {c'}_p^m$ for every $p \in P \setminus Q$; and ${c'}_p^v = c_p^v \setminus \{[t]\}$ for every $p \in P$.
(3) A *volitional transaction* is a volition change or a volitional machine transaction.

When a volitional machine transaction induced by (t, Q') is taken, the class $[t]$ is removed from every agent's volitional state: the will is fulfilled by any equivalent transaction. A person may independently change their volitional state via volition changes, which may add or remove classes; beyond these, the framework removes a class from c_p^v only upon fulfilment.

2.2 Volition-Guarded If-Then-Else Transactions

In the transactions above, a person's not willing a transaction has no consequence: no transaction is enabled by the absence of a will. Communicating volitional agents (Section 3) need such a consequence. Without it, the implementation of the transactions by communicating volitional agents (Section 3.2) could not close an attempt whose guard does not will it, nor release the parts its participants hold prepared for it. This is the *volition-guarded if-then-else transaction* of [28]: in machine state s, if p wills it, s' is taken, else s''. Its else-branch is taken when the person does not will the then-branch. In this paper the else-branch is used only within CVA.

Definition 2.5 (Volition-Guarded If-Then-Else Transaction, Then-Branch, Else-Branch [28]). Given machine states S, a *volition-guarded if-then-else transaction* over a single participant $p \in \Pi$ is a triple $s \to s'\,;\, s''$ *guarded by* p, with $s, s', s'' \in S$, $s \neq s'$ and $s \neq s''$; the unary machine transaction $s \to s'$ is its *then-branch* and $s \to s''$ its *else-branch*.

Definition 2.6 (Else Transaction [28]). Given agents P, states S, machine transactions T over P and S, and equivalence $\sim$ on T, an *else transaction* induced by a volition-guarded if-then-else transaction $s \to s'\,;\, s''$ guarded by $p \in P$, with $s \to s' \in T$, is a pair $c \to c'$ of agent configurations over P, S, T, and $\sim$ such that $[s \to s'] \notin c_p^v$; $c_p^m = s$ and ${c'}_p^m = s''$; $c_q^m = {c'}_q^m$ for every $q \in P \setminus \{p\}$; and ${c'}_q^v = c_q^v$ for every $q \in P$.

The then-branch, taken when p wills it, is the unary volition-guarded transaction $(s \to s', \{p\})$ and induces volitional machine transactions per Definition 2.4; a unary volition-guarded transaction is the special case with no else-branch, and an unguarded one is $(s \to s', \emptyset)$. The else transaction is taken when p does not will the then-branch; it fulfils no will and removes no class. The two are placed in one class below, so that liveness obliges the machine to take one branch or the other, with no time bound. Only a unary guarded transaction has an else-branch: an unguarded one requires no will, and a binary one would let an agent of another group, who never wills it, trigger it (Definition A.3, Proposition A.6). Where a set of transactions may include volition-guarded if-then-else transactions, as a CVA platform's may (Section 3), *volition-guarded transaction* covers them.

2.3 Transition Systems

Definition 2.7 (Transition System, Computation, Run, Safe, Live, Correct [28, 48]). A *transition system* is a tuple $TS = (C, c_0, T, \sim)$ where C is an arbitrary set of *configurations*, $c_0 \in C$ a designated *initial configuration*, $T \subseteq C \times C$ a set of *transitions*, each a pair $c \rightarrow c'$ of non-identical configurations $c \neq c' \in C$, and $\sim$ a partial equivalence relation on T — an equivalence on a subset $dom(\sim) \subseteq T$; we write $[t]$ for the class of $t \in dom(\sim)$ under $\sim$.

A *computation* is a (nonempty, finite or infinite) sequence of configurations $c_1, c_2, \ldots$; it is a *run* if $c_1 = c_0$, and *safe* if $c_i \rightarrow c_{i+1} \in T$ for every two consecutive configurations. We write $c \xrightarrow{*} c'$ for the existence of a safe computation from c to c' (empty if $c = c'$). A class $[t]$ is *enabled* in c if $c \rightarrow c' \in [t]$ for some c'. A run is *live* if no class $[t]$, $t \in dom(\sim)$, is enabled in every configuration of some suffix in which no member of $[t]$ occurs, and *correct* if it is safe and live.

Liveness is weak fairness, per class. A partial equivalence, rather than a total one, is used so that some transitions may carry no liveness requirement: only transitions in the domain of $\sim$ form classes and thereby incur a liveness obligation, while transitions outside the domain — belonging to no class — may occur in a correct run but are never required to.

Definition 2.8 (Multiagent Transition System [48]). Given agents $P \subset \Pi$ and an arbitrary set S of *local states* with a designated *initial local state* $s_0 \in S$, a *multiagent transition system* over P and S is a transition system $TS = (C, c_0, T, \sim)$ with *configurations* $C := S^P$, *initial configuration* $c_0 := \{s_0\}^P$, *transitions* $T \subseteq C^2$ a set of transactions over P and S, and $\sim$ a partial equivalence on T.

Rather than specifying a multiagent transition system directly, we specify it via transactions, each over its own participants; the closure operator induces from them transitions over a fixed P, in which non-participants are stationary.

Definition 2.9 (Transaction Closure [28]). Let $P \subset \Pi$, S a set of local states, and $C := S^P$. For any transition or transaction $t = c \rightarrow c'$, we write $t_q := c_q \rightarrow c'_q$ and say p is *stationary* in t if $c_p = c'_p$. For a transaction $t = (c \rightarrow c')$ over S with participants Q, the *P-closure of t*, $t{\uparrow}P$, is the set of transitions over P and S defined by:

$$t{\uparrow}P := \begin{cases} \{t' \in C^2 : \forall q \in Q.(t_q = t'_q) \wedge \forall p \in P \setminus Q.(p \text{ is stationary in } t')\} & \text{if } Q \subseteq P \\ \emptyset & \text{otherwise} \end{cases}$$

If R is a set of transactions, each $t \in R$ over some Q and S, then the *P-closure of R* is $R{\uparrow}P := \bigcup_{t \in R} t{\uparrow}P$. Given a relation $\sim$ on R, its *P-closure* $\sim{\uparrow}P$ is the relation on $R{\uparrow}P$ with $\hat{t}$ $(\sim{\uparrow}P)$ $\hat{t}'$ iff $\hat{t} \in t{\uparrow}P$ and $\hat{t}' \in t'{\uparrow}P$ for some $t \sim t'$.

A transaction names only its participants. Its closure makes it transitions of the whole system over P: the participants move as the transaction says, and every other agent stays where it is. The closure and the equivalence do different jobs: the closure turns transactions into transitions, and the equivalence groups the transitions that are one act. If the P-closures of distinct transactions in R are disjoint, then each transition in $R{\uparrow}P$ is induced by one transaction, and $\sim{\uparrow}P$ relates two transitions exactly when their inducing transactions are related, so $\sim{\uparrow}P$ is a partial equivalence whenever $\sim$ is. Disjointness holds when every participant of every transaction changes state, but a participant need not: a participant whose state is part of the precondition and does not change, such as a voter, is allowed, provided no other transaction in R moves the same agents in the same way. For volition-guarded transactions it is required only of unguarded ones (Definition 2.11).

Definition 2.10 (Transactions-Based Multiagent Transition System, adapted from [28, 48]). Given agents $P \subset \Pi$, local states S with initial local state $s_0 \in S$, a set of transactions R, each $t \in R$

over some $Q \subseteq P$ and S, the P-closures of distinct transactions in R being disjoint, and a partial equivalence $\sim$ on R, the *transactions-based multiagent transition system* over P, S, R, and $\sim$ is the multiagent transition system $TS = (S^P, \{s_0\}^P, R{\uparrow}P, {\sim}{\uparrow}P)$.

A set of volition-guarded transactions with an equivalence on their underlying machine transactions likewise induces a multiagent transition system, over agent states:

Definition 2.11 (Volition-Guarded Transaction Set [28]). Let $P \subset \Pi$ be agents and S machine states with initial state s_0. A *volition-guarded transaction set* over P and S is a set R of volition-guarded transactions and volition-guarded if-then-else transactions, where a volition-guarded if-then-else transaction $s \to s'\,;\, s''$ guarded by p stands in R for its then-branch $(s \to s', \{p\})$ together with its else-branch, such that every $(t, Q') \in R$ has the participants of t in P, and every guard p of a volition-guarded if-then-else transaction is in P. An equivalence $\sim$ on the set $T_R := \{t : (t, Q') \in R \text{ for some } Q'\}$ of underlying machine transactions (a total equivalence, hence a partial equivalence in the sense of Definition 2.7) is *compatible* with R if no else-branch is in T_R, two volition-guarded if-then-else transactions with the same else-branch have $\sim$-equivalent then-branches, and the P-closures of two machine transactions of different classes, each the machine transaction of an unguarded transaction in R or an else-branch, are disjoint.

Definition 2.12 (Volitional Multiagent Transition System [28]). Given a volition-guarded transaction set R over P and S and an equivalence $\sim$ compatible with it, the *volitional multiagent transition system induced by* $(S, R, \sim)$ *over* P is the multiagent transition system $(\mathcal{A}^P, c_0, T_V, {\sim_V})$ where:

(1) $\mathcal{A} := 2^{T_R/\sim} \times S$ is the *agent state space*;

(2) $c_0 \in \mathcal{A}^P$ is the *initial agent configuration*, with ${c_0}_p^v = \emptyset$ and ${c_0}_p^m = s_0$ for every $p \in P$;

(3) T_V consists of all transitions $e \to e' \in (\mathcal{A}^P)^2$ of one of three forms: (*i*) a *volition change* of some $p \in P$—$e_p^v, {e'}_p^v \subseteq (T_R/\sim)_p$ and $e_p^v \neq {e'}_p^v$, $e_p^m = {e'}_p^m$, and $e_r = e'_r$ for every $r \in P \setminus \{p\}$; (*ii*) a *volitional machine transaction* induced by some volition-guarded transaction $(t, Q') \in R$ per Definition 2.4(2); or (*iii*) an *else transaction* induced by some volition-guarded if-then-else transaction in R per Definition 2.6;

(4) ${\sim_V}$ is the restriction of ${\sim}{\uparrow}P$ (Definition 2.9) to the volitional machine transactions and else transactions, relating two of them whenever their inducing machine transactions are $\sim$-equivalent, where the inducing machine transaction of an else transaction is the then-branch $s \to s'$ of its volition-guarded if-then-else transaction, and leaves every volition-change transition outside its domain.

Every transition of T_V other than a volition change has $\sim$-equivalent inducing machine transactions, so ${\sim_V}$ is a partial equivalence. A step induced by a guarded transaction (t, Q'), $Q' \neq \emptyset$, removes $[t]$ from the volitional state of each guard, which held it; a step of another class removes another class, and an else transaction removes none, so neither is that step, and the P-closures of two guarded transactions of different classes need not be disjoint. Two unguarded transactions of different classes share no step, by the condition on P-closures. The two conditions on else-branches give each else transaction one class: an else transaction is related through the then-branch of its if-then-else transaction, and two if-then-else transactions with the same else-branch have equivalent then-branches. The set R of volition-guarded transactions that specifies a platform thus determines the volitional machine transitions of the induced volitional multiagent transition system, while volition-change transitions are freely available to every agent independent of R. The system of [28] has a fourth form of transition, an *input transition*, induced by an *input*, a unary machine transaction of one agent that is in no class and requires no will; we recall the system with no inputs.

We define when a volition-guarded transaction is enabled, combining the machine precondition with the volitions of its guards.

Definition 2.13 (Enabled [28]). Given a set of volition-guarded transactions, each (t, Q') with $t = d \rightarrow d'$ a machine transaction over some $Q' \subseteq Q \subseteq P$ and S, and an equivalence $\sim$ on machine transactions: the volition-guarded transaction (t, Q') is *machine-enabled* in agent configuration c over P if $c_p^m = d_p$ for every $p \in Q$, and *enabled* if in addition $[t] \in c_q^v$ for every $q \in Q'$. The else-branch of a volition-guarded if-then-else transaction $s \rightarrow s' ; s''$ guarded by p is *enabled* in c if $c_p^m = s$ and $[s \rightarrow s'] \notin c_p^v$. An equivalence class $[t]$ is *enabled* in c if some volition-guarded transaction (t', Q') with $t' \in [t]$ is enabled in c, or the else-branch of some volition-guarded if-then-else transaction with then-branch $t' \in [t]$ is enabled in c.

A volition-guarded transaction with an empty guard ($Q' = \emptyset$) requires no volitions and is enabled whenever its machine precondition is met. The class of the then-branch of a volition-guarded if-then-else transaction guarded by p is thus enabled whenever p's machine is in the state s: by its then-branch if p wills it, and by its else-branch if not. For the volitional multiagent transition system above, a class of volitional machine transactions and else transactions is enabled at a configuration in the sense of Definition 2.7 exactly when some volition-guarded transaction or else-branch inducing it is enabled in the above sense; this determines which runs are live and correct per Definition 2.7. Volition-change transitions belong to no class and so impose no liveness obligation; personal choices remain free.

A *protocol* $\mathcal{F}$ is a family of multiagent transition systems, $\mathcal{F}(P)$ for each finite $P \subset \Pi$, over a common local-states function assigning to each P a set of local states containing the initial state [28, 50]. A set of volition-guarded transactions with a transaction equivalence over a local-states function thus defines a protocol: it assigns to each P the volitional multiagent transition system it induces over P.

2.4 Implementations

An implementation relates a specification transition system to one that implements it, via a mapping of configurations.

Definition 2.14 (Implementation (Refinement) [28, 48, 57]). Given two transition systems $TS = (C, c_0, T, \sim)$, the *specification*, and $TS' = (C', c_0', T', \sim')$, an *implementation of TS by TS'* is a function $\sigma : C' \rightarrow C$ with $\sigma(c_0') = c_0$, in which case the pair (TS', σ) is *an implementation of TS*. Given a computation $r' = c_1', c_2', \ldots$ of TS', $\sigma(r')$ is the computation $\sigma(c_1'), \sigma(c_2'), \ldots$ with consecutive repetitions removed; a transition $c' \rightarrow d'$ of TS' with $\sigma(c') = \sigma(d')$ is a *stutter*, contributing no step to $\sigma(r')$.

Definition 2.15 (Correct and Complete Implementation [28, 48, 57]). An implementation (TS', σ) of TS is *correct* if σ maps every correct run of TS' to a correct run of TS, and *complete* if every correct run r of TS is $\sigma(r')$ for some correct run r' of TS'.

A correct implementation is a refinement: a map from the implementing system's configurations to the specification's under which every correct run of the one is a correct run of the other. The implementations of this paper each hold for the runs of the implementing system that meet a condition — that an initiator knows its participants, that variables are named apart, that a question put to a person is answered — and we attach the condition to the system.

Definition 2.16 (Fairness Condition). A *fairness condition* on a transition system TS is a set F of runs of TS. A run of TS under F is *correct* if it is correct and in F. Where a fairness condition refers to infinitely many configurations of a run, for a finite run it refers to its last configuration.

Definition 2.15 applies to a transition system with a fairness condition, whether it is implemented or implements, with its correct runs being those that are correct under the condition: where no condition is stated, every run is in it.

3 Communicating Volitional Agents

The framework of Section 2 abstracts over *how* a volition-guarded multiagent atomic transaction is carried out: a k-ary transaction simultaneously updates the machine states of all its participants, with no notion of how participating machines come to act in concert. An implementation on networked smartphones has no such simultaneity: every exchange between two agents is a separate asynchronous message-passing event, and a k-ary transaction must be implemented by a protocol composed of such events.

3.1 Definition

To achieve that, we define *communicating volitional agents* (CVA), an implementation-ready restriction of volition-guarded transactions; the implementation of volition-guarded transactions by CVA is Section 3.2. CVA volition-guarded machine transactions are drawn from four syntactic forms: binary *discover* transactions by which two agents come to know each other, by the will of both persons; binary *introduce* transactions by which two agents that a common peer introduced come to know each other, with no person asked; binary *communicate* transactions that move a message from one agent's outbox to another's inbox; and unary *app* transactions. Thus, a CVA platform need only specify its app transactions. Moreover every CVA protocol is volitionally grassroots (Proposition 3.6).

Machine states. A CVA agent's machine state has four components: a set of *known peers* with whom the agent can communicate, an *outbox* of messages awaiting delivery, an *inbox* of messages received, and an *app state* in which each platform specifies whatever data it requires (friend sets, groups, coins, feeds, and so on). A *message* is a triple of sender, recipient, and cargo; outboxes and inboxes are sets of messages. The cargo space C and the app state space A are parameters of the CVA platform.

Definition 3.1 (Messages). Given a set C of *cargoes*, the set of *messages* over $P \subset \Pi$ is

$$M(P) \;:=\; \{\mathsf{message}(s, r, c) \mid s \neq r \in P,\; c \in C\},$$

with $\mathsf{message}(s, r, c)$ a message of *sender* s, *recipient* r, and *cargo* c.

A message goes to another agent: an agent changes its own state by an app transaction, with no message.

Definition 3.2 (CVA Machine States and Configurations). Given messages over P and an *app state space* A with initial state $a_0 \in A$, a *CVA machine state* over P is a tuple

$$(\mathit{known},\, o,\, i,\, a) \;\in\; S(P) \;:=\; 2^P \times 2^{M(P)} \times 2^{M(P)} \times A,$$

where $\mathit{known} \subseteq P$ is a set of *known peers*, $o \subseteq M(P)$ is an *outbox*, $i \subseteq M(P)$ is an *inbox*, and $a \in A$ is an *app state*; the *initial machine state* is $(\emptyset, \emptyset, \emptyset, a_0)$. A *CVA machine configuration* over P is a member $c \in S(P)^P$; for $p \in P$ we write $c_p = (\mathit{known}_p,\, o_p,\, i_p,\, a_p)$ for the machine state of p in c. The *initial machine configuration* over P assigns the initial machine state to every $p \in P$.

$P \subseteq P' \implies S(P) \subseteq S(P')$, so that the agent states of the transition system below (Definition 2.3) grow with P, as a protocol requires.

Definition 3.3 (App Transaction, Effect). Given cargoes C and an app state space A, an *app transaction* of p is a unary volition-guarded transaction, or a volition-guarded if-then-else transaction (Definition 2.5), over p and CVA machine states, each of whose targets c'_p, from its source c_p, has $\mathit{known}'_p = \mathit{known}_p$, $i'_p \subseteq i_p$, $o_p \subseteq o'_p$ and $o'_p \setminus o_p \subseteq \{\mathsf{message}(p, r, x) : r \in \mathit{known}_p,\; x \in C\}$, a cargo $\mathsf{intro}(q)$ having $q \in \mathit{known}_p$. Its *effect* $\mathit{eff}(\tau)$ is $(a'_p,\; o'_p \setminus o_p,\; i_p \setminus i'_p)$, and its *else-effect* $\mathit{eff}'(\tau)$ is the same triple for the else-branch's target.

Definition 3.4 (CVA Platform). A *CVA platform* is a tuple $(C, A, a_0, T, \sim)$ of cargoes C, which include the *introductions* $\text{intro}(q)$, $q \in \Pi$, an app state space A with initial state $a_0 \in A$, a set T of app transactions, and a transaction equivalence $\sim$ on T, such that for $\tau \sim \tau'$ in T whose sources differ only in *known*, $\mathit{eff}(\tau) = \mathit{eff}(\tau')$ and $\mathit{eff}'(\tau) = \mathit{eff}'(\tau')$. The *transactions* of the platform over $P \subset \Pi$ are the members of T over P and, for all $p \neq q \in P$ and $m \in M(P)$, the following machine transactions, in which nothing else changes:

(1) **Discover**(p, q), over $\{p, q\}$ and guarded by $\{p, q\}$: $\mathit{known}'_p = \mathit{known}_p \cup \{q\}$ and $\mathit{known}'_q = \mathit{known}_q \cup \{p\}$;
(2) **Introduce**(p, q), over $\{p, q\}$, unguarded, provided $q \notin \mathit{known}_p$ and, for some r and r', $m = \text{message}(r, p, \text{intro}(q))$ is in i_p and $m' = \text{message}(r', q, \text{intro}(p))$ in i_q: $\mathit{known}'_p = \mathit{known}_p \cup \{q\}$, $\mathit{known}'_q = \mathit{known}_q \cup \{p\}$, $i'_p = i_p \setminus \{m\}$ and $i'_q = i_q \setminus \{m'\}$;
(3) **Communicate**(p, q, m), over $\{p, q\}$, unguarded, provided $m = \text{message}(p, q, x) \in o_p$ and $m \notin i_q$: $o'_p = o_p \setminus \{m\}$ and $i'_q = i_q \cup \{m\}$;

with $\sim$ extended to relate every Discover and Introduce of the same p and q, and two Communicates of the same p, q and m.

Definition 3.5 (CVA Agent State, Transition System, Protocol). A *CVA agent state* of p is a pair (V, s) of a *volitional state* V, a set of classes of transactions guarded by p, and a CVA machine state $s \in S(P)$; a *CVA agent configuration* over P assigns each $p \in P$ a CVA agent state of p. The *transition system* of a CVA platform over P is the volitional multiagent transition system its transactions over P induce (Definition 2.12), on CVA agent configurations; the *protocol* of the platform assigns it to each finite $P \subset \Pi$.

Discover, Introduce and Communicate are binary. Knowing is mutual, so Discover and Introduce each add each of the two agents to the other's known peers at once, and Discover is guarded by both persons, which a unary transaction, guarded at most by its one participant, cannot be. Communicate moves a message from one agent to the other.

A CVA platform specifies its app transactions; Discover, Introduce and Communicate are built in. An app transaction reads its agent's machine state and changes only the app state, the outbox, adding messages addressed to known peers, and the inbox, deleting messages it has read. It may require that a peer be known, and its effect does not depend on which other peers are known. The reason is the implementation of Section 5.4: there an agent comes to know a peer by exchanging hellos with it, one hello at a time, so while a Discover is under way the two agents' sets of known peers differ from the specification's. A test that one peer is known stays correct, and a computation over the whole set would not. The transition system of a platform is a volitional multiagent transition system (Definition 2.12), which requires the P-closures of two unguarded transactions of different classes to be disjoint, and the P-closures of any two distinct CVA transactions are. Each changes the state of every one of its participants: a binary one changes the states of both, and a unary one the state of its one participant. Two transactions of different forms change different components: Discover changes only the known peers; Introduce changes the known peers and the two inboxes; Communicate changes an outbox and an inbox; and an app transaction changes the app state, the outbox and the inbox, and not the known peers.

An app transaction may be unguarded, no person's will being needed for it, and its precondition may require an arrived message. A transaction that requires a message may delete it. A guarded app transaction that requires an arrived message may have an else-branch. Then it is taken on the message either way, by its then-branch where its person wills it and by its else-branch where they do not, and either branch may delete the message. A Discover carries no message: the caller has the other's identifier, and an implementation realises it by a cold call, carrying the caller's identifier,

that the called person accepts, the two hellos of Section 5.4. An Introduce is taken by the machines, with no person asked: a peer that knows two agents sends each an introduction to the other, and the two come to know each other; it is equivalent to their Discover, and so fulfils a will to discover. Machines come to know each other by a Discover their persons will, or by an introduction; the platform records who is a friend, a parent or a child.

CVA protocols are volitionally grassroots. A protocol is *grassroots* if its groups run undisturbed by one another and can come to interact, and *volitionally grassroots* if its groups run undisturbed by one another, two groups each holding an agent that has not left can come to interact, and their first interaction is willed by a member of each [28]; the definitions are recalled in Appendix A. Both rest on the same fact: two agents learn of each other only by Discover, which both persons will, or by Introduce, through a peer that already knows both, and an agent addresses only agents it knows.

Proposition 3.6 (CVA protocols are volitionally grassroots). *The protocol of every CVA platform is grassroots and volitionally grassroots.*

The proof is in Appendix B.

3.2 Implementing the Transactions by Communicating Volitional Agents

A volition-guarded transaction changes the machine states of its k participants at once (Section 2). A communicating volitional agent changes only its own state, and reaches another agent only by a message (Section 3). Here we implement the first by the second, for every local volition-guarded specification, by two-phase commit coordinated by the agent that initiates the transaction [19].

Local specifications. In the framework of Section 2 each agent has a machine state (Definition 2.1), and a transaction's precondition may relate the states of its participants. In CVA a participant reads and changes only its own state, and learns of the transaction, with its arguments, from the initiator's message. The implementation therefore requires two things of each *part* of a class, a set of its transactions that one attempt carries out, the class being the union of its parts. First, its precondition is a conjunction of preconditions, one on the state of each participant, which may be the trivial one, true. Second, its effect on each participant depends only on that participant's state and on the transaction's arguments. The implementation below assumes this. For example, the precondition that two participants' dates be equal is made local by naming the date in the transaction, which the initiator sends to every participant, each checking its own date against it. The platforms of Section 7 are local: befriending adds each party to the other's friend set, provided each is absent from it, and a swap moves each party's coins to the other, provided each holds what it offers.

The framework of Section 2 specifies a platform by a set of volition-guarded transactions and an equivalence on them. The definition below calls these, with the machine states, a *specification*, and Definition 3.8 names, for each part of a class of transactions, the set of persons whose volitions can enable it, which is what the implementation needs.

Definition 3.7 (Volition-Guarded Specification). A *volition-guarded specification* is a tuple $(S, s_0, R, \sim)$ of machine states S with initial state $s_0 \in S$, a set R of volition-guarded transactions over S, none of them a volition-guarded if-then-else transaction, and a transaction equivalence $\sim$ on their underlying machine transactions T_R, as in Definition 2.11.

For a class $[t] \in T_R/\sim$, $Q([t])$ is the set of participants of its members.

Definition 3.8 (Local Specification, Part, Guard Set, Opener, Request). A volition-guarded specification is *local* if every class $[t]$ is the union of sets of its members, its *parts*, each part $[t]_j$ with a *guard set* $G([t]_j) \subseteq Q([t])$, such that: for every part $[t]_j$ and every $p \in Q([t])$ there is a partial

function $f_p^{[t]_j} : S \rightharpoonup S$ with the members of $[t]_j$ exactly the machine transactions $d \longrightarrow d'$ over $Q([t])$ with $d_p \in dom(f_p^{[t]_j})$ and $d'_p = f_p^{[t]_j}(d_p)$ for every $p \in Q([t])$, every such pair having $d' \neq d$; $(t', G([t]_j)) \in R$ for every member t' of $[t]_j$; and every $(t', Q') \in R$ with $t' \in [t]$ has a part of $[t]$ that contains t' and whose guard set is contained in Q'. A *guard* of $[t]_j$ is a member of $G([t]_j)$, and a guard of $[t]$ is a guard of one of its parts; $[t]_j$ is *unguarded* if $G([t]_j) = \emptyset$, and *guarded* otherwise. A local specification designates for each unguarded part $[t]_j$ a participant $o([t]_j) \in Q([t])$, its *opener*, and may designate one for a guarded part with one guard, a participant $o([t]_j) \in Q([t]) \setminus G([t]_j)$, such that the guarded parts of a class that one participant guards have one opener, or none. A *request* of a guard p to an agent o is a finite nonempty set of classes whose guarded parts that p guards have the opener o, or, for $o = p$, a single class whose guarded parts that p guards have no opener.

Under locality each participant p has, in each part $[t]_j$, its own precondition, the domain $dom(f_p^{[t]_j})$, and its own effect, $f_p^{[t]_j}$, and a member of $[t]_j$ is enabled exactly when the precondition in $[t]_j$ of every participant holds. A part carries the transaction's arguments, so $f_p^{[t]_j}$ may depend on them. For example, the effect of a swap on p removes the coins p offers and adds the coins p wants, both named in the swap, and does not read the other party's state. A class has more than one part where its transactions can be taken in different ways: where two friends each parent both of two children, the befriending of the children may be approved by either parent for either child [54], and each way is a part; and where a transaction is guarded by either of two persons, as unfriending is, its class has two parts with the same members, one guarded by each. A guarded part has an opener where only a participant that is not its guard can propose it: a parent of a child may close a group the child leads, and only the child's state lists the group's members, so the child's machine proposes the closing, on the parent's request [54]. A request is a set of classes, since the guard need not know which of them is to be taken: the members of the group take part in its closing, so the closing with each list of members is a class of its own, and the parent wills the closing whatever the list. A guard of a part with no opener makes its request to its own machine.

The *protocol* of a local specification assigns to each $P \subset \Pi$ the volitional multiagent transition system that $(S, R, \sim)$ induces over P (Definition 2.12), restricted to the agent configurations in which every agent wills only classes of which it is a guard.

The protocol. A guard's person approves a transaction once, by a *request* or by answering yes when asked, and the *approval* stands until the transaction is taken or refused. An *attempt* at a transaction is proposed by one of its participants, its *initiator*, and names a part of the transaction's class; each participant checks its own precondition in that part. A part with an opener is proposed by its opener, an unguarded one whenever the opener can carry out its effect, and a guarded one on a request of its guard, which the attempt names; the opener votes no on such an attempt at once if it cannot carry out its effect. A guarded part with no opener is proposed by one of its guards whose approval stands, which votes no on it at once if it can carry out no such part. The other guards of the part are *asked*: the guard of an attempt that names a request votes yes only while that request stands, a guard whose approval stands votes without asking its person, and the persons of the other guards must approve. The machine of every other participant votes on its own precondition, without asking its person. Each participant, and no other agent, votes once, yes or no, to the initiator. The initiator commits when it holds a yes from every participant, and aborts on the first no, a *refusal*, or when asked to yield before it holds them all. A commit ends every approval of the transaction, and a refusal ends the approvals of the guards that voted on the attempt and the request it names, and their persons are told who refused it and why, declined or unable; an attempt that yields ends none, and its initiator proposes it again. A machine that cannot carry out

a transaction its person approved refuses it, so that the approval ends and the person is told. A participant votes yes only if it can carry out its effect and holds no other attempt *prepared*. From its yes vote until it learns the outcome, it holds the attempt prepared. No transaction of the protocol withdraws an approval. An asked participant first checks that it can carry out its effect, and only then, where no approval of it stands, asks its person, once. It votes no at once if its person answered no; on yes, it votes once it holds no other attempt prepared, yes if it still can and no otherwise.

What should a participant do with a start that arrives while it holds another attempt prepared? Voting no would lead to livelock: two attempts that reach two agents in different orders could abort each other again and again. Waiting would lead to deadlock: two attempts could each wait for the other. We use wound-wait [45]. A participant that holds another attempt prepared, or has a question of the same class pending, lets the start wait, and takes waiting starts in order of *priority*. The priority of an attempt is its *rank*, a number, the smaller rank having priority, with ties broken by the *order of identifiers*, a fixed total order of the agents. Where a waiting attempt has priority over the attempt that blocks it, the participant asks the initiator of the blocking attempt to *yield*, and that initiator aborts it unless it has already decided it. An initiator proposes nothing while a start waits in its inbox, or while a question it answered yes waits. So an attempt waits only on one with priority over it, or on one that is yielding or has been decided and so ends without waiting, and no cycle of waiting arises. No participant votes no because of contention: contention aborts an attempt only by a yield to an attempt with priority over it. An attempt aborted in favour of one with priority over it keeps its rank when its initiator proposes the part again, so only finitely many attempts ever have priority over it, and none is starved.

Definition 3.9 (Attempt). Fix a local volition-guarded specification. An *attempt* is a tuple $\alpha = (i, k, \pi, [t]_j, r)$ of an *initiator* i, a *number* $k \in \mathbb{N}$, a *rank* $\pi \leq k$, a part $[t]_j$ of a class $[t]$, and, where $[t]_j$ is guarded and has an opener, a request $r = (\rho, k')$ of its guard numbered k', the request that α *names*, and $r = \bot$ otherwise; it is an attempt of $[t]_j$ and of $[t]$. The initiator is the opener $o([t]_j)$ of $[t]_j$ if $[t]_j$ has one, and a guard of $[t]_j$ if not. The *participants* of α are those of $[t]$. Its *guard set* G_α is $G([t]_j)$, and the members of $G_\alpha \setminus \{i\}$ are *asked* by it.

The number identifies the attempt: each attempt an initiator proposes has a new number, larger than every number it has seen. The rank orders the attempt: it is the number itself, or, where the attempt retries one of the same part that was aborted in favour of an attempt with priority over it, the rank of that one, so that the retried attempt keeps its priority.

Definition 3.10 (Cargo, App State, Question, Approval, Free, Ready). A *cargo* of the protocol is $\mathsf{start}(\alpha)$, $\mathsf{vote}(\alpha, y)$ with $y \in \{\mathsf{yes}, \mathsf{no}\}$, a vote no carrying its *reason*, declined or unable, $\mathsf{outcome}(\alpha, z)$ with $z \in \{\mathsf{commit}, \mathsf{abort}, \mathsf{yield}\}$, an abort carrying its *refuser*, the participant whose vote no refused α, with that vote's reason, $\mathsf{yield}(\alpha)$, or $\mathsf{request}(\rho, k)$ with ρ a request and $k \in \mathbb{N}$. An *app state* of p is a tuple (s, n, L, R) of a machine state $s \in S$; p's *number* $n \in \mathbb{N}$, the largest number of an attempt it has proposed or received a start of, or of a request it has made; a *lock* L, either $\bot$ or the attempt p holds *prepared*; and a finite *record* R of terms $\mathsf{asked}(\alpha)$, $\mathsf{answered}(\alpha, y)$, $\mathsf{voted}(\alpha, y)$, $\mathsf{got}(\alpha, q, y)$, $\mathsf{closed}(\alpha)$, $\mathsf{wounded}(\alpha)$, $\mathsf{yielded}(\alpha)$, $\mathsf{requested}(\rho, k)$, $\mathsf{ended}(x)$ and $\mathsf{introduced}(q, r)$. An attempt α of $[t]$ is a *question* of $[t]$ *pending* at p if p has recorded $\mathsf{asked}(\alpha)$ and neither a vote on α nor $\mathsf{closed}(\alpha)$. The *approvals* of p are the requests $\mathsf{requested}(\rho, k)$ it has recorded, approvals of each class of ρ, and its answers $\mathsf{answered}(\alpha, \mathsf{yes})$, approvals of the class of α; an approval is *pending* at p until p *ends* it, recording $\mathsf{ended}(\rho, k)$ or $\mathsf{ended}(\alpha)$. An agent p *holds* a request (ρ, k) of $[t]$ from q if $\mathsf{message}(q, p, \mathsf{request}(\rho, k))$, with $[t] \in \rho$, is in its inbox. The agent p is *free for* $[t]$ if $L = \bot$ and no question of $[t]$ is pending at p, and *ready for* a part $[t]_j$ of $[t]$ if moreover its precondition in that part holds, $s \in dom(f_p^{[t]_j})$.

Being free and being ready are conditions on the machine state, not on the person's will.

Definition 3.11 (Priority, Waiting, Blocked, Clear). Fix a total order on Π, the *order of identifiers*. An attempt α of rank π has *priority* over an attempt β of rank π' if $\pi < \pi'$, or $\pi = \pi'$ and the initiator of α precedes that of β in the order of identifiers. A message start(α), α of $[t]$, in p's inbox *waits* at p in three cases: while p's inbox holds the start of an attempt with priority over α; while a question with priority over α, answered yes, is pending at p; and while p is not free for $[t]$. In the last case the start is *blocked* by L if $L \neq \perp$, and otherwise by the question of $[t]$ pending at p. A question α pending at p and answered yes *waits* while a question with priority over it and answered yes is pending at p, and while $L \neq \perp$, when it is *blocked* by L. The agent p is *clear* if no start is in its inbox and no question answered yes is pending at it.

The app transactions of the canonical platform, defined below, use two operations on an attempt $\alpha = (i, k, \pi, [t]_j, r)$ at an agent p: *tell* x adds message(p, q, x) to o_p for every participant $q \neq p$ of α, and *vote* y records voted(α, y), adds message$(p, i,$ vote$(\alpha, y))$ to o_p, and, if $y =$ yes, sets $L := \alpha$, and, if $y =$ no and p is a guard of $[t]_j$, ends every approval of $[t]$ pending at p; the reason of a vote no is declined where p has recorded answered$(\alpha,$ no$)$ and unable otherwise. A transaction's precondition is a condition on p's machine state, and its guard, where it has one, is p (Definition 2.1). Every app transaction addresses only known peers (Definition 3.3).

Definition 3.12 (Proposing). The *proposing and voting transactions* of an agent p are the app transactions of this definition and the next two. Each deletes the message it reads, except **Wound**, and each is taken only where p knows every agent it sends a message to.

(1) **Propose** α, by its initiator $p = i$. Precondition: p is free for $[t]$, so $L = \perp$, and clear; if $[t]_j$ is unguarded, p is ready for $[t]_j$; if $[t]_j$ is guarded and p is its opener, α names a request r of $[t]$ from the guard of $[t]_j$ that p holds; and if $[t]_j$ is guarded and has no opener, an approval of $[t]$ is pending at p, and $s \in dom(f_p^{[t]_j})$ or $s \notin dom(f_p^{[t]_{j'}})$ for every guarded part $[t]_{j'}$ of $[t]$ with no opener that p guards. Effect: p's number n is increased by one, and the new value is the number k of α. The rank π of α is that of p's last attempt of $[t]_j$ if p has recorded that attempt wounded, and k otherwise. If $s \in dom(f_p^{[t]_j})$, p records voted$(\alpha,$ yes$)$, sets $L := \alpha$ and tells start(α). Otherwise p records voted$(\alpha,$ no$)$ and closed(α), tells outcome$(\alpha,$ abort$)$ with itself as refuser, and, being the opener of $[t]_j$, deletes r from i_p, or, being a guard of $[t]_j$, ends every approval of $[t]$ pending at it. Unguarded.

(2) **Request** ρ to o, by a guard p, ρ a request of p to o. Precondition: p is free for every class of ρ, ready for a guarded part of a class of ρ that it guards, and ρ is not a request pending at p. Effect: p's number n is increased by one, to k, p records requested(ρ, k), and, if $o \neq p$, adds message$(p, o,$ request$(\rho, k))$ to o_p. Guarded by p.

Definition 3.13 (Starting and Wounding). The proposing and voting transactions of p include the following.

(3) **Start**, on start(α). Effect: p's number n becomes the larger of n and k. If closed(α) is recorded, nothing more. Otherwise, provided the start does not wait at p: if $s \notin dom(f_p^{[t]_j})$, p votes no; and otherwise, where α names a request r of p, p votes yes if r is pending at p and no if not; where p is asked by α, which names none, and no approval of $[t]$ is pending at p, p records asked(α); and else p votes yes. Unguarded.

(4) **Wound**, on a start of α waiting at p, or on a question α pending at p and answered yes. Precondition: α is blocked by an attempt β over which α has priority; p has not recorded yielded(β); and, if p initiated β, p has not recorded a yes of every other participant of β. Effect: if p initiated β, $L := \perp$, p records closed(β) and wounded(β), and p tells outcome$(\beta,$ yield$)$.

Otherwise p adds message$(p, j, \mathsf{yield}(\beta))$ to o_p, j being the initiator of β, and records yielded(β). A start it reads stays in p's inbox. Unguarded.

Definition 3.14 (Answering and Voting). The proposing and voting transactions of p include the following.

(5) **Answer** α. Precondition: α is a question pending at p, and p has recorded no answer to it. Effect: p records answered(α, yes). Guarded by p. Its *else-branch* (Definition 2.5) records answered(α, no).

(6) **Vote** α. Precondition: α is a question pending at p, p has recorded answered(α, y), and $y = \mathsf{no}$ or α does not wait at p (Definition 3.11). Effect: p votes yes if $y = \mathsf{yes}$ and $s \in dom(f_p^{[t]_j})$, and no otherwise. Unguarded.

Definition 3.15 (Deciding). The *deciding transactions* of an agent p are the app transactions below. Each deletes the message it reads, and each is taken only where p knows every agent it sends a message to.

(7) **Tally**, by the initiator $p = i$, on vote(α, y) from q. Effect: if closed(α) is recorded, nothing more. Otherwise p records got(α, q, y), and then, if $y = \mathsf{no}$, $L := \bot$, p records closed(α) and tells outcome(α, abort) with q as refuser and the reason of its vote; p ends every approval of $[t]$ pending at it if it is a guard of $[t]_j$, and deletes the request α names, if any, from i_p. Unguarded.

(8) **Commit** α, by its initiator $p = i$. Precondition: $L = \alpha$, and a yes of every participant other than p is recorded. Effect: $s := f_p^{[t]_j}(s)$, $L := \bot$, and p records closed(α) and tells outcome$(\alpha, \mathsf{commit})$; p ends every approval of $[t]$ pending at it, and deletes from i_p every request of $[t]$ it holds; and for every two participants $q \neq r$ of α other than p for which p has not recorded introduced(q, r), p adds message$(p, q, \mathsf{intro}(r))$ and message$(p, r, \mathsf{intro}(q))$ to o_p and records introduced(q, r). Unguarded.

(9) **Yield**, by the initiator $p = i$, on yield(α) from q. Effect: if closed(α), or a yes of every participant other than p, is recorded, nothing more. Otherwise $L := \bot$, and p records closed(α) and wounded(α) and tells outcome(α, yield). Unguarded.

(10) **Outcome**, by a participant $p \neq i$, on outcome(α, z) from i. Effect: $s := f_p^{[t]_j}(s)$ if $z = \mathsf{commit}$, $L := \bot$ if $L = \alpha$, and p records closed(α). If $z = \mathsf{commit}$, p ends every approval of $[t]$ pending at it and deletes from i_p every request of $[t]$ it holds. If $z = \mathsf{abort}$, p ends the request α names if it is p's, and, where α names none, every approval of $[t]$ pending at it if p is a guard of $[t]_j$ that has recorded voted(α, yes) or answered(α, yes). Unguarded.

Definition 3.16 (Equivalence of the Canonical Platform). The *transaction equivalence* of the canonical platform has the following classes.

- For a class $[t]$ and a guard p of it, p's *approval class* on $[t]$: p's **Request** transactions of $\{[t]\}$, and its **Answer** transactions, by either branch, on attempts of $[t]$. It is guarded by p.
- For a guard p and a request ρ of p of two or more classes, p's *request class* on ρ: its **Request** transactions of ρ. It is guarded by p.
- For a part $[t]_j$ and an agent p, p's *proposing class* on $[t]_j$: its **Propose** transactions for attempts of $[t]_j$.
- For an agent and a message: its **Start**, **Tally**, **Yield** or **Outcome** transactions on that message.
- For an agent and an attempt: its **Vote** transactions on that attempt, and its **Commit** transactions on it.
- For an agent and two attempts: its **Wound** transactions for them.

The approval class is what a person wills in CVA when they will $[t]$ in the specification: their machine fulfils it by requesting $[t]$ or by answering yes to an attempt of it. A request class is what a person wills when they will each class of the request. The else-branch of **Answer** records the answer no, which no other transaction records, so no else-branch is a machine transaction of the platform. It records its attempt, so two **Answer** transactions with the same else-branch are of one class (Definition 2.12).

Definition 3.17 (Canonical CVA Platform). The *canonical CVA platform* of a local volition-guarded specification is the CVA platform (Definition 3.4) with the cargoes of Definition 3.10 and the introductions, the app states of Definition 3.10, initial app state $(s_0, 0, \bot, \emptyset)$, the app transactions of Definitions 3.12 to 3.15, and the transaction equivalence of Definition 3.16.

A run of $\mathcal{P}$ is correct only where the initiator of an attempt knows its participants (Definition 3.21 below), and agents come to know each other by Discover, which both persons will, or by an introduction from a peer that knows both. **Commit** introduces the other participants of its attempt to each other, so the parties to a transaction taken come to know each other, and a transaction refused introduces no one. A canonical platform may also send introductions by a rule of its own.

Definition 3.18 (Introduction Rule, Introduce Peers). An *introduction rule* ι gives, for every agent p and machine state $s \in S$, a finite set $\iota_p(s)$ of two-element sets of agents other than p, and, for every agent p and part $[t]_j$ with $p \in Q([t])$, a finite set $\iota_p([t]_j)$ of such sets. The canonical CVA platform *with* ι is the canonical CVA platform with, for every agent p and $\{q, r\}$, the app transaction **Introduce peers** $\{q, r\}$, unguarded: precondition: $\{q, r\} \in \iota_p(s)$, $q, r \in known_p$, and p has not recorded introduced(q, r); effect: p adds message$(p, q, \text{intro}(r))$ and message$(p, r, \text{intro}(q))$ to o_p and records introduced(q, r); and in which **Commit**, and **Outcome** on a commit, of an attempt of $[t]_j$ at p add the same messages and record for every $\{q, r\} \in \iota_p([t]_j)$ with $q, r \in known_p$ for which p has not recorded introduced(q, r). The **Introduce peers** transactions of an agent for one pair form a class.

Introduce peers, and the introductions a commit sends, change the outbox and the record, and the Introduce they lead to changes the known peers; the specification sees neither, so an introduction rule leaves the results below as they are.

Correctness. The transaction is carried out, at every participant at once, by the **Tally** that records the last yes of its attempt at its initiator, or by its **Propose** where it has one participant: from then on every participant's effect is settled, and the **Commit** and **Outcome** that apply it later change nothing the specification records.

Definition 3.19 (Decided, Committed, Aborted, Settled State). Let c be a configuration of the canonical CVA platform over P. An attempt α is *decided* in c if its initiator has proposed it, voting yes, and has recorded a yes of every other participant. It is *closed* at a participant p once p has recorded closed(α), *committed* there if the transaction that recorded it applied p's effect, and *aborted* there otherwise. The *settled state* of p in c is s_p, with $f_p^{[t]_j}$ applied if L_p is an attempt of a part $[t]_j$ decided in c and $s_p \in dom(f_p^{[t]_j})$.

For the rest of this section fix a local volition-guarded specification, an introduction rule or none, and agents $P \subset \Pi$, and write $\mathcal{P}$ for the transition system of the canonical CVA platform with that rule over P and $\mathcal{F}(P)$ for the volitional multiagent transition system the specification induces over P.

Proposition 3.20 (Atomic Commitment). *In every safe run of $\mathcal{P}$: (*agreement*) no attempt is committed at one participant and aborted at another; and (*validity*) an attempt is committed at a*

participant only if it is decided, each participant having voted yes on it holding no other attempt prepared, at a machine state in the domain of its effect, and each member of its guard set having voted yes on it with an approval of its class pending, made by a ***Request*** *or an answer yes, which its person willed. In every safe, live run of* $\mathcal{P}$*:* (termination) *every attempt is eventually closed at each of its participants.*

The proof is in Appendix B.

Definition 3.21 (Holds, For, Connected, Unobstructed, Steady). A part $[t]_j$ *holds* in a configuration if the settled state of every $p \in Q([t])$ is in $dom(f_p^{[t]_j})$, and an agent p is *for* a class $[t]$ if it holds its approval class on $[t]$ or its request class on a request holding $[t]$, or an approval of $[t]$ is pending at it. A run of $\mathcal{P}$ is *connected* if, for every part $[t]_j$ of a class $[t]$ that holds, and all of whose guards are for $[t]$, in every configuration of a suffix, its opener, if it has one, and otherwise one of its guards, knows every other participant of $[t]$ from some configuration on. A run is *unobstructed* if, for every agent p, class $[t]$, part $[t]_j$ of it, and each of these conditions — p is a guard of $[t]_j$ holding its approval class (Definition 3.16) on $[t]$ or its request class on a request holding $[t]$, its settled state (Definition 3.19) in $dom(f_p^{[t]_j})$; p is a guard of $[t]_j$ at which an approval of $[t]$ is pending, its settled state in $dom(f_p^{[t]_j})$; p is the opener of $[t]_j$, unguarded, its settled state in $dom(f_p^{[t]_j})$; p holds a request of $[t]$ — if infinitely many of its configurations meet the condition, then p is free for $[t]$ and clear in infinitely many of them. A run is *steady* if, for every class $[t]$, if from some configuration on no attempt of $[t]$ is decided while in every configuration some part of $[t]$ holds and all its guards are for $[t]$, then one part of $[t]$ holds, and all its guards are for $[t]$, in every configuration of a suffix.

Definition 3.22 (Strongly Fair, Fairness of $\mathcal{P}$*).* A run of $\mathcal{P}$ is *strongly fair* if, for every agent p and part $[t]_j$ of a class $[t]$ such that a **Propose** of p for an attempt of $[t]_j$ is enabled in infinitely many of its configurations, p takes infinitely many **Propose** transactions for attempts of $[t]_j$, or infinitely many attempts of $[t]$ are decided; and likewise for every agent p, class $[t]$ and agent o and the **Request** transactions of p to o of requests holding $[t]$. The *fairness condition* of $\mathcal{P}$ is the set of its connected, unobstructed, steady and strongly fair runs.

The protocol cannot secure these conditions. Knowing is never undone, so whether a run is connected follows from its Discover and Introduce transactions. Unobstructedness excludes an agent kept busy without end. Steadiness excludes a class, willed and not taken, whose part that holds with all its guards for it changes without end. The protocol resolves contention: contention makes an attempt yield only to an attempt with priority over it, and the attempt is proposed again, the approvals given to it standing. A refused attempt changes no machine state of the specification, and ends the approvals of the guards that voted on it and the request it names, a volition change of each of them (Definition B.1): a guard who still wills the transaction wills it again. The specification's liveness is weak fairness. For our implementation to achieve it, strong fairness is needed. The reason is that **Propose** may be enabled infinitely often but never continuously. The compiled agent of Section 5.4 secures strong fairness (Section 6.4).

Definition 3.23 (Implementation of Volition-Guarded Specifications by CVA, Correct, Complete). The *implementation of volition-guarded specifications by CVA* is the function taking a local volition-guarded specification and an introduction rule or none to its canonical CVA platform with that rule (Definitions 3.17 and 3.18). It is *correct* if for every such specification and $P \subset \Pi$ some σ is a correct implementation of $\mathcal{F}(P)$ by $\mathcal{P}$, and *complete for strongly fair runs* if for every such specification and $P \subset \Pi$ some such σ also takes some correct run of $\mathcal{P}$ to each correct run of $\mathcal{F}(P)$ that is *strongly fair*, every class enabled in infinitely many of its configurations being taken in infinitely many.

Theorem 3.24 (CVA Implementation). *The implementation of volition-guarded specifications by CVA is correct, and complete for strongly fair runs.*

The proof, with the mapping σ, is in Appendix B. Correctness is per transition and then per class: the yes that decides an attempt is the specification's transaction, enabled by validity, and every other step a stutter or a volition change; and a class enabled forever is decided, since, the run being steady, one of its parts holds with all its guards for the class from some point on and is proposed again and again, contention aborts its attempts only in favour of the finitely many attempts with priority over them, and no guard of that part refuses it. Steadiness is needed: with two parts guarded by Alice with Bob and by Alice with Carol, Alice's attempt that asks Bob while only Carol wills fails, and so does her next, which asks Carol while only Bob wills, and the class, enabled throughout, may never be taken. Completeness is by construction: for a correct, strongly fair run of the specification, a run of $\mathcal{P}$ in which every agent knows every other carries its transactions out one at a time, and idles, between them, on the attempts its persons' unmet wills open.

4 Grassroots Logic Programs

This section recalls GLP [53], the polymorphically-typed multiagent concurrent logic programming language that vGLP extends (Section 5): its syntax, its reduction, and its multiagent operational semantics maGLP, which vmaGLP extends with the person. GLP is typed by the parameterised moded type system of [58], which proved indispensable for AI-based programming and is not restated here: a program carries type definitions and procedure declarations. Its implementation is [57].

4.1 Syntax

The language extends logic programs by adding a paired *reader X*? to every "ordinary" logic variable X, now called a *writer* [53].

Definition 4.1 (Variables, Terms, Goals, Clauses [53]). Let $\mathcal{V}$ denote the set of LP variables (identifiers beginning with uppercase), henceforth called *writers*, and $\mathcal{V}? = \{X? \mid X \in \mathcal{V}\}$ the *readers*; $\hat{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}?$, and a writer X and its reader $X?$ form a *variable pair*. Terms, unit goals, goals, and clauses are as in LP, over $\hat{\mathcal{V}}$. A term, goal, or clause satisfies the *single-occurrence (SO) invariant* if every variable occurs in it at most once, and a clause satisfies the *single-reader/single-writer (SRSW) restriction* if it satisfies SO and a variable occurs in it iff its paired variable does.

Definition 4.2 (Guarded Clause, Procedure, GLP Program [53]). A *guarded clause* has the form H `:-` G `|` B, where H is the head, G a conjunction of guard predicates, and B the body; guard arguments are readers paired to head writers. A *GLP program* is a finite sequence of guarded clauses, each satisfying SRSW; the clauses for the same predicate form a *procedure*. The set of goals $\hat{\mathcal{G}}(M)$ of a program M includes all goals over $\hat{\mathcal{V}}$ and the vocabulary of M that satisfy SO.

Example 4.3 (Fair Merge [53]). The quintessential concurrent logic program for fairly merging two streams, in ordinary clauses:[1]

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

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

[1]Parameterised moded-type definitions (`T ::= ...`) and declarations (`procedure ...`) used informally in examples are formally introduced in [58].

```
merge([], Ys, Ys?).
merge(Xs, [], Xs?).
```

The goal `merge([1,2,3|Xs?],[a,b|Ys?],Zs)` satisfies SO; each clause satisfies SRSW. The first clause swaps inputs in the recursive call, ensuring fairness when both streams are available.

4.2 Reduction

Definition 4.4 (Substitutions and Assignments [53]). A *writer assignment* is a term of the form $X := T$, $X \in \mathcal{V}$, $T \notin \mathcal{V}$, satisfying SO. Similarly, a *reader assignment* is a term of the form $X? := T$, $X? \in \mathcal{V}?$, $T \notin \mathcal{V}$, satisfying SO. A *writers (readers) substitution* σ is the substitution implied by a set of writer (reader) assignments that jointly satisfy SO. Given a writers assignment $X := T$, its *readers counterpart* is $X? := T$, and given a writers substitution σ, its *readers counterpart* $\sigma?$ is the readers substitution defined by $X?\sigma? = X\sigma$. Given a reader assignment $X? := T$, its *writers counterpart* is $X := T$, and given a readers substitution τ, its *writers counterpart* $\tau!$ is the writers substitution defined by $X\tau! = X?\tau$. The *pair completion* of a readers substitution σ is $\sigma^\star = \sigma \cup \sigma!$, applied to a fixed point.

Definition 4.5 (Renaming, Renaming Apart, Writer MGU [53]). A *renaming* is an injective substitution $\rho : \hat{\mathcal{V}} \to \hat{\mathcal{V}}$ such that for each $X \in \mathcal{V}$: $X\rho \in \mathcal{V}$ and $X?\rho = (X\rho)?$. Two terms *have a variable in common* if for some writer $X \in \mathcal{V}$, either X or $X?$ occurs in both. A renaming σ *renames* T' *apart from* T if $T'\sigma$ and T have no variable in common. Given two unit goals A and H, a *writer mgu* is a writers substitution σ such that $A\sigma = H\sigma$ and σ is most general among such substitutions.

Definition 4.6 (Reduction [53]). Given a unit goal A and an ordinary clause C, with H `:-` G `|` B the result of renaming C apart from A, the *reduction* of A with C *succeeds with result* (B, σ) if A and H have a writer mgu σ and $G\sigma?$ succeeds.

Guards have three-valued semantics [53]. Each guard predicate explicitly defines its *success* condition. A guard *suspends* if it does not succeed but some instance of it under a readers substitution would succeed. A guard *fails* if no such instance exists. A guard conjunction succeeds if all members succeed; it suspends if any member suspends and none fail; it fails if any member fails. Guard occurrences count toward SRSW satisfaction, and a guard whose success implies that $X?$ is ground licenses multiple occurrences of X and $X?$ in the clause [53].

4.3 Multiagent Semantics

The operational semantics of GLP is maGLP. Each agent's initial resolvent is the atomic goal `agent(ch(_?,_),ch(_?,_))`, providing two bidirectional channels: the first to the person operating the machine, the second to the network [53]. Anonymous variables make the initial local states syntactically identical yet semantically unshared; variable pairs whose writer and reader are held by different agents are the communication channels, and Cold-call bootstraps shared variables between previously-disconnected agents through the network streams [53].

Definition 4.7 (Variables of an Agent [53]). Let $\mathcal{V}_p \subset \mathcal{V}$, for $p \in \Pi$, be pairwise disjoint infinite sets of writers. The *variables of agent* p are $\hat{\mathcal{V}}_p = \mathcal{V}_p \cup \{X? \mid X \in \mathcal{V}_p\}$.

The anonymous variables of the initial goal at p, and those a Reduce at p renames its clause into, are variables of p [53].

Definition 4.8 (Goal Identity [53]). Each unit goal in a computation carries a unique *identifier* assigned at spawn time: goals in an initial goal receive distinct initial identifiers, and goals spawned by a Reduce transition receive fresh identifiers. Communicate transitions preserve identifiers: the goal containing $X?$ retains its identity after instantiation.

Definition 4.9 (Asynchronous Resolvent [53]). Given a GLP program M, an *asynchronous resolvent* over M is a pair (G, σ) where $G \in \hat{\mathcal{G}}(M)$ and σ is a readers substitution.

Each rule below is read as in an operational semantics: above the line its condition, below it the transaction, from its participants' local states to their states after it, every other agent stationary; the prose before each rule says what the premise names.

Definition 4.10 (maGLP Transactions [53]). Given a GLP program M and agents $P \subset \Pi$, the *maGLP transactions* over asynchronous resolvents over M are the following.

Reduce p. $A \in G_p$ is a unit goal, and $C \in M$ is the first clause for which the reduction of A with C (Definition 4.6) succeeds, with result $(B, \hat{\sigma})$, C renamed into $\hat{\mathcal{V}}_p$.

$$\frac{A \in G_p \qquad C \text{ the first clause reducing } A, \text{ with result } (B, \hat{\sigma})}{(G_p,\ \sigma_p) \ \rightarrow\ ((G_p \setminus \{A\} \cup B)\hat{\sigma},\ \sigma_p \circ \hat{\sigma}?)}$$

Communicate p **to** q. Participants $p, q \in P$, possibly $p = q$: $\{X? := T\} \in \sigma_p$ and $X?$ occurs in G_q.

$$\frac{\{X? := T\} \in \sigma_p \qquad X? \text{ occurs in } G_q}{(G_p,\ \sigma_p),\ (G_q,\ \sigma_q) \ \rightarrow\ (G_p,\ \sigma_p \setminus \{X? := T\}),\ (G_q\{X? := T\},\ \sigma_q)}$$

Where $p = q$ the transaction is unary: $(G_p,\ \sigma_p) \rightarrow (G_p\{X? := T\},\ \sigma_p \setminus \{X? := T\})$. Communicate removes the assignment it communicates [53].

Cold-call p **to** q. Participants $p \neq q \in P$: the network output stream in (G_p, σ_p) has a new message $\mathtt{msg}(q, X)$; (G'_p, σ'_p) is the result of advancing the network output stream and (G'_q, σ'_q) the result of adding $\mathtt{msg}(q, X)$ to the network input stream.

$$\frac{\text{the network output stream of } p \text{ has a new } \mathtt{msg}(q, X)}{(G_p,\ \sigma_p),\ (G_q,\ \sigma_q) \ \rightarrow\ (G'_p,\ \sigma'_p),\ (G'_q,\ \sigma'_q)}$$

Definition 4.11 (maGLP Transition System [53]). Given a GLP program M and agents $P \subset \Pi$, the *maGLP transition system* over P and M is the transactions-based multiagent transition system (Definition 2.10) over P, local states being asynchronous resolvents, initial local state $(\{a_0\}, \emptyset)$ with a_0 the goal $\mathtt{agent(ch(_?,_),ch(_?,_))}$, its anonymous variables at p of $\hat{\mathcal{V}}_p$, the transactions of Definition 4.10, and the equivalence $\sim$ relating $t_1 \sim t_2$ iff both are Reduce transactions at the same agent p reducing the same goal (by identity, Definition 4.8) 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, or both are Cold-call transactions from the same p to the same q delivering the same message.

Reduce differs from LP in the use of a writer mgu instead of a regular mgu, and in the choice of the first applicable clause instead of any clause; Communicate communicates an assignment from its writer to its paired reader — within one agent, or across the two agents holding the pair's two sides; Cold-call transfers a term with its variables through the network streams, creating paired variables between previously-disconnected agents [53]. Every participant of each transaction changes state, so the P-closures of distinct transactions are disjoint and $\sim\uparrow P$ is an equivalence (Definition 2.9). Implementation-wise, if a goal A cannot be reduced now, but there is a readers substitution under which it can, the goal *suspends* on these readers and is rescheduled once any of them is assigned.

Definition 4.12 (Proper Run [53]). A maGLP run is *proper* if, at each agent p, a variable that occurs in G_p after a transition but not before it also does not occur in G_p at any earlier configuration.

Definition 4.13 (cGLP [53]). Given a GLP program M, *cGLP*(M) is the maGLP transition system over a singleton set of agents and M.

Cold-call requires two agents, so cGLP has Reduce and Communicate only; it is the single-agent semantics.

Proposition 4.14 (Monotonicity [53]). *In any proper run of cGLP(M), if unit goal A can reduce with clause C at step i, then either an instance of A has been reduced by step $j > i$, or an instance of A can still reduce with C at step j.*

5 Volition-Guarded GLP

Here we define volition-guarded GLP (vGLP), the extension of GLP (Section 4) with volitional procedures: a procedure may put a question to the person. Its goal receives an *interactive variable* as an added argument, which the program and the person write together, each the positions its type gives it, and a clause that reads the person's part reduces only after the person has answered. vGLP is typed as GLP is: the declaration of a volitional procedure carries its interactive type, and widget declarations say how interactive types are displayed (Section 6).

5.1 Volitional Procedures

Definition 5.1 (Volitional Procedure, Interactive Type). A *volitional procedure* p of arity n is declared `procedure` $(T)*p(T_1,\ldots,T_n)$., where $T_1,\ldots,T_n$ are the types of its arguments and T, in writer or reader mode as an argument type is, is its *interactive type*. An interactive type is built from constants, `String`, `Integer`, `Real`, `Key`, `Constant`, `Module`, named types so built, tuples and unions, and `Stream` or `List` as the whole type or as an argument of a tuple, never under another `Stream` or `List`, and a `Stream` only at a position the person writes; a named type in it is recursive only if it has no mode inversion [58], and it has no type parameter at a position the person writes.

A speech act forwarded with its provenance is signed again at each forward [16], so its type is recursive, and a request to forward it carries it. With no mode inversion, a recursive type is written whole by the program or whole by the person.

Definition 5.2 (Clause of a Volitional Procedure, Interactive Term). A clause of a volitional procedure p with interactive type T has the form $(A)*p(S_1,\ldots,S_n)$ `:-` G `|` B, where the *interactive term* A is a term of type T, possibly a variable but not the anonymous variable, and it is the guarded clause $p(S_1,\ldots,S_n,A)$ `:-` G `|` B (Definition 4.2), of arity n+1. A clause of a volitional procedure reads an argument at an input position, by a non-variable term at that position in its head or by a guard on it, only if every later clause of the procedure reads it too, and every call of a volitional procedure, in a clause body or in the initial goal, holds a writer at each output position.

Definition 5.3 (Ordinary Clause, vGLP Program). A clause is *ordinary* if it is not a clause of a volitional procedure. A *vGLP program* is a finite sequence of ordinary clauses and clauses of volitional procedures, each satisfying SRSW, a clause of a volitional procedure satisfying it as a guarded clause of arity n+1; the clauses for the same predicate form a *procedure*. The set of goals $\hat{\mathcal{G}}(M)$ of a vGLP program M includes all goals over $\hat{\mathcal{V}}$ and the vocabulary of M that satisfy SO.

The last argument of the guarded clause is matched against the goal's end of its interactive variable, and a goal of a volitional procedure of arity n is n-ary until it is asked and of arity n+1 after (Definition 5.8).

Here are vGLP fragments of the currency agent of the repository (Section 7), written by hand in this syntax, in which `...` elides arguments the clause carries through unchanged. The agent is a volitional procedure whose question is the stream of the person's requests, of a union type, an alternative per request, served by an ordinary procedure:

```
Friend ::= Key.
```

```
Amount ::= Integer.
Request ::= mint(Amount) ; pay(Friend, Amount) ; ... .
procedure (Stream(Request)?)*agent(...).

%% Ask the person for their requests, once, and serve them.
(Reqs)*agent(Id, ..., Holdings, Serial) :-
    serve(Reqs?, Id?, ..., Holdings?, Serial?).

%% Pay_{p,q}(k), guarded by p: k q-coins to q.
serve([pay(Friend, Amount)|Reqs], Id, ..., Holdings,
      Serial) :-
    ground(Id?), ground(Friend?), integer(Amount?),
    Amount? >= 1 |
    select_coins(Friend?, Amount?, Holdings?, Status,
                 Give, Rest),
    do_pay(Status?, Id?, Friend?, Amount?, Give?, Rest?,
           Reqs?, ..., Serial?).
%% A peer proposes a swap: spawn a responder, which asks
%% the person, and go on without waiting for a request.
on_act(swap(Want, Offered, Resp?), From, Id, Reqs,
       UserIn, ...) :-
    ground(Want?), ground(Offered?) |
    respond_swap(offer(From?, Want?, Offered?), Resp, Ans),
    merge(Ans?, UserIn?, UserIn1),
    serve(Reqs?, Id?, UserIn1?, ...).

%% yes accepts, no declines; the agent hears of either
%% on its own input.
decide_swap(yes, From, Want, Offered, Resp?, Ans?) :-
    accept_answer(From?, Want?, Offered?, Resp, Ans).
decide_swap(no, From, _, Offered, Resp?, Ans?) :-
    decline_answer(From?, Offered?, Resp, Ans).
```

The agent asks its question once and passes the stream to `serve`, ordinary GLP; the person answers one request at a time, and the clause of that request's alternative reads it at the head of the stream: the pay clause waits for a pay request. The clause of an arriving swap proposal reads no request, so it reduces without waiting and passes the stream on. The responder's question, `respond_swap` of Section 1, carries output as well as input: its interactive type is in writer mode, so its clause does not wait but writes the approval question at once, with the proposal at the position `Offer` for the person to see, and hands the person the writer at the position `YesNo?`; the clauses of `decide_swap` each wait for the person's answer, and a decline is an answer, `no`, taken by its clause. A clause whose interactive term is a variable reduces without waiting and passes the reader on, so the question stays open until the person answers it:

```
Status ::= sent.
Line ::= line(String, Status?).
Msg ::= msg(Peer, String).
procedure (Stream(Line)?)*chat(Peer?, Stream(Msg)).
(Ls)*chat(Peer, Out?) :- ground(Peer?) |
```

```
    send_all(Peer?, Ls?, Out).

send_all(Peer, [line(Text, sent)|Ls], [msg(Peer?, Text?)|Out?]) :-
    ground(Peer?) | send_all(Peer?, Ls?, Out).
send_all(_, [], []).
```

The chat input is one question whose answer is a stream, which the person gives one line at a time, and `send_all` is ordinary GLP, calling itself on the tail of the stream after each line. Each line has a position the program writes, its status: the person's answer hands the program its writer (Answer, Definition 5.7), and `send_all` writes `sent` into it, by head unification, as it puts the message on the network, so the line the person typed shows as pending until it is sent and as sent after. A question whose type has no position the program writes is bare, as the agent's request stream is.

5.2 Operational Semantics

Here we define vmaGLP, the operational semantics of vGLP, which extends maGLP (Definition 4.11) with the person of each agent: an agent's state adds the variables its person holds, and three transactions, each at one agent, add the person's part. What vGLP adds is thus local; the semantics is multiagent only as maGLP is, by Communicate and Cold-call, and Communicate delivers the program's output to the person as it delivers an assignment to a goal.

Definition 5.4 (vmaGLP Agent State). Given a vGLP program M, an *agent state* is a triple (H_p, G_p, σ_p) of a finite set H_p of typed variables *held* by the agent's person and of assignments *shown* to them and an asynchronous resolvent (G_p, σ_p) over M (Definition 4.11). A shown assignment is a record of what the person was shown, and its variables are not occurrences.

Definition 5.5 (Reduce and Communicate in vmaGLP). Reduce applies to a unit goal that is not an n-ary goal of a volitional procedure, and a clause of a volitional procedure is taken as its guarded clause (Definition 5.2). Communicate delivers an assignment $\{X? := T\} \in \sigma_p$ to a reader $X? \in H_q$ once $\hat{T}$, the term T with each reader replaced, repeatedly, by the term assigned to it in the configuration, has no reader, and then delivers $\hat{T}$: it removes $X?$ from H_q and adds the variables of $\hat{T}$, each with the type of its position in $\hat{T}$, and the assignment, shown, the assignments used in forming $\hat{T}$ being communicated with it; Reduce, Cold-call and every other Communicate leave the held variables unchanged.

Definition 5.6 (Ask). The *transactions on the person's variables* at an agent p are *Ask* of this definition and *Answer* of the next. Ask p: A is an n-ary goal of a volitional procedure q with interactive type T, X a writer of $\mathcal{V}_p$ occurring nowhere in the configuration, the *interactive variable* of the asked goal, of type T' where T is T' or $T'?$; the goal takes its end of X and the person the other: $(A', Y) = (q(S_1, \ldots, S_n, X?), X)$ where T is in reader mode, and $(q(S_1, \ldots, S_n, X), X?)$ where it is in writer mode. The asked goal keeps the identifier of A.

$$\frac{A = q(S_1, \ldots, S_n) \in G_p \qquad X \text{ fresh} \qquad (A', Y) \text{ as above}}{(H_p,\ G_p,\ \sigma_p) \ \rightarrow\ (H_p \cup \{Y\},\ G_p \setminus \{A\} \cup \{A'\},\ \sigma_p)}$$

Definition 5.7 (Answer). Answer p: $X \in H_p$ is a writer, and t a term of the type of X, ground but for variables of $\hat{\mathcal{V}}_p$ occurring nowhere in the configuration: a reader $Y_i?$ at each position the person writes of type `Stream(`T'`)` or `List(`T'`)`, unless the person ends the list there with `[]`, and a writer W_j at each outermost position the program writes; where X itself is of such a type, t is `[`e`|`$Y_1?$`]` with e such a term of type T', or `[]` for a list. The person holds Y_i and $W_j?$, each with the type of

its position in t.

$$\frac{X \in H_p \qquad t \text{ as above}}{(H_p,\ G_p,\ \sigma_p) \ \rightarrow\ (H_p \setminus \{X\} \cup \{Y_1, \ldots, Y_k, W_1?, \ldots, W_m?\},\ G_p,\ \sigma_p \cup \{X? := t\})}$$

Definition 5.8 (vmaGLP Transition System). Given a vGLP program M and agents $P \subset \Pi$, the *vmaGLP transition system* over P and M is the transactions-based multiagent transition system (Definition 2.10) over P, local states being agent states, initial local state $(\emptyset, \{a_0\}, \emptyset)$ with a_0 as in Definition 4.11, the transactions being the Reduce, Communicate and Cold-call transactions of maGLP as Definition 5.5 adapts them and the transactions on the person's variables, Ask and Answer (Definitions 5.6 and 5.7), read as those of Definition 4.10, and the equivalence $\sim$ below. $\sim$ is the partial equivalence whose domain is the Ask, Reduce, Communicate and Cold-call transactions, relating two Reduce, Communicate or Cold-call transactions as maGLP does, the person counting as a goal, and $t_1 \sim t_2$ iff both are Ask transactions at the same agent p asking the same goal (by identity, Definition 4.8).

Every participant of each transaction changes state, so the P-closures of distinct transactions are disjoint and $\sim\!\uparrow\!P$ is an equivalence (Definition 2.9). Ask is the pre-reduction of a goal of a volitional procedure. Before the goal can reduce, it receives its end of its interactive variable as its added argument: the reader where the person writes the variable, and the writer where the program does. The person holds the other end. The semantics fixes no moment for Ask other than before the goal's reduction; the implementation asks at the goal's first reduction attempt (Section 6).

Answer is the person's act: the program reads an answer only through its reader, and the person performs every answer. Answer lies outside the domain of the equivalence and carries no liveness obligation, since the machine cannot compel the person. An answer is an assignment and cannot be withdrawn; the person edits it in its construct before submitting it (Section 6).

An interactive type carries output to the person as well as input from the person. Its moded type [58] says which positions the program writes and which the person writes, and each side, writing a position, hands the other the variables of the positions below it that the other writes. Communicate delivers the program's output to the person whole, every position the program writes in it written, with the writers the person answers, and keeps it in H_q as shown. So the person holds the writer of an input only with the whole output around it, and sees what they are asked about before answering (Section 6). An answer is ground at the positions the person writes, but for a stream or a list. An answer of type `Stream(`T`)` is given one element at a time, each element leaving the person the writer of the stream's tail as a new question, and an answer of type `List(`T`)` likewise until the person ends it. A stream or list inside an answer is left to the person as a new question, answered element by element.

Proposition 5.9 (Conservativity). *For a GLP program M, the runs of the vmaGLP transition system over P and M, with $(\emptyset, G_p, \sigma_p)$ identified with (G_p, σ_p), are the runs of the maGLP transition system over P and M, and the same runs are correct in both.*

Proof. M has no volitional procedure, so in a run no goal is asked and the person holds no variable: Ask and Answer have no instances and every H_p is $\emptyset$, so no reader is held. Reduce, Communicate, and Cold-call are those of maGLP with H_p unchanged, and $\sim$ is its equivalence, total on them. □

Definition 5.10 (Proper Run, adapted from [53]). A vmaGLP run is *proper* if, at each agent p, a variable that occurs in G_p after a transition but not before it also does not occur in G_p or in H_p at any earlier configuration, and the clause each Reduce renames is renamed apart from every agent's

goals, held variables and readers substitution, in the configuration it reduces and in every earlier one.

Definition 5.11 (vcGLP). Given a vGLP program M, *vcGLP*(M) is the vmaGLP transition system over a singleton set of agents and M.

Cold-call requires two agents, so vcGLP has Reduce, Communicate, Ask and Answer only; it is the single-agent semantics.

Proposition 5.12 (Monotonicity in vcGLP). *In any proper run of vcGLP(M), if unit goal A can reduce with clause C at step i, then either an instance of A has been reduced by step $j > i$, or an instance of A can still reduce with C at step j.*

Proof. In vcGLP a goal of a volitional procedure reduces only once asked, as a GLP goal with the guarded clauses of its procedure, and an answer reaches it as a communicated assignment, so Proposition 4.14 holds verbatim. □

Definition 5.13 (Question, Open, Closed, Volition). In a configuration of the vmaGLP transition system, a writer $X \in H_p$ is a *question* of p's person: an interactive variable the person writes, introduced by an Ask, a writer inside the program's output delivered to the person, or the writer of a stream or list that an Answer leaves to the person, the tail of the stream or list answered or one inside the answer; it is *open* if its reader $X?$ occurs in the configuration, and *closed* otherwise. A *volition* of p's person is an Answer transition of p: the person's answer to a question.

An open question is what the machine puts to its person: *what is your answer?* A question is answered by Answer, which takes it from the person's variables; it closes when its reader no longer occurs, the program having dropped the reader by the anonymous variable at an input position [53]; in writer mode the program so closes a question inside its output and, where the type provides for it, writes more of the output — a status that the widget reads, say. A closed question asks nothing more of the person: an answer to it reaches no goal. How questions reach the screen and answers return is the interface's task (Section 6).

5.3 Volitional Soundness

The machine never answers for the person: a question is assigned by its person's answer and by no transition of the machine.

Theorem 5.14 (Volitional Soundness). *Let $c^1 \to c^2 \to \cdots$ be a proper safe run of the vmaGLP transition system over P and M, $p \in P$, and X a question of p's person in c^i. Then the writer X occurs in c^i in H_p and nowhere else, and a transition $c^i \to c^{i+1}$ that adds an assignment $X? := t$ to a readers substitution is the Answer of p to X.*

The proof is in Appendix B. Every reduction that reads an answer therefore reads a term the person gave. The theorem says who gives an answer. The person sees a whole proposal before answering it, since a question reaches its person with its output written (Definition 5.5). Once answered, a goal is a GLP goal, and a correct run reduces it (Proposition 5.15).

Proposition 5.15 (Answered Goals Reduce). *Let r be a proper correct run of vcGLP(M) and A an asked goal in it whose question the person answers with a term on which some clause of A's procedure reduces. Then an instance of A is reduced in r.*

Proof. The answer is an assignment $X? := t$ in the readers substitution with $X?$ in A; the Communicate of it to A is in the domain of the equivalence and enabled until taken, so r takes it, and A then holds t. From there A is a GLP goal with the guarded clauses of its procedure, and a

clause reduces it on t; by Monotonicity (Proposition 5.12) every clause that succeeds on A goes on succeeding until an instance of A is reduced, so the first clause that succeeds on A, which is the clause of its Reduce, changes only to an earlier one, finitely often; from some configuration on it is one clause, the Reduce class of A with that clause is enabled in every configuration, and r takes it. □

The person's requests are served in the same way: in the agent of Section 5, a request at the head of the stream is read by the clause of its alternative, which comes before the message clause, so the next reduction of `serve` takes it; a clause placed before the request clauses that reduces without reading the stream could put a request off while its trigger keeps arriving, and the agent goals of Section 5.4 place none, reading every trigger from one fair merge.

5.4 Compiling the Canonical CVA Platform into vGLP

A CVA platform specifies a set of app transactions (Definition 3.4), possibly infinite and written in no syntax, and no program implements every such set. Here we compile into vGLP the canonical platform of Section 3.2 of every effective local specification, with a computable introduction rule or none (Definition 3.18), one goal per agent, and prove the compilation correct.

Definition 5.16 (Effective Local Specification). A local volition-guarded specification is *effective* if its machine states are finite terms, its guarded parts, which enter the approval questions, are terms of an interactive type (Definition 5.1), each with a *request name*, a term of an interactive type in which no `Stream` or `List` occurs, and its unguarded parts are finite terms; the parts of a class are finitely many and computable from any one of them, $Q([t])$ from any part of $[t]$, and $G([t]_j)$ and $o([t]_j)$, where it has one, from $[t]_j$; $f_p^{[t]_j}(s)$ is computable from $[t]_j$, p and s, and whether $s \in dom(f_p^{[t]_j})$ is decidable from them; for every agent p, machine state s and request name N, the guarded parts $[t]_j$ with no opener of the classes of the parts named N, with p a guard of $[t]_j$ and $s \in dom(f_p^{[t]_j})$, are finitely many and computable from p, s and N; the guarded parts with an opener named N have one opener, computable from N, and over each finite $P \subset \Pi$ the classes of those that a guard p guards are finitely many, the request ρ_N of p, whether a class is in ρ_N being decidable from a part of it and N, and whether p guards one of those parts with $s \in dom(f_p^{[t]_j})$ decidable from p, s and N; for every agent o, machine state s and request name N, the guarded parts $[t]_j$ named N with $o([t]_j) = o$ and $s \in dom(f_o^{[t]_j})$ are finitely many and computable from o, s and N, and, where there is none, some guarded part named N with opener o is computable from o and N; and for every agent p and machine state s, the unguarded parts $[t]_j$ with $o([t]_j) = p$ and $s \in dom(f_p^{[t]_j})$ are finitely many and computable from p and s.

The conditions and effects of the canonical platform of an effective specification are computable and terminate: they test and apply the specification's functions and the guard sets of its parts. A name of a transaction is of an interactive type: a guarded part is a term of one, and an attempt of an unguarded part, which may hold a list, as a delivery holds its members, is named by its initiator and number.

Triggers. The compiled agent takes each app transaction of the canonical platform on a *trigger*: **Request** on its person's answer `propose(N)` to a request question, N the request name of a guarded part, on which it takes, among the **Request** of $\{[t]\}$ to itself for each class $[t]$ of a guarded part with no opener named N that it guards and is ready for, and the **Request** of ρ_N to the opener of the parts with an opener named N, the one it has taken least recently; **Start**, **Tally**, **Yield** and **Outcome** on the arriving message; **Answer** on its person's answer to the *approval question* that **Start** raises, one per attempt, asked by a goal that shows the attempt and takes yes or no; and

Propose, **Vote**, **Commit**, **Wound** and **Introduce peers**, its *own transactions*, one at a time, each on an *own trigger* that the agent writes on a stream merged fairly with its other triggers, when it starts, after another trigger and after each own transaction it takes, until none holds. So the person answers two kinds of question: *request questions*, each taking one request, and approval questions; the agent's own key and the persons it knows are in the view it writes. A request is taken when it is read, or held while the agent is not free for its classes and taken when it is, or dropped, with a notice, where the agent can carry out no part it names; once taken it is an approval, as an answer yes is, which stands until the transaction is taken or refused, and the person sees their pending approvals in the view; the agent proposes, and proposes again an attempt that yields, with no further question to its person. A machine that cannot carry out a transaction its person approved refuses it; when a refusal ends an approval, the person who made it is told who refused the transaction and why, declined or unable. The else-branch of **Answer** is taken on the person's answer no, where CVA takes it on the absence of the will, so an approval question waits for its person.

The compilation. Given an effective local volition-guarded specification and its canonical platform Θ, the *compilation* $\lceil\Theta\rceil$ is the following vGLP program, standard stream techniques used without comment [53]. The *agent* calls the agent goal, which asks its person for requests by *request questions*: each is a goal of a volitional procedure whose question, of the request type `Request?`, takes one request, of a type with the alternative `propose(`N`)` for the request name N of each guarded part ; the agent goal asks one when it starts and the next when it reads the answer to the last. On the program's output to its person (Section 6.3) the agent goal writes its *view* `view(`k`, `s`, `K`, `A`)`, its own key k, the machine state s of its app state, the persons K it knows and the approvals A its person made that are pending, each by its request name, a request's own and an answer's that of its attempt's part, when it starts and whenever any changes; and, when an approval its person made, by a request or by an answer yes, ends by a refusal, a *notice* `refused(`N`, `q`, `d`)`: N the request name of the part of the refused attempt, q its refuser and d the reason, from the abort outcome, or from the vote where the agent's own vote no refused it. The compilation defines `Output` as the type of the views and the notices, and prints the platform's three widget declarations (Definition 6.5, Section 6.2): for the request type, for the approval type and for `Stream(Output)`.

The *agent goal* is an ordinary procedure holding the *agent state* — the app state, initially $(s_0, 0, \bot, \emptyset)$; a count of the own transactions it has taken, initially `0`; the peers `hello` has been sent to and received from; the starts it holds waiting; the requests it holds; the requests it has made, in the order it last made them; a writer for each open approval question, by which it closes it; the count at which each class of its own transactions began to wait; and the writer of its stream of *own triggers* — and reading one input stream, the fair merge (Example 4.3) of the answers to its request questions, the messages arriving on the network, the answers to its approval questions, its own triggers and, where the contract requires the day, the calendar, on each date of which it replaces $today(d)$ and puts up again the starts and requests it holds.

The agent goal has one clause per kind of trigger, which reads the trigger at the head of the stream, computes in its body the condition of the transaction the trigger puts up, on the state it was called with, and calls a *dispatch procedure* with its value, each of whose clauses calls the agent goal again. A dispatch clause that *takes* a transaction carries out its effect: the new app state; the messages it sends, appended to the network output stream; for a **Start** that records a question, an *approval question*; and the closing of the open approval question of an attempt the transaction closes; and, where the machine state changes, it writes the new view on the output. An approval question is asked by a volitional procedure whose question shows the attempt and takes yes or no; the procedure also takes the reader of the writer by which the agent goal closes it, and its one

clause hands that reader and the reader of the answer to an ordinary goal, which passes the answer on where it comes first and, where that reader is assigned first, drops the answer's reader, which closes the question (Section 5).

On the answer `propose(`N`)` to a request question the dispatch takes, among the **Request** of $\{[t]\}$ to itself for each class $[t]$ of a guarded part with no opener named N that the agent guards and is ready for, where it is free for $[t]$ and no such request is pending at it, and the **Request** of ρ_N to the opener of the guarded parts with an opener named N, where the agent is free for every class of ρ_N, guards one of those parts it is ready for, and ρ_N is not pending at it, the one it has taken least recently, where there is one; where the agent is not free for a class of such a **Request**, it holds the request and puts it up again after every dispatch; and it drops the request where the agent is free and none is enabled, with a notice `refused(`N`, `p`, unable)` where the agent is ready for no part named N that it guards or opens; the message of the **Request** carries N for ρ_N. On a message request(ρ, k) it holds the request until it deletes it, at the refusal of an attempt naming it or at a commit of its class. On a message start(α) it takes **Start** where the start does not wait (Definition 3.11) or α is closed at the agent, and otherwise holds it waiting; on vote(α, y) it takes **Tally**, on yield(α) **Yield**, and on outcome(α, z) **Outcome**. On the answer to the approval question of α it takes the then-branch of **Answer** on yes and its else-branch on no, where α is a question pending at the agent and unanswered, and drops the answer otherwise.

The agent goal takes its own transactions one at a time, one on each own trigger it reads. It writes an own trigger when it starts, after each other trigger it reads unless an own trigger it has written is still unread, and after each own transaction it takes, so that it reads its other triggers between its own transactions and writes none once none holds. On an own trigger it puts up the **Propose** of each unguarded part the agent opens, and of each guarded part with no opener that it guards of the class of an approval pending at it, named by that approval where the approval is a request, where its machine state is in the part's domain, or, where it is in the domain of no such part, of one of them, which it refuses at once; for each request it holds, the **Propose** of each guarded part named by the request's name that it opens and whose domain holds its machine state, naming that request, or, where there is none, of one such part, which it refuses at once; the **Vote** on each answered question pending at it, the **Commit** of each attempt it initiated with a yes of every other participant recorded, and the **Wound** of each start it holds waiting, or answered question, blocked by an attempt over which it has priority, and the **Introduce peers** of each pair the introduction rule gives it at its machine state; the dispatch takes, among those whose condition holds, one of the class that has waited longest — since it was last taken or, never taken, since its condition was first found holding — ties broken in a fixed order, and counts it; where none holds, it takes none. After every dispatch that changes the app state or consumes a start the agent goal puts up again the starts and the requests of its person it holds.

The compilation is given, with the specification, its *acquaintance parts*: guarded parts with two guards and no opener whose guards need not know each other, for a contract the parts of a *acquaintance schema*, one with two roles, both guarding, that requires at neither role an atom naming the other. GKK's *acquaint* is one, by which two persons come to know each other (Section 7); CSSN's *enter* is another, by which a parent, holding both phones, sets up their child's phone, the bootstrap of parenting [54]. The request of a acquaintance part is also a cold call. On the request `propose(`N`)`, N the request name of a acquaintance part with a peer Q, the agent goal, besides what it does on any request, sends Q `hello` carrying N and records Q as sent to, unless Q is known or already sent to. On an arriving `hello` from Q it records Q as received from and, unless it has sent Q a hello, asks its person the approval question of the clause of N, with its pragma, by a goal that shows Q and takes yes or no: the person's yes is their request `propose(`N`)` with Q, which the agent goal reads as any request, and on their no it does nothing. On reading an introduction

intro(Q) it sends Q `ihello`, unless Q is known or it has sent one, and an arriving `ihello` it records likewise. A peer is *known* at the agent when `hello`, or `ihello`, has been sent to it and received from it, and a condition's test that a peer is known is of these; when a peer becomes known, the agent goal writes the new view on the output.

The reduction of a dispatch clause that takes a transaction, or the else-branch of **Answer**, is that transaction. Discover is the sending of the second of the two hellos, and Communicate is the agent goal's reading of a message the network streams carried to it. Every other reduction is a stutter.

Definition 5.17 (Platform Mapping). Given an effective local volition-guarded specification, its canonical platform Θ, $P \subset \Pi$ and a configuration of the vmaGLP transition system over P and $\lceil\Theta\rceil$, the *platform mapping* σ yields the configuration of $\mathcal{P}$ with, at each agent p: $known_p$ the $q \in P$ such that p and q have each sent the other `hello`, or each sent the other `ihello`; o_p the app messages on p's network output stream not yet read by their recipient's agent goal; i_p the app messages p's agent goal has read and not yet dispatched, the starts it holds waiting, the requests it holds, and the introductions it has read of peers with which it has not exchanged `ihello`; a_p the app state the agent goal holds or, between the reduction of a dispatch clause and the agent goal's next call, the one its effect yields, the atom $today(d)$ the calendar writes left out.

Definition 5.18 (Platform Mapping: Volitional State). At each agent p, the platform mapping σ yields the volitional state holding p's approval class on a class $[t]$ while the agent goal has read a request `propose(`N`)` that puts up the **Request** of $\{[t]\}$, N naming a guarded part of $[t]$ that p guards, and has not yet taken or dropped it, or reads the answer yes to the approval question of an attempt of $[t]$ — from its reading until its dispatch — the condition of the **Answer**, but for its test of known peers, holding at p's machine state; holding p's request class on ρ_N while the agent goal has read a request `propose(`N`)` that puts up the **Request** of ρ_N, of two or more classes, and has not yet taken or dropped it; and the class of Discover(p, q), $q \in P$, while a request of p's person of a acquaintance part with q is pending at p and $q \notin known_p$.

Theorem 5.19 (Compilation of the Canonical Platform). *For every effective local volition-guarded specification and $P \subset \Pi$, the platform mapping is a correct implementation of $\mathcal{P}$, the transition system of its canonical platform with a computable introduction rule or none over P under its fairness condition (Definition 3.22), by the vmaGLP transition system over P and $\lceil\Theta\rceil$, under the fairness condition that a run is proper, no approval question stays open in it, its calendar does not stop where the specification requires the day, and its image under the platform mapping is connected, unobstructed and steady.*

The proof is in Appendix B. The condition on the person concerns the approval questions: the request question need not be answered, since a request not made wills nothing, but an approval question left open keeps **Answer** from being taken by either branch; an approval question whose attempt closes first is closed by the program. The condition on the run is that of Theorem 3.24 but for strong fairness, which the compilation secures (Section 6.4). Completeness is not claimed: the compilation takes the agent's own transactions as soon as their conditions hold, in a fixed order, which $\mathcal{P}$ does not require.

Grassroots. The canonical platform is volitionally grassroots (Proposition 3.6). The *protocol* of its compilation assigns to each $P \subset \Pi$ the vmaGLP transition system over P and $\lceil\Theta\rceil$, its local states over P being the agent states in which every agent identifier is of an agent of P, and its correct runs being those that are correct and proper.

Proposition 5.20 (Grassroots Compilation). *For every effective local volition-guarded specification with acquaintance parts guarded by every two agents, and computable introduction rule or none, the protocol of the compilation of its canonical platform is grassroots (Definition A.3).*

The proof is in Appendix B. vmaGLP has no guards, so the stronger property, volitionally grassroots, does not apply to it.

Speech-act schemas. A contract written as speech-act schemas [16] is the layer above the transactions in Figure 1. A compiler written in GLP takes such a contract, whose roles are party roles or listed roles [16], checks that it is a local and effective specification, and prints the compilation of its canonical platform in vGLP, in the form of this section. It does so in one pass: it produces neither the volition-guarded transactions the contract specifies nor the canonical platform, and the program it prints carries the schemas and computes the condition and effect of each transaction from them as it runs. It compiles the social graph and the social network, GSG and GSN [16], and the currencies, GC [49] (Section 7), and the test suite runs each compiled platform on GLP with scripted persons. A vote unable of a compiled program carries the atoms the schema requires at the voter that its state lacks, and those it forbids that its state has, and the notice shows them.

Speech-act schemas as local specifications. A schema names its *roles*, the parties to the act, marks those whose person must will the act, and lists at each role the *atoms* — facts in that party's state, as $friend(q)$ in the state of p — that the act requires, forbids, deletes and adds there; a *schema-binding* is a schema with its roles bound to people, and its other variables to people and to speech acts, a speech act being an utterance or a speech act, signed; a signed term $sgn(y, i)$ in an added atom is the speech act bound to y, signed by the person in role i; and a *name term* $\langle u \rangle \cdot y_1 \cdots y_n$ is the *community name* of the person bound to u followed by the speech acts bound to $y_1, \ldots, y_n$, as $\langle a \rangle \cdot n$ names the group n led by a [16]. A state is a finite multiset of atoms, so a predicate of arity zero carries a count; a contract opens by declaring its names, as "Alice, Bob and Carol are persons", and distinct roles are bound to distinct agents, unless the clause says of two names "possibly one person" and neither role adds or deletes an atom naming the other, when the two may be bound to one agent; a listed role and the role whose agent lists it, and two roles so bound to one agent, are carried out together, the atoms at both required, forbidden, deleted and added at that agent [16]; and a predicate of arity one that the contract declares *reflexive* holds at every person of that person. A *listed role* $\pi[\rho(*, u_2, \ldots, u_r)]$, π a party role, its *base*, has as its people at a schema-binding the persons q with $\rho(q, u_2, \ldots, u_r)$, so bound, in the state of the person in π [16]. Speech-act schemas allow numbers [16]: an argument of an atom may be a number or a *numeric variable*, a schema may carry a *constraint* on its numeric variables, and an atom it adds or deletes may carry a *multiplicity*, a number or a numeric variable, for that many copies of it. A numeric variable that occurs only in atoms the schema requires or deletes at one role, and in the constraint, is *local* to that role; a constraint is local to the role whose local variables it names, and to every role where it names none. Every other numeric variable is bound by the schema-binding, as a speech-act variable is, and a request names it, so the request form takes it. A contract may name *environment predicates* [16], whose atoms are over values the platform keeps current, which no schema adds or deletes and a schema may require or forbid at a role and constrain a numeric variable by; the one so far is the day, $today(d)$, which the compiled agent realises by a *calendar*: it reads, with its triggers, a stream of dates the device writes, one a day, and holds $today(d)$, d the last date read, beside its app state, as GB's *advance* requires $date(t)$ and $today(d)$ with $t < d$ [55], so that it is proposed once a day; what the first date is, the contract says. The specification has no day, so $today(d)$ is no part of the machine state the platform mapping reads, a part that requires it being enabled in the specification whether or not the day has come; the results of this section therefore hold for a contract that requires the day under the further condition that the calendar does not stop. The classes are those of the contract's equivalence, generated by relating two transactions of the same participants when one schema-binding determines both [16], and each schema-binding determines a part $[t]_j$ of a class for each list L of the people of its listed roles: the set of transactions it determines with L in

those roles, with $f_p^{[t]_j}$ defined at d_p iff the schema-binding has the atoms it requires at p in d_p, with the atoms of the reflexive predicates naming p and the variables local to p's role bound by them in one way, those it forbids there not, a constraint local to the role holding under that binding, and, for p the person in the base of a listed role, p lists in d_p exactly the people of that role in L, and yielding d_p less the deleted atoms plus the added ones, as multisets, under that binding, a multiplicity k deleting or adding k copies, and with the people in its marked roles as its guard set; its request name is the schema-binding. Such a contract is a local specification: unfriending, whose schema determines the same transactions at the binding of either party first, is one class with two parts, one guarded by each party. The opener of an unguarded part is the least, in the order of identifiers, of the people in the first role of the schema-bindings that determine it, and so is the opener of a guarded part determined by a schema with a listed role whose base is not a guarding role, where that person is not a guard of the part; the contract is a local specification where every such schema has one guarding role or none, every other guarded part of the class that a guard of such a part guards is determined likewise, with the same opener, and a constraint names the local variables of one role. The contract is effective (Definition 5.16) where, moreover, every variable of a schema that occurs in an atom it requires or forbids, but a numeric variable local to a role, also occurs in an atom it deletes or adds, a variable inside a signed term or a name term included, so that a class has finitely many parts, and the atoms each unguarded schema requires at its first role name every other role and every variable of its deleted and added atoms, or the constraint gives the variable finitely many values once those are bound, so that a state, a finite multiset of atoms, admits finitely many unguarded parts its holder opens; and the listed roles of a schema have one base, which is its only guarding role or, if it is not a guarding role, its first role, so that the agent that proposes a part the schema determines, its guard or its opener, reads the people of those roles from its own state. The introduction rule of a contract (Definition 3.18) gives each person p, for a part determined by a schema-binding in which p's role requires an atom naming an agent c bound to a variable and the person q in another role gains an atom naming c, the pair $\{q, c\}$: at the commit p, who knows both, introduces them to each other, so that the machine of a person who comes to know an agent can reach it; under GKK:3 (Section 7) Alice introduces Carol and Bob. A contract may moreover name a predicate of arity one its *introduction predicate* e: its introduction rule gives each person p every two persons $q \neq r$, both other than p, with $e(q)$ and $e(r)$ in p's state, so the persons p names by e come to know each other.

The contract's words. Beside its clauses and its schemas a contract carries the words its app shows, its *pragmas* [16], and the compiler takes them into the platform it prints, so the app is compiled from the contract (Section 8). The platform's own words are four: the answers yes and no of an approval question, and the reasons declined and unable of a notice. Every other word the app shows is the contract's.

The first pragma names a tab of the app and what it lists, a predicate of the contract, named in the line by its name, an underscore read as a space. The tab lists the atoms of that predicate in the state; a row shows the atoms of the state naming its item, with its count where the contract adds or deletes the predicate with a multiplicity, the view carrying the whole state.

Each other pragma names a clause and gives, on the clause's line, the person who proposes, by the clause's name for them, and then, for each act they propose, its *button*, in brackets, bearing the act's name, and its *fields*, in angle brackets: a button is pressed in the row of a tab's entry, a person who takes a role of the act or an item the act names, and the fields are what the proposer fills besides, one for each other role of the act and each variable of its schema after the roles, in the schema's order: a role is picked among the persons the viewer knows; a variable is picked among the items the viewer holds where the proposer's role requires it held, and typed otherwise, an

agent by key (Section 6.2). A line follows for each other person who proposes, likewise, and for each person asked, their name, a colon, and the *question* put to them; a question line belongs to the button line nearest above it, so one person's acts share a line only where no question follows them. A button is offered only to the person named before it. A guarded act has a button, since a person proposes it; an unguarded one is proposed by its opener's machine and has none.

A name in a pragma is the schema's role of that name, the schemas naming their roles as the clauses do [16], and a question names the person asked 'you'. The platform fills the names from the binding of the attempt it asks about, and shows after the question, by the names of their fields, the values of the binding the question does not name, so that the person sees the whole proposal. A person no line names is never asked, so an act is proposed as the person named before its button, by whoever proposes it, and the persons named before the questions are asked: GSG's Friendship is proposed as Alice, and Bob is asked "Alice asks to be your friend"; GSN's Leaving gives Leave to Bob, a member, and Close to Alice, the leader, and asks no one (Section 7). A question line naming the proposer is text of the request form, shown with the button: the request is the will, elicited once.

The compiler carries the pragmas into the platform it prints: the tab's name and its predicate into the view the agent writes; the buttons and the fields into the widget declaration it prints for the platform's request type (Section 6.2); and a question into the approval question that **Start** raises at the role it is put to, and that a `hello` asks. A contract that extends others has their pragmas with its own, and its app is compiled from all of them; where it writes a schema anew for a clause it inherits, its line for that clause replaces the inherited line.

6 Elicitation: Implementing vGLP by GLP

Here we implement vGLP by GLP (Section 4). A question stays open until its person answers it or the program drops its reader (Definition 5.13), and only the person's answer reaches the goal that asked it, so the implementation requires elicitation: offering the person, for each open question, the means to answer it with a term of its type, and showing them what the program writes for them to see. We derive the elicitation constructs from the interactive types, and compile vGLP onto GLP.

6.1 The Elicitation Construct

A question's type fixes what the person is shown and what they may answer: the program writes the positions the type gives it and the person the others, and a question reaches the person with the program's output in it written whole (Definition 5.5). A *widget* renders a type: it shows what the program writes, takes the person's input, and forms from it a term of the type. Widgets are associated with types by declarations (Section 6.2), and the constructs they render are standard [40, 60].

Definition 6.1 (Construct, Submission, Complete Widget). The *construct* of a question X at agent p is the widget associated with the interactive type of the interactive variable that is X or whose term holds X, presented at p while p's person holds X. It shows the assignments shown at p (Definition 5.5) to the readers of the interactive variable's term, as they are shown, and forms from the person's input a term of the type of X; a *submission* of it is such a term, which the person gives as their Answer to X. Where X is of type `Stream(`T'`)` or `List(`T'`)`, a submission gives one element and leaves the tail an open question with the same construct, or, for a list, ends it. A widget of T is *complete* if, for every question whose construct it is, every answer the Answer transition allows for the question (Definition 5.7) is formed by some input.

The person edits the input of a construct until they submit it; they give a submission as their Answer, and an answer cannot be withdrawn (Definition 5.7).

Definition 6.2 (Offered). A term t is *offered* for a question X at p if the construct of X presented at p forms t, as a submission, from some input.

Two widgets recur, and we name questions after them, informally. A question of a tuple type is a *compose form*, whose fields are the tuple's arguments, and a question of a union of tuple types is a menu of such forms: the volition originates with the person, who supplies the transaction's parameters — mint, pay, post, offer friendship. A question whose type shows content the program writes and takes one of a few constants is an *approval question*, whose content is the offer and whose buttons are the constants: the responder's `Approval` of Section 1, with the proposal shown and the buttons yes and no.

Theorem 6.3 (Elicitation Completeness). *In every configuration of a safe run of the vmaGLP transition system over P and M and every $p \in P$, a term offered for a question X at p is an answer to X that the Answer transition allows; and if the widget of the construct of X is complete, every answer the Answer transition allows for X is offered.*

The proof is in Appendix B. The first half follows from typing: a widget forms only terms of its type, so the validation of an input is its typing. The second is a requirement on widgets, which the default widgets below meet, the picker excepted (Proposition 6.7). A construct family draws the widgets (Remark 6.4); nothing below the widget is proved.

6.2 Widgets

User-interface design and specification languages are a large field [1, 2, 5, 29, 37], and any of them could be brought to bear here. For the purpose of this paper we present the basic types and their widgets, and the declaration that associates a widget with a type.

Remark 6.4 (Construct Families). A question's type fixes a construct's semantic content — what it shows, what it takes, and the answer each submission forms — and nothing of its layout. A *construct family* is a rendering of the widgets that preserves this content; the theorems of this section, Theorem 6.3 and Theorem 6.10 below, hold for every construct family whose widgets are complete, so families are interchangeable and each person may choose theirs — the same program rendered plainly for one person and child-friendly for another.

Definition 6.5 (Widget Declaration). A *widget declaration* has the form T `=::=` W, where T is a moded type and W a widget of the construct family that renders the program (Remark 6.4). Widget declarations are scoped as type declarations are [58]: a declaration at the root holds for every program, one in a module holds in that module, and a local declaration overrides a global one.

Definition 6.6 (Default Widget). The widget of a question's type is the one named by the widget declaration in scope where the volitional procedure asking the question is declared or, where none does, its *default widget*, built from the type's constructors:

- a constant: a button; a union of constants: a row of buttons; a position of type `Constant` or `Module`: a text field;
- `String`, a number (`Integer` or `Real`) and `Key`, the agent identifier: a text field, a number field and a peer field;
- a tuple: a form headed by its functor, with one field per argument, each labelled by the name of its declared type, a field of a recursive type drawn when the person opens it; a stream or list inside a form the person writes: submitted as a separate question and drawn under the form; any other union: a menu, one item per alternative, each drawn by its widget;

- a value the program writes: shown and not edited; a tuple of two arguments, a list the program writes and a position of its element type that the person writes: a picker in place of a form, offering the elements of the list;
- a stream the person writes: the widget of its element type, kept open, one element per submission; a list the person writes: the same, with a button that ends the list; a stream the program writes: a thread.

A value the program writes to its person outside a question is drawn likewise, by the widget of its declared type.

The mode matters: the same `Stream(String)` is an input box when the person writes it and a thread when the program writes it. The widget holds the other end of the question, so its type is the dual of T: it writes where the program reads and reads where the program writes. The values of a primitive type are those the runtime represents, and a field takes each of them. Every interactive type has a default widget, so a program with no widget declarations still renders; the default widgets are complete for their types, except the picker, which offers the elements of its list, where the declared widgets that draw their fields and items are complete:

Proposition 6.7 (Default Widgets). *The default widget of every interactive type is complete, but for a picker, where the declared widgets that draw its fields and items are complete.*

Proof. By induction on the answer, a finite term. The answers the Answer transition allows for the question (Definition 5.7) are the terms of T' ground at the positions the person writes, with a fresh reader at each stream and list position and a fresh writer at each outermost position the program writes, and, where T' is `Stream(`T''`)` or `List(`T''`)`, the terms `[`e`|`Y`?]` with e such a term of T'', and `[]` for a list. A constant, or a union of constants: each answer is a constant, formed by its button. `String`, `Integer`, `Real`, `Key`, `Constant` and `Module`: each answer is a value of the type, formed by its field. A tuple: each answer is a tuple whose arguments are answers for their types at the positions the person writes, formed by the form's fields by induction, a field of a recursive type once the person opens it, a fresh reader at a stream or list argument, which the form submits as a separate question, and a fresh writer at a position the program writes, which the form shows and does not edit; the form forms the tuple. Any other union: the menu selects the alternative, and the alternative's widget forms the answer. A stream the person writes: each answer `[`e`|`Y`?]` has e an answer for the element type, which holds no stream or list, formed by the element's widget by induction, and the widget forms the fresh tail; a list likewise, and `[]` by the button that ends it, whose grant is distinct from every element. A named type, recursive or not, is its definition. A field or an item drawn by a declared widget forms each answer of its type, that widget being complete. A question inside the program's output is answered by the field of its position, the default widget of its type, by the same induction. A picker forms only the elements of its list. □

A declared widget chooses within the construct family, and Theorem 6.3 holds for it where it is complete. It is drawn with the default widget of its type beside its name, from which it reads the type's constructors, so that a menu declared once lists the alternatives of the request type of each program it is declared for; a construct family that has no widget of that name draws the default widget. A declared widget is drawn also with the last value drawn on construct `0` (Section 6.3), and is drawn again when that value changes, so that a request form can show the state the program last wrote to its person. A field of a form, or an item of a menu, is drawn by the widget of its type, declared or default. In the grassroots app (Section 8), W names a Dart widget: the construct family draws it, and its type fixes the terms it forms.

The words a platform shows. The words a platform compiled from a contract shows are the contract's pragmas (Section 5.4), and they reach the person two ways. A question put to a guarding

role names the roles of its schema, so the platform writes it into the approval question with the persons of the attempt and the values of its binding, and its widget draws it as it draws any output of the program. A button and a field's name are the same for every question of their type, so the compiler writes them into the widget declaration it prints for the request type, and a widget term carries a label beside each constructor of Definition 6.6; a construct family draws the label it is given, and the name of the type where it is given none.

The compiler prints three widget declarations for a platform: for its request type, a menu of the buttons and the fields of the contract's acts; for its approval type, the question with the buttons yes and no; and for `Stream(Output)`, the type of the program's output, a widget that draws the last view written and the notices written after it.

The request widget offers an act only where the person's machine can propose it (Section 3.2), to the person the contract names as its proposer (Section 5.4). A button is drawn in the row of a tab's entry, a person who takes a role of the act or an item the act names, with the act's fields beside it: in a row of the tab whose predicate the act requires or deletes at the proposer's role naming the entry, and only where the viewer's state holds the atoms the act requires at the proposer's role and lacks those it forbids there. The other roles' preconditions are checked at their own machines, but for what the schemas prove of them: where every schema that adds an atom naming x at a role requires or adds at x's role an atom that no schema deletes, the viewer holding the first shows that x holds the second, and an act that forbids the second at x's role is not drawn in x's row. A field that takes a person offers the persons the viewer knows, since the proposer's machine must know every participant, other than the viewer and the persons of the act's other places, since distinct roles are bound to distinct persons, but for two the clause says are possibly one person, which one person may fill (Section 5.4), and, where that person's role requires an atom that the schemas prove of a person from the viewer's records, as above, only a person so proved; the button is drawn only where each such field has someone to offer. The button of an act that names no entry, and of an acquaintance with a peer not listed, is drawn on the tab of its contract, where the viewer's state holds the atoms the act requires at the proposer's role that name no other role or variable, and lacks those it forbids there that name none. A button that covers several schemas is drawn once.

An approval question is drawn as its question with the buttons yes and no. A speech act, in a row, a question or a field, is drawn with its signers, innermost first, by the names the viewer's state holds for them. A notice is drawn as the refuser, the reason and the act's button — "Bob declined Befriend", "Bob unable Befriend" — with, where the reason is unable, the atoms the schema requires at the refuser that its state lacks and those it forbids that its state has.

6.3 The Implementation: Compiling vGLP onto GLP

vGLP needs no new machine: it compiles onto GLP, clause by clause, into a GLP program over an uninterpreted person channel, and Theorem 6.10 relates the runs of the two. GLP already connects the program to the person: each agent's first channel is to the person operating it; messages flowing toward the person are rendered by the runtime, and the person's responses flow back as GLP messages [53, 57]. In the compiled program each goal of a volitional procedure, at its first reduction attempt, has the dispatcher spawn a *construct process* for its question and receives its end of its interactive variable. The *bridge*, the Dart side of the runtime, draws each construct's widget and returns the person's acts on it, each act a *grant*. We first make the person's side formal.

Definition 6.8 (Person Channel, Person Writer, GLP with Persons, Grant). Let N be a GLP program and $P \subset \Pi$. A *person channel* of a goal G_0 satisfying SO is a variable pair $(U, U?)$ such that $U?$ occurs in G_0 and U does not; U is its *person writer*. *GLP with persons* over P and N, with a designated person channel of each agent's initial resolvent, is the transition system obtained from the maGLP

transition system over P and N (Definition 4.11) by adding, at each agent p, *grant* transitions

$$(G_p, \sigma_p) \;\rightarrow\; (G_p,\ \sigma_p \circ \{U? := [c \mid U_1?]\})$$

where c is a ground term `input(`Id`,` Pos`,` R`)`, U the person writer of p, and U_1 a writer of $\mathcal{V}_p$ occurring nowhere in the configuration, the person writer after the transition, every other agent stationary; the partial equivalence is the maGLP transaction equivalence, with grant transitions outside its domain.

A grant is the person's act: a tap, a keystroke or a submission on a construct grants the term it carries, the construct *Id*, the position *Pos* and the person's raw input *R*. The person channel is typed by these declarations, at the root of every program:

```
PersonChannel(T) ::= Channel(PersonIn(T), PersonOut(T)).
PersonIn(T) ::= [Input(T) | PersonIn(T)].
PersonOut(T) ::= [Draw(T) | PersonOut(T)].
Input(T) ::= input(Integer, Path, T).
Path ::= [] ; [Integer | Path].
Draw(T) ::= draw(Integer, Widget, T) ; withdraw(Integer).
```

Neither stream ends: the person and the program stay connected, and a goal standing for the person, as in a test, waits on the draws when it has nothing to grant. The parameter `T` is the type of the values drawn and of the person's raw inputs, which the compilation forms (Appendix B, **The construct process**); `Widget` is the type of the terms that draw widgets: the terms of the default widgets, built from the constructors of Definition 6.6, each carrying the label its widget declaration gives it, and, for a declared widget, the term `named(`W`,` D`)` of the name W its declaration gives and the default widget D of its type. Definition 5.10 (proper run) applies to GLP with persons verbatim, with no held variables and grants leaving the goals unchanged, and properness is the fairness condition of GLP with persons: nothing is required of the person, since the machine owes nothing to a question its person leaves open.

The compilation is defined for every vGLP program M. A goal of a volitional procedure q with interactive type T, called with its n arguments, first reduces with an *asking clause*, which sends an *ask* to the *dispatcher* and calls the guarded clauses of q, under a fresh name, with its end of a new interactive variable of type T added; this reduction is the Ask. Asks travel on *ask streams*: every procedure that may reach a question carries one, and a clause merges the ask streams of its body goals into its own. On an ask the dispatcher spawns the *construct process* of the question, which holds the person's end of the variable: the dispatcher routes the person's grants to it by its identifier; once the output the program writes into the variable is written whole, it has the bridge draw the widget of T with that output; and it binds each question it holds with the term the person's input forms, one element per submission for a stream type. The construct process withdraws when every question it holds is answered. The dispatcher takes asks, grants and draws in turn, so a grant is routed and a draw is passed on however many asks arrive. A reader the program drops leaves its construct on screen, showing what the program writes into it. The person's input on it answers the question, and the answer reaches no goal; the construct of a stream stays on screen, its further input reaching no goal.

The canonical compilation. Let M be a vGLP program. A procedure of M *reaches a question* if it is volitional or a clause of it calls a procedure that does; each such procedure has one more argument, last, its *ask stream*, which its goal writes and the dispatcher reads. In a clause of such a procedure, each call in the body of a procedure that reaches a question is given a fresh writer as its ask stream, and the head carries $D?$, where D is the clause's ask stream, with the body's ask

streams merged into D: the call is given D itself where there is one such call, `merge` goals merge them where there are more, and the head carries `[]` in place of D? where there is none.

For a volitional procedure q of M with interactive type T, the *asking clause* of q, under the name q so that every call of q reaches it unchanged, is

```
q(S1, ..., Sn, [ask(T, t(X))|D?]) :- q1(S1', ..., Sn', X?, D).
```

where S_l' passes the argument on: at an input position the head carries S_l and S_l' is its reader $S_l?$, and at an output position the head carries $S_l?$ and S_l' is the writer S_l. `X` and `X?` are exchanged where T is in writer mode. `t` is the name of T in lowercase, the names of its type arguments, so formed, appended, with `_w` for writer mode and `_r` for reader mode, as in `approval_w` and `stream_request_r`. Where two modules of M define types of one name, the name of each is prefixed with the path of its module, the names of its directories and its file joined by `_`, as in `parent_approval_w`. So the asks of M are of one type, the union over those functors.

The *canonical compilation* $\lceil M \rceil$ of a vGLP program M is the GLP program consisting of:

- every ordinary clause of M, extended with its ask streams (above);
- for each volitional procedure q of M, its clauses as guarded clauses of arity n+1 under a fresh name, `q1` above, extended, and its asking clause;
- for each interactive type T of M, the *construct process* of T, which the dispatcher spawns on an ask with its variable and its channels to the dispatcher (Appendix B);
- the *dispatcher*, which reads the ask stream of the initial goal and the person channel, spawns a construct process with a fresh identifier on each ask, and routes the person's grants to the construct processes by identifier, by the standard stream techniques (Appendix B prints it).

The initial goal of $\lceil M \rceil$ at each agent is that of M, extended, and the dispatcher on the ask stream of that goal and on the person channel, which is its person side.

A vGLP program is organised in modules as a GLP program is [58]. A `.vglp` source in the directory tree of a program is compiled into a module of that program, as a `.glp` source is, and the `self.vglp` of a directory holds the procedures, volitional or not, and the widget declarations shared by every module in that directory and below it, as its `self.glp` holds those of GLP; the `self.vglp` of the root holds widget declarations only. A procedure that reaches a question is exported, imported and shared as any procedure is, with its ask stream added, so the questions of a program are those of all its modules. The compilation forms the type of the asks, the construct processes and the dispatcher for the linked program, and the GLP module that forms the program's initial goal imports the compiled procedures it calls, ask streams included, and the dispatcher. The compilation is typed: $\lceil M \rceil$ carries the type definitions and procedure declarations of M, each procedure that reaches a question declared with its ask stream added, a volitional procedure's declaration being its asking clause's and its guarded clauses declared under their fresh name with the argument of its interactive type added in the mode of that type, the type of the asks, and the declarations of the construct processes and the dispatcher. Every module of $\lceil M \rceil$ type-checks as GLP [58] with no type written by hand: the compilation writes into each module that imports a compiled procedure or the dispatcher the types it forms for them, among them the type of the asks, the view types and the parameter of the person channel (Appendix B). A GLP module that calls a compiled procedure is a module of the vGLP program, every GLP program being a vGLP program, and is compiled with it.

A vGLP program declares the type of its output to its person by defining the type `Output`, a reserved name; the compiled program draws that output, construct `0`, by the widget of its type `Stream(Output)`, by default a thread whose elements are drawn by the widget of `Output` (Definition 6.6), and the dispatcher merges these draws with the constructs'. A program that writes

nothing to its person defines no `Output` and draws no construct `0`. A program that writes to its person and defines no `Output` is refused, and so is one whose `Output` has a position the person writes. Construct `0` is drawn when the program starts, and the dispatcher numbers the questions from 1.

The asking clause passes each argument of the goal on through a pair of its own, so until the pair's assignment is communicated a compiled clause that reads the argument suspends; a later clause that does not read it could then be taken where the source goal takes the earlier one, and the restriction on the clauses of a volitional procedure (Definition 5.2) excludes it. At an output position the asking clause's head carries a reader, which a writer of the goal matches and a term of the goal that is not a writer does not, so the calls of a volitional procedure hold writers there (Definition 5.2), and a goal keeps them until it is asked, holding their one occurrence.

The construct process is the program's end of the widget: it forms from the person's input a term of its question's type, so the type checker [58] guarantees that a question is bound only with a well-typed term. The bridge draws the widget and reports raw input; everything crossing to Dart is a ground term or a construct identifier, and answers and channels stay on the GLP side.

Definition 6.9 (Implementation of vGLP by GLP, Correct, Complete). The *implementation of vGLP by GLP* is the function taking a vGLP program M to its canonical compilation $\lceil M \rceil$ (Section 6.3). It is *correct* if for every M and $P \subset \Pi$ some σ is a correct implementation of the vmaGLP transition system over P and M, under the fairness condition that its runs are proper, by GLP with persons over P and $\lceil M \rceil$, and *complete* for M if for every P some such σ is also complete.

Theorem 6.10 (Compilation). *The implementation of vGLP by GLP is correct, and complete for every program whose interactive types have complete widgets.*

The proof, with the mapping σ, is in Appendix B. Correctness is per transition and then per class: the asking clause's reduction is the Ask, a construct process binding its question with an answer is the Answer, its draw of a question's output is the Communicate of that output to the person, and every other step of a construct process, of the dispatcher, of a `merge` goal the compilation adds and of a grant is a stutter; a class enabled forever is taken, the asking clauses and the compiled clauses being GLP clauses. Completeness is by construction: for each Answer of the given run, the person grants the input that forms its term, which a complete widget allows. The compilation is per agent, the person channel being local to its agent, and Communicate and Cold-call are carried unchanged by the multiagent GLP runtime [53, 57].

The implementation. The standard GLP implementation, extended with the compiler of the canonical compilation and the Dart bridge rendering the constructs, implements vGLP: a vGLP program is compiled into its canonical compilation, the engine runs it unchanged, and the bridge draws each construct's widget and returns the person's raw input as grants. The composite — engine, compiler, construct processes, dispatcher and bridge — is the implementation of vGLP; Section 7 shows each platform's requests and approval questions.

We claim, informally, that the composite is GLP with persons over the compiled program: a tap, a keystroke or a submission on a construct is a grant, and the engine is the GLP runtime of [57]. One step is serial: the person's raw input reaches the GLP side on one stream, and the dispatcher routes it to the construct processes by identifier.

6.4 The Cascade

The implementations of Sections 3.2, 5.4 and 6.3 compose. For an effective local specification (Definitions 3.8 and 5.16), let Θ be its canonical platform with a computable introduction rule or none, $\lceil \Theta \rceil$ the compilation of Θ into vGLP, and $\lceil \lceil \Theta \rceil \rceil$ the canonical compilation of that program onto

GLP; let σ_3 be the compilation mapping (Definition B.2), σ_2 the platform mapping (Definition 5.17), and σ_1 the CVA implementation (Definition B.1).

Theorem 6.11 (Cascade). *For every effective local volition-guarded specification, computable introduction rule or none, and $P \subset \Pi$, $\sigma_1 \circ \sigma_2 \circ \sigma_3$ is a correct implementation of $\mathcal{F}(P)$ by GLP with persons over P and $\lceil\lceil\Theta\rceil\rceil$, under the fairness condition that a run ρ is proper, that no approval question stays open in $\sigma_3(\rho)$ and its calendar does not stop where the specification requires the day, and that $\sigma_2(\sigma_3(\rho))$ is connected, unobstructed and steady.*

The proof is in Appendix B. The fairness condition asks of the persons that they answer the approval questions put to them, and of the run that one who can propose a transaction willed and not taken knows its other parties, that no agent is kept busy without end, and that a class willed and not taken has, from some point on, one part that holds with all its guards for the class. It does not ask for the strong fairness of Definition 3.22: the compiled agent takes a person's request at the dispatch that reads it, and a proposal among the own transactions it takes, one on each own trigger, until none holds, so a **Request** or a **Propose** enabled again and again is taken again and again.

The protocol of GLP with persons over $\lceil\lceil\Theta\rceil\rceil$ is defined as that of $\lceil\Theta\rceil$ in Section 5.4, with GLP with persons in place of vmaGLP.

Proposition 6.12 (Grassroots Cascade). *For every effective local volition-guarded specification with acquaintance parts guarded by every two agents, and computable introduction rule or none, the protocol of GLP with persons over $\lceil\lceil\Theta\rceil\rceil$ is grassroots.*

The proof is in Appendix B.

The results of the paper, and the conditions each places on the runs of the implementing system, are these:

Result	Arrow	Condition on the runs
Proposition 3.20		safe; for termination, also live
Theorem 3.24	(3) → (4)	connected, unobstructed, steady and strongly fair
Theorem 5.19	(4) → (5)	proper, no approval question staying open, the calendar not stopping, and its image connected, unobstructed and steady
Theorem 6.10	(5) → (6)	proper
Theorem 6.11	(3) → (6)	proper, no approval question staying open, the calendar not stopping, and its image in $\mathcal{P}$ connected, unobstructed and steady

7 Grassroots Platforms

This section presents three grassroots platforms — the grassroots social graph, grassroots social network, and grassroots currencies — as the contracts the compiler of Section 5.4 compiles: GSG and GSN [16], and GC [49]. Each contract is in a box of its own, its text the repository's copy, opening with the declaration of its names and cited by the contract's name and a clause's number, as GSG:2; a contract that extends another cites its clauses and does not restate them. Beside the clauses of each contract the repository holds its speech-act schemas [16], one for each act a clause describes, by which the contract is a local specification (Definition 3.8), and its *pragmas*, the words its app shows, printed in its box below its clauses, from which the compiler takes the words of the platform it prints (Section 5.4). A clause opens with its legal modality [16]: ■ enforced, ◆ attested, ○ undertaken. The CVA implementation of each is its canonical CVA platform (Definition 3.17), correct by Theorem 3.24, and, the contract being effective (Definition 5.16), the compiler checks it and prints the compilation of that platform into vGLP (Section 5.4), correct by Theorem 5.19. A binary transaction guarded by both parties is then the two-phase commit over two participants

(Section 3.2): an offer is a request of one party to their own machine, which proposes it; an acceptance is the other party's **Answer** to the approval question that **Start** raises, and the proposing machine commits on receipt of the acceptance's vote and tells the other party; the person's no to it takes **Answer**'s else-branch.

The programs are in the public GLP repository, https://github.com/EShapiro2/GLP, under `programs/`: GSG and GSN, with their schemas, in `gsc`, and GC in `currencies/coins`. The apps of GKK, GSG, GSN and GC are in `apps` (Section 8). A run of the cascade on each contract is recorded in `cascade/`: its clauses and schemas, the checker's verdict, the compiled program in vGLP and in GLP, and the log of its run with scripted persons, which the test suite repeats. Programs written earlier, by hand, are in `social/graph` (the social graph, one-to-one messaging and the super-app [13]), in `currencies` (the coins agent, in the syntax of Section 5) and in `cssn` (the child-safe network, in an earlier syntax of vGLP). A `.vglp` source in the syntax of Section 5 is compiled by the canonical compilation of Section 6.3 into the `.glp` file beside it. The requests below are given in the syntax of Section 5.

7.1 Grassroots Social Graph

In a grassroots social network [51], the social graph is stored distributively under the control of the people themselves, with each person storing the local neighbourhood pertaining to them, and no third-party having access unless explicitly granted. A *person* is a legally culpable entity: a natural person of age, also referred to as an *adult*, or a legal person of good standing, such as a business. A person takes the pronoun 'they'. Every contract extends GKK, Grassroots Keys and Knowledge [16], and cites its clauses whole; a person knows another by holding their public key.

GKK: Grassroots Keys and Knowledge [16]

Alice, Bob and Carol are persons.

1. ○ *Identity.* Alice may act once to establish their identity by choosing their unique key pair. They make themselves known by the public key, sign their acts with the private key, and are accountable for every act signed with it.
2. ■ *Acquaintance.* Alice and Bob may come to know each other by an act of both.
3. ■ *Introduction.* If Alice and Bob know each other and Alice knows Carol, then an act of Alice and Bob can introduce Carol to Bob, following which Bob knows Carol.

Pragmas

Contacts: the persons Alice knows.
Identity: `Alice [Establish]`
Acquaintance: `Alice [Acquaint] <person>`
`  Bob: Alice wishes that you know each other.`
Introduction: `Alice [Introduce] <person>`
`  Bob: Alice asks to introduce Carol to you.`

GSG extends GKK with friendship.

GSG: Grassroots Social Graph [16]

Alice and Bob are persons.

1. ◆ *Friendship.* Alice and Bob, who know each other personally and are not friends, may jointly decide to become friends.
2. ■ *Ending a friendship.* Alice may end their friendship with Bob.

Pragmas

Friends: the friends of Alice.
Friendship: `Alice [Befriend]`
`  Bob: Alice asks to be your friend.`
Ending a friendship: `Alice [Unfriend]`

GSG is the grassroots social graph, introduced and analysed in [28, 51].

The schemas. The schemas of GKK are over two predicates: *identity*, of arity zero, an identity established under GKK:1, and *knows*, of arity one, *knows*(q) in the state of p saying that p holds q's public key. The schema *establish*(p?) adds *identity* at p; *acquaint*(p?, q?) requires *identity* at both roles and adds *knows*(q) at p and *knows*(p) at q; and *introduce*(p?, q?; c), c bound to an agent, requires *identity* at both roles, *knows*(q) and *knows*(c) at p and *knows*(p) at q, and adds *knows*(c) at q. GSG adds *friend*, of arity one, *friend*(q) in the state of p saying that q is a friend of p. The schema *befriend*(p?, q?) requires *identity* at both roles, *knows*(q) at p and *knows*(p) at q, forbids *friend*(q) at p and *friend*(p) at q, and adds them; both roles are marked, so the act takes the will of both. The schema *unfriend*(p?, q) requires *identity* at both roles and deletes the two atoms; its class has two parts, at the binding of either friend in the first role, each guarded by that friend (Section 5.4). The accountability GKK:1 states and the acquaintance GSG:1 requires describe no act.

vGLP implementation. The compilation of the canonical platform (Section 5.4). The agent's request type has the alternative `propose(`N`)` for each request name N, a schema-binding of *establish*, of *acquaint*, *introduce*, *befriend* or *unfriend* with a peer, and an acquaintance with a peer whose key the person has is also the cold call by which their machines come to know each other (Section 5.4). The approval question **Start** raises on an arriving offer of friendship shows the offering peer and takes yes or no, no being the refusal; the approval questions of acquaintance and of introduction show the peer, and the peer introduced, likewise. The request form is a menu of the requests.

7.2 Grassroots Social Network

GSN extends GSG with messages and groups [16, 51]. Its clauses speak of agents: an *agent* is a person, a child or a computational (AI) agent [16].

GSN: Grassroots Social Network [16]

Alice and Bob are persons.

1. ■ *Message.* Alice may send a message to their friend Bob.
2. ■ *Group.* Alice may create a group, of which they are the leader and first member.
3. ■ *Joining.* Alice, the leader of a group, may add their friend Bob to it, by an act of both.
4. ■ *Leaving.* Bob, a member of a group, may decide to leave it; when the leader leaves, the group ends. A member of a group is removed from it if they cease to be a friend of its leader.
5. ■ *Group message.* Bob, a member of a group, may send a message to the group, and the leader delivers it to every member.

Pragmas

Groups: the groups Alice is a member of.
Message: `Alice [Message] <text>`
Group: `Alice [Create] <name>`

Joining: `Alice [Join]`
  `Bob: Alice asks you to join their group.`
Leaving: `Bob [Leave]`
  `Alice [Close]`
Group message: `Bob [Post] <text>`
  `Alice [Post own] <text>`

The schemas. A group is named by the community name $\langle a \rangle \cdot n$ of its leader a and a speech act n [16]: $member(\langle a \rangle \cdot n)$ in the state of q says that q is a member of the group, and $listed(q, n)$ in the state of a says the same. GSN:1 is the schema $message(p?, q; x)$, which adds at q, a friend of p, the content x received from p. GSN:2 is $create(a?; n)$. GSN:3 is $join(a?, q?; n)$, guarded by both, q a friend of a. GSN:4 is $leave(a, q?; n)$, guarded by the member; $close(a?, a[listed(*, n)]; n)$, by which the leader ends the group and every membership in it in one act, its listed role being every member the leader lists; and $drop(a, q; n)$, unguarded, which removes a member who is not a friend of the leader. GSN:5 is $post_own(a?; x, n)$, the leader's own message; $post(p?, a; x, n)$, a member's message, sent to the leader; and $deliver(a, q; x, n)$, unguarded, by which the leader delivers a message to each member. A post and a delivery require that the member be the leader's friend, so a group's messages travel between friends.

vGLP implementation. The compilation of the canonical platform (Section 5.4). The schema-bindings of *message*, *create*, *join*, *leave*, *close* and the two posts are request names, alternatives `propose(`N`)` of the agent's request type. A join proposed by either party raises an approval question at the other, which shows the group and its leader and takes yes or no. The leader's agent proposes a close, reading the members from its own state. Drop and deliver, unguarded, are proposed by the leader's machine, their opener, and voted on by the member's. Posting is a request, one message per submission, as the chat of Section 5 is.

7.3 Grassroots Currencies

Grassroots coins [27, 49] are units of debt that can be issued and traded digitally by any person. Each person's coins are backed by the goods and services they offer, priced in their own currency, with liquidity arising from mutual credit via coin exchange among persons that know and trust each other. A *party* is a person, natural or legal (with the gender-neutral pronoun 'they'), with *Alice*, *Bob* and *Carol* as variables naming distinct parties to the contract. A *grassroots coin* is a unit of debt that names its issuer, with *Alice-coin* referring to a grassroots coin issued by Alice.

GC: Grassroots Currencies [49]

Alice and Bob are persons.

1. ■ *Minting.* Alice may decide to issue Alice-coins.
2. ■ *Swap.* Alice and Bob, who know each other, may jointly decide to swap coins they hold.
3. ■ *Payment.* Alice may decide to pay Bob-coins they hold to Bob, whom they know.
4. ■ *Redemption.* Alice may decide to redeem a Bob-coin they hold against any coin held by Bob, whom they know.
5. ○ *Pricing.* Alice prices their offerings in Alice-coins and accepts them in payment at those prices.

Pragmas

Coins: the coins Alice holds.

Minting: `Alice [Mint] <amount>`
Swap: `Alice [Swap] <issuer given> <amount given> <issuer wanted> <amount wanted>`
  `Bob: Alice asks to swap coins with you.`
Payment: `Alice [Pay] <amount>`
Redemption: `Alice [Redeem] <issuer wanted>`

The schemas. GC extends GKK (Section 7.1). The schemas in the repository [16, 49] are over one predicate beside GKK's, *coin*, of arity one, $coin(u)$ in the state of p being a coin issued by u and held by p, as many copies as p holds such coins, and amounts are the numbers speech-act schemas allow (Section 5.4). The schema $mint(p?; k)$ adds k copies of $coin(p)$ at p; $swap(p?, q?; u, k, v, k')$, guarded by both, who know each other, moves k copies of $coin(u)$ from p to q and k' copies of $coin(v)$ from q to p; $pay(p?, q; k)$, p knowing q, moves k copies of $coin(q)$ from p to q; and $redeem(p?, q; r)$, p knowing q, moves $coin(q)$ from p to q and $coin(r)$ from q to p. A redemption and the swap of one coin for one of the same two coins are one transaction, determined by two schema-bindings, so their class has a part guarded by both parties and a part guarded by the bearer. GC:5 describes no act.

vGLP implementation. The compilation of the canonical platform (Section 5.4): every schema is guarded, so each schema-binding is a request name, an alternative `propose(`N`)` of the agent's request type whose arguments the request form takes. A swap raises an approval question at the other party, which shows the coins given and the coins taken and takes yes or no. The other party's machine votes no on a swap whose wanted coins it does not hold, without asking its person. It votes on a payment and on a redemption by its own precondition, and its person is asked by neither. The coins agent of the repository, in `currencies/coins`, written earlier by hand, has the requests `mint(Amount)`, `swap(Friend, GiveCoin, GiveAmount, WantCoin, WantAmount)`, `pay(Friend, Amount)` and `redeem(Friend, WantCoin)`, and the holdings are a list the program writes to the person.

The agent's request form is a menu of its requests, each a form whose fields are the request's arguments, so the person's input is the term the agent consumes.

Effective specifications. The three contracts meet the conditions of Section 5.4, which the compiler checks: every variable of a schema occurs in an atom it deletes or adds; GSN's unguarded schemas, drop and deliver, name the member, the message and the group in atoms of the leader's state; and GSN's close has one listed role, whose base, the leader, is its only guarding role. So each is an effective local specification, and its canonical platform compiles into vGLP (Section 5.4).

8 The Grassroots App

The apps of GKK, GSG, GSN and GC are compiled from their contracts down the cascade (Section 7) and run as mini-apps of the grassroots super-app [13] on an iPhone simulator; Figure 2 shows them. Each app's directory in the repository, `programs/apps/gkk`, `gsg`, `gsn` and `gc`, holds the canonical CVA platform the compiler of Section 5.4 prints from the contract's schemas and pragmas, `platform.vglp`, which the canonical compilation (Section 6.3) compiles to GLP when the directory loads, with the three widget declarations the compiler prints (Section 6.2); the mini-app, which wraps the compiled agent and its dispatcher; and a scripted person. Every element of a screen is a construct (Definition 6.1) or the program's output: the request question, drawn as the tabs' buttons and fields, a button beside each row it names; the approval questions, each drawn at the top of its tab as its question with yes and no; and the view the agent writes, drawn as the tabs' lists, a row for

each atom of the tab's predicate, with the notices. Every word on a screen is the contract's or one of the platform's four, yes, no, declined and unable (Section 6.2); the app's title is the contract's name.

GKK. (1) Alice's *Contacts* tab, the tab of GKK, before she establishes her identity: Establish, and Acquaint with its field *person*, for the key of a peer not listed. (2) Alice typed Bob's key and tapped Acquaint: her request is the cold call, which asks Bob "Alice wishes that you know each other." with yes and no at the top of his *Contacts* tab (Section 5.4). (3) Bob's *Contacts* tab after he answered yes to "Alice asks to introduce Carol to you.": a row for each person he knows, Carol and Alice, each with Meet and Introduce.

GSG. (4) Alice, who met Bob, tapped Befriend beside him: Bob's *Friends* tab, the tab of GSG, with "Alice asks to be your friend." and yes and no. (5) Alice's *Friends* tab after the befriend commits: Bob, with Meet, Introduce, Befriend and Unfriend. (6) Alice's *Contacts* tab after Bob ends the friendship: Bob listed there, and her *Friends* tab lists no one.

GSN. (7) Bob's chat with Alice in his *Friends* tab, her message "Hi Bob" with her name under it, delivered under GSN:1, and the field *text* with Message. (8) Bob's *Groups* tab, the tab of GSN, with "Alice asks you to join their group.", the group's name, *friends*, and yes and no. (9) Bob's *Groups* tab with the group *friends*, led by Alice, and his message "Hi all" as Alice delivered it to him under GSN:5, with Post and Leave. (10) Alice's *Groups* tab with the group she leads, its members Bob and Alice, Bob's "Hi all" and her "Welcome", with Post own and Close.

GC. (11) Alice typed 2 into Mint's field *amount* and tapped Mint: her *Coins* tab, the tab of GC, with a row for her coins, Alice, and their count, 2. (12) Alice asked Bob to swap her two Alice-coins for his three Bob-coins: his *Coins* tab with "Alice asks to swap coins with you.", the binding's values under it — issuer given Alice, amount given 2, issuer wanted Bob, amount wanted 3 — and yes and no, and his row, Bob 3. (13) Bob answered yes and the swap committed: his *Coins* tab with the Alice-coins he holds, Alice 2, with Pay and Redeem; Swap is in the rows of his *Contacts* tab. (14) Alice tapped Pay on Bob's row and typed 2: one Bob-coin left.

How the screens are taken. A scenario in the repository, `glp_multiagent/test/apps_screens_test.dart`, runs every agent's mini-app in one GLP engine, the agents' conversations channels within it, and takes each person's acts by tapping and typing on the tabs' controls; the test suite runs it headless and checks every screen. On the simulator the same scenario runs in one app on an iPhone 16e, one person's phone on the screen at a time, and each screen is that phone's screenshot.

The implementation as it stands. The compilation of Section 6 is implemented: the compiler emits each program's asking clauses, construct processes and dispatcher, the type checker checks the result, and the test suite runs the programs of Sections 1 and 5 with scripted persons that read the draws and grant on the constructs, each run checked against the committed sources. A compiled module `M` is run by the goals

```
M # agent(MCh?, NetCh?, Asks), M # dispatch(Asks?, PersonCh?, MCh)
```

with the person channel `PersonCh = ch(Grants?, Draws)`, its grants `input(Id, P, R)` — the construct, the position and the person's raw input — and its draws `draw(Id, W, V)` and `withdraw(Id)` (Section 6.3). The Dart bridge, one construct family (Remark 6.4), draws these constructs and returns the person's input, and the apps above run through it. The super-app of [13], as deployed on a physical smartphone, runs earlier programs of the three platforms, in an earlier syntax of vGLP, compiled by an earlier compilation into an agent and a mediator; the four apps above are compiled from their contracts.

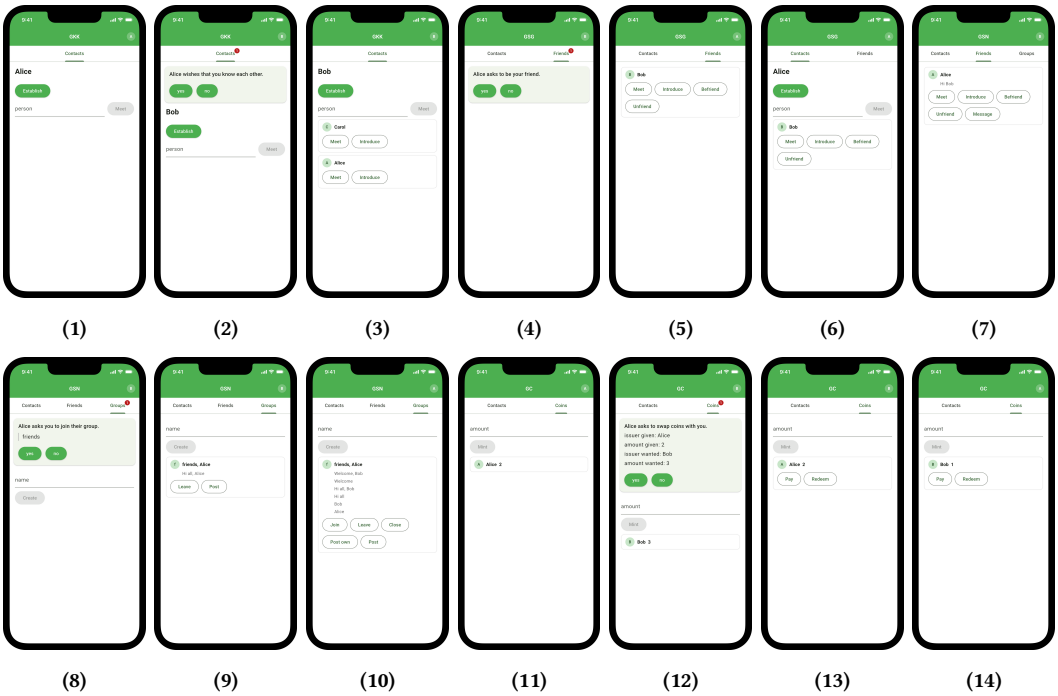


Fig. 2. **The apps of GKK, GSG, GSN and GC.** Each is a screenshot of the GKK, GSG, GSN or GC mini-app, compiled from that social contract down the abstraction cascade [16, 28, 53, 57, 58]. One instance of the grassroots app [13] on an iPhone simulator runs the mini-app of every agent [13, 57]; the agents communicate over channels within it [53], and the screenshot shows one person's screen. All source code and all artefacts, including the social contracts and the artefacts the abstraction cascade produces from them, are in the public repository https://github.com/EShapiro2/GLP. GKK: (1) Alice before establishing her identity; (2) Bob asked to come to know Alice; (3) Bob's contacts after an introduction. GSG: (4) Bob asked to befriend Alice; (5) Alice's friends after the befriend; (6) after Bob ends the friendship. GSN: (7) Bob's chat with Alice; (8) Bob asked to join Alice's group; (9) the group, with Bob's message delivered; (10) Alice's group. GC: (11) Alice minted; (12) Bob asked to swap; (13) Bob's holdings after the swap; (14) Alice paid.

9 Related Work

vGLP makes two independent contributions: it conditions a machine step on a person's volition, and it derives the interface from the program's semantics. The first relates to formal approaches to user interface semantics, in which the person is a source of input, an environment, a chooser, an editor, or a signer; we review these first, by the person's role. The second relates to work that derives the interface automatically, but from an application's data, object model, or tasks rather than from a semantics whose steps a person guards; and a further tradition predicts or verifies a given interface. In none does a transaction wait on the wills of its guards, with an else-branch taken when a guard does not will it, and in none is a running interface compiled from such transactions with a proof that it implements them. Underlying these differences is what a human input *is*: in the languages that present a program to its user it is a *command* driving a tool; in those that generate an interface from data or tasks it is a *value* completing a task; in vGLP it is a *volition* authorising an otherwise-ready machine transaction — a guard.

Formal approaches to user interface semantics. Functional reactive programming gave interaction a denotational semantics with Fran [14]; Fudgets composed graphical interfaces from stream processors in a lazy functional language [8]; Elm gave a GUI language an operational semantics [10]; and Krishnaswami and Benton gave GUIs a denotational model with user input as nondeterminism, by a monad [25]. Task-oriented programming executes typed editors the person fills: iTasks [40, 41], and its formal core TopHat, whose labelled transitions consume user inputs [60]. iTasks derives those editors generically from the task's data types, and TopHat from the task's syntax too, the user continuing a step explicitly, so an interface *is* derived, from the data and the task, and the inputs drive a workflow. TopHat proves that the inputs it computes for a task are exactly those its semantics accepts [60], as Theorem 6.3 proves of a construct's answers. In vGLP only the person gives an answer (Theorem 5.14), the answers guard multiagent transactions, and the cascade is proved correct. Game semantics models a program against an Opponent whose strategy is the environment's [23]; interaction trees represent impure programs whose events the environment answers [63]; interactive computation makes the environment part of the computational model [61]. Formal human-computer interaction modelled interactive systems directly: the PIE model [11], interactors [12], Petri-net interactive cooperative objects [36], statecharts [20], and the synchronous languages [4]. These works define the semantics of an interface already designed. The concurrent logic languages connected keyboard and screen as stream-processing agents [47]; a message to the person could carry an unbound variable that the person's response assigns. Ceptre's interactive stages let the player select which enabled linear-logic rule fires [30]: the person resolves nondeterminism. Hazelnut gives an operational semantics to the person's edit actions on programs [34]. Smart-contract semantics fire a transition only on a transaction signed by a key [46]. Beliefs, desires, and intentions in agent programming [42] are the software agent's own, not a person's.

The person in concurrency theory. In CSP [21], CCS [31] and the π-calculus [32] the environment is a process, and a person is one more process or an external choice the environment resolves; session types [22] discipline the conversation between processes and say nothing of who wills it. The person's role across the formal traditions — choice machines, I/O automata and reactive systems, angelic and demonic nondeterminism, alternating-time logics, ceremony analysis — is reviewed in [28], and GLP's moded types are set against session types in [58]. Here the agent is a person and a machine, and a machine transaction is subject to the person's volition. A process calculus would host the transition systems of Section 2 as well as GLP does; it does not by itself derive an interface from a procedure's interactive type.

The interface derived from the program. Deriving the interface from the program, rather than authoring it separately, is an established idea, on bases distinct from ours. Naked objects presents a system's domain objects directly and creates the interface automatically from their definitions by reflection [38, 39], in a noun-verb style: the person views objects and invokes their methods, with no hand-written presentation layer. Model-based user-interface development derives interfaces from separate task and domain models — the CAMELEON reference framework [5], ConcurTaskTrees and model-based design [37], and description languages such as UIML [1] and UsiXML [29], surveyed in [2]; task-oriented programming (above) derives its editors generically from data types and tasks; and structure editors derive an editing interface from a grammar [44]. In each the interface follows from a description of the application's data, objects, or tasks.

Typed widgets. Three works associate an interface element with a type, as vGLP associates a widget with an interactive type (Section 6.2). Formlets [9] abstract a part of a web form that collects a value of a given type: a formlet renders its HTML and parses the submission into a typed value, formlets compose as idioms, and the handler attached to a form consumes the value. Tangible values [15] pair a value with a visualizer whose structure follows the structure of the value's type,

with a default input and output per type and alternatives chosen explicitly, and end users compose them gesturally into new functions. Livelits [33] fill a typed hole in a program with a user-defined GUI whose expansion is an expression of the hole's type, kept live by evaluating the expressions spliced into it in the closures collected around the hole. In each, the interface produces a value of a type for a program or a programmer; in vGLP a widget forms a person's answer to a question the running program asks, and the reduction that reads the answer is the transaction the person wills.

Atomic commitment. Two-phase commit [19, 26] commits a transaction across sites on the votes of all, a coordinator collecting the votes and announcing the outcome; Skeen's decentralised form has every site send its vote to every site and decide by itself, and his nonblocking protocol adds a phase so that a site may decide under failures [59]; the textbooks [3, 18] treat the family. The two-phase commit of Section 3.2 is the coordinated form, the agent that initiates the transaction coordinating it, with a guard set in place of unanimous consent: the persons of the guard set whose approval does not stand are asked, every other participant votes on its own precondition, and the else-branch is taken when a person does not will the then-branch. Concurrent attempts are ordered by wound-wait [45], a deadlock-prevention scheme for distributed databases: a busy participant lets a start wait rather than refuse it, an attempt of higher priority makes the one of lower priority that blocks it yield, and an attempt tried again after yielding keeps its priority, so that contention neither deadlocks nor livelocks. Fault tolerance is left to work in preparation.

Predicting and verifying a given interface. Interface design is often treated as a craft outside formal computer science, yet two lines formalise it. Predictive engineering models estimate a design's performance from the person's elementary actions — the keystroke-level model and the GOMS family [6, 7, 24]. Formal methods for interactive systems specify and verify interfaces — from grammars for user interfaces [43] through the collected work of the field [35, 62], of which the dialogue models above are part.

To the best of our knowledge, vGLP is the first programming language whose operational semantics conditions transactions on a person's volition — elicited, awaited until given, and fulfilled — with the interface derived from the questions that semantics asks, each rendered by the widget of its interactive type, rather than from the application's data or tasks.

10 Conclusion

We incorporated people in the operational semantics of a programming language: a goal of a vGLP volitional procedure puts a typed question to its person, a clause that reads the answer reduces only after the person has answered, and the person's answers are their own. The structure of the interface is derived from the semantics: a program's questions are the interactive variables of its goals, each rendered by the widget of its interactive type — a pair of yes and no buttons, a form, the request form of a stream question, an input box for a stream. The abstraction cascade proceeds from volition-guarded multiagent atomic transactions, by two-phase commit coordinated by the initiating agent, to communicating volitional agents, then by compilation to vGLP, and by compilation onto GLP; it is proved correct down to GLP. The three platforms are compiled down the cascade from their contracts and run on GLP with scripted persons, and the apps of the three, so compiled, run on an iPhone simulator. A contract carries the words its app shows besides its clauses and its schemas, so the app is compiled from the contract. The two-phase commit orders contending attempts by wound-wait: every attempt ends, and contention keeps no transaction that stays enabled from being carried out. An interactive type is the contract a user-interface developer programs against: any construct family whose widgets preserve it inherits the theorems, so the rendering of a platform is the person's choice.

Future work: a widget language beyond the basic types, drawing on user-interface specification languages; a GLP channel per construct in the bridge, removing the dispatcher's serial step; and standing volitions — a person willing a class of goals rather than answering one question.

References

[1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. 1999. UIML: An Appliance-Independent XML User Interface Language. In *Proc. 8th Int. World Wide Web Conf. (WWW)*.

[2] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. 2015. Adaptive Model-Driven User Interface Development Systems. *Comput. Surveys* 47, 1 (2015), 9:1–9:33.

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[4] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. doi:10.1016/0167-6423(92)90005-V

[5] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. 2003. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers* 15, 3 (2003), 289–308.

[6] Stuart K. Card, Thomas P. Moran, and Allen Newell. 1980. The Keystroke-Level Model for User Performance Time with Interactive Systems. *Commun. ACM* 23, 7 (1980), 396–410.

[7] Stuart K. Card, Thomas P. Moran, and Allen Newell. 1983. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates.

[8] Mats Carlsson and Thomas Hallgren. 1993. FUDGETS: A Graphical User Interface in a Lazy Functional Language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA).* ACM, 321–330. doi:10.1145/165180.165228

[9] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2008. The Essence of Form Abstraction. In *Programming Languages and Systems, 6th Asian Symposium (APLAS).* 205–220. doi:10.1007/978-3-540-89330-1_15

[10] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM, 411–422. doi:10.1145/2491956.2462161

[11] Alan J. Dix. 1991. *Formal Methods for Interactive Systems.* Academic Press.

[12] David J. Duke and Michael D. Harrison. 1993. Abstract Interaction Objects. *Computer Graphics Forum* 12, 3 (1993), 25–36. doi:10.1111/1467-8659.1230025

[13] Ohad Eitan, Idit Keidar, and Ehud Shapiro. 2026. The Grassroots Super-App: A Secure Operating System for Grassroots Mini-Apps. arXiv:XXX:XXXXX. arXiv:XXX:XXXXX [cs.DC]

[14] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM, 263–273. doi:10.1145/258948.258973

[15] Conal M. Elliott. 2007. Tangible Functional Programming. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM, 59–70. doi:10.1145/1291151.1291163

[16] James Golike, Andy Lewis-Pye, and Ehud Shapiro. 2026. Formalising and Implementing Grassroots Social Contracts: From Legal Text to Grassroots Platforms (Full Version). *arXiv:2609.06231* (2026).

[17] Google. 2024. Dart Programming Language. https://dart.dev/

[18] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

[19] James N. Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course.* Lecture Notes in Computer Science, Vol. 60. Springer, 393–481.

[20] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (1987), 231–274. doi:10.1016/0167-6423(87)90035-9

[21] C. A. R. Hoare. 1985. *Communicating Sequential Processes.* Prentice-Hall.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. doi:10.1145/2827695

[23] J. M. E. Hyland and C.-H. Luke Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Information and Computation* 163, 2 (2000), 285–408. doi:10.1006/inco.2000.2917

[24] Bonnie E. John and David E. Kieras. 1996. The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast. *ACM Transactions on Computer-Human Interaction* 3, 4 (1996), 320–351.

[25] Neelakantan R. Krishnaswami and Nick Benton. 2011. A Semantic Model for Graphical User Interfaces. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM, 45–57. doi:10.1145/2034773.2034782

[26] Butler W Lampson. 1981. Chapter 11. atomic transactions. In *Distributed Systems—Architecture and Implementation: an Advanced Course.* Springer, 246–265.

[27] Andrew Lewis-Pye, Oded Naor, and Ehud Shapiro. 2023. Grassroots Flash: A Payment System for Grassroots Cryptocurrencies. *arXiv preprint arXiv:2309.13191* (2023). https://arxiv.org/abs/2309.13191

[28] Andy Lewis-Pye and Ehud Shapiro. 2026. Volition-Guarded Multiagent Atomic Transactions: Describing People and their Machines. *arXiv preprint arXiv:2604.25596* (2026). https://arxiv.org/abs/2604.25596

[29] Quentin Limbourg, Jean Vanderdonckt, Benoît Michotte, Laurent Bouillon, and Víctor López-Jaquero. 2004. USIXML: A Language Supporting Multi-Path Development of User Interfaces. In *Engineering Human Computer Interaction and Interactive Systems (EHCI-DSVIS) (LNCS, Vol. 3425)*. Springer, 200–220.

[30] Chris Martens. 2015. Ceptre: A Language for Modeling Generative Interactive Systems. In *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. AAAI Press, 51–57.

[31] Robin Milner. 1980. *A Calculus of Communicating Systems*. LNCS, Vol. 92. Springer.

[32] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I and II. *Information and Computation* 100, 1 (1992), 1–77.

[33] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 511–525. doi:10.1145/3453483.3454059

[34] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 86–99. doi:10.1145/3009837.3009900

[35] Philippe Palanque and Fabio Paternò (Eds.). 1998. *Formal Methods in Human-Computer Interaction*. Springer.

[36] Philippe A. Palanque and Rémi Bastide. 1994. Petri Net Based Design of User-Driven Interfaces Using the Interactive Cooperative Objects Formalism. In *Design, Specification and Verification of Interactive Systems (DSV-IS)*. Springer, 383–400.

[37] Fabio Paternò. 2000. *Model-Based Design and Evaluation of Interactive Applications*. Springer.

[38] Richard Pawson. 2004. *Naked Objects*. Ph. D. Dissertation. Trinity College Dublin.

[39] Richard Pawson and Robert Matthews. 2002. *Naked Objects*. Wiley.

[40] Rinus Plasmeijer, Peter Achten, and Pieter W. M. Koopman. 2007. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 141–152. doi:10.1145/1291151.1291174

[41] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. In *Proc. 14th Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*. ACM, 195–206.

[42] Anand S. Rao and Michael P. Georgeff. 1995. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multiagent Systems (ICMAS)*. The MIT Press, 312–319.

[43] Phyllis Reisner. 1981. Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Transactions on Software Engineering* 7, 2 (1981), 229–240.

[44] Thomas Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*. 42–48.

[45] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. 1978. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems* 3, 2 (1978), 178–198.

[46] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30. doi:10.1145/3360611

[47] Ehud Shapiro. 1989. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR)* 21, 3 (1989), 413–510. doi:10.1145/72551.72555

[48] Ehud Shapiro. 2021. Multiagent Transition Systems for Composing Fault-Resilient Protocol Stacks. *arXiv preprint arXiv:2112.13650* (2021). https://arxiv.org/abs/2112.13650

[49] Ehud Shapiro. 2022. Grassroots Currencies: Turning Mutual Trust into Liquidity. *arXiv preprint arXiv:2202.05619* (2022). https://arxiv.org/abs/2202.05619

[50] Ehud Shapiro. 2023. Grassroots Distributed Systems: Concept, Examples, Implementation and Applications (Brief Announcement). In *37th International Symposium on Distributed Computing (DISC 2023). (Extended version: arXiv:2301.04391)*. LIPICS, Italy, 47:1–47:7. https://arxiv.org/abs/2301.04391

[51] Ehud Shapiro. 2023. Grassroots Social Networking: Serverless, Permissionless Protocols for Twitter/LinkedIn/WhatsApp. In *OASIS '23* (Rome, Italy). Association for Computing Machinery. doi:10.1145/3599696.3612898

[52] Ehud Shapiro. 2025. Characterising Global Platforms: Centralised, Decentralised, Federated, and Grassroots. *arXiv preprint arXiv:2511.03286* (2025). https://arxiv.org/abs/2511.03286

[53] Ehud Shapiro. 2025. GLP: A Grassroots, Multiagent, Concurrent, Logic Programming Language for AI (full version). *arXiv preprint arXiv:2510.15747, Summary in Proc. of ICLP'26, EPTCS 450, pp. 119–133* (2025). https://arxiv.org/abs/

2510.15747

[54] Ehud Shapiro. 2026. Child-Safe Social Networking: From a Grassroots Social Contract to an App. *In preparation* (2026).

[55] Ehud Shapiro. 2026. Grassroots Bonds: Financing by the People, for the People. arXiv:2603.13671. arXiv:2603.13671 https://arxiv.org/abs/2603.13671

[56] Ehud Shapiro. 2026. Grassroots Platforms with Atomic Transactions: Social Graphs, Cryptocurrencies, and Democratic Federations. In *Proceedings of the 27th International Conference on Distributed Computing and Networking*. 71–81. doi:10.1145/3772290.3772309 arXiv preprint arXiv:2502.11299.

[57] Ehud Shapiro. 2026. Implementing Grassroots Logic Programs with Multiagent Transition Systems and AI (Full Version). *arXiv:2602.06934. Summary to appear in Proc. of LOPSTR+PPDP'26* (2026). https://arxiv.org/abs/2602.06934

[58] Ehud Shapiro. 2026. Types for Grassroots Logic Programs (Full version). *arXiv preprint arXiv:2601.17957* (2026). https://arxiv.org/abs/2601.17957

[59] Dale Skeen. 1981. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*. ACM, 133–142.

[60] Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019. TopHat: A Formal Foundation for Task-Oriented Programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP)*. ACM, 17:1–17:13. doi:10.1145/3354166.3354182

[61] Peter Wegner. 1997. Why Interaction Is More Powerful Than Algorithms. *Commun. ACM* 40, 5 (1997), 80–91. doi:10.1145/253769.253801

[62] Benjamin Weyers, Judy Bowen, Alan Dix, and Philippe Palanque (Eds.). 2017. *The Handbook of Formal Methods in Human-Computer Interaction*. Springer.

[63] Li yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 51:1–51:32. doi:10.1145/3371119

A Grassroots Protocols

Section 3 states that the protocol of every CVA platform is volitionally grassroots; the definitions are these. We are particularly interested in *grassroots protocols* [28, 50, 56], which formalise the idea that grassroots platforms can have multiple instances that form and operate independently, yet may coalesce into ever-larger instances, possibly (but not necessarily) into a single global platform. We recall the definitions of [28]. A *prefix* of a computation is a finite initial segment of it.

Definition A.1 (Interleaving [28]). Let $P, P' \subset \Pi$ be disjoint and nonempty, $r = c_0, c_1, \ldots$ a run of $\mathcal{F}(P)$ and $r' = d_0, d_1, \ldots$ a run of $\mathcal{F}(P')$. An *interleaving* of r and r' is a sequence $e_0, e_1, \ldots$ of configurations of $\mathcal{F}(P \cup P')$ for which there are non-decreasing index sequences (i_k) and (j_k) with $i_0 = j_0 = 0$ such that for every k: $(e_k)_p = (c_{i_k})_p$ for every $p \in P$; $(e_k)_q = (d_{j_k})_q$ for every $q \in P'$; and if e_{k+1} exists then exactly one of $i_{k+1} = i_k + 1$ with $j_{k+1} = j_k$, a *P-step*, or $i_{k+1} = i_k$ with $j_{k+1} = j_k + 1$, a *P'-step*. If r is finite of length n then $i_k = n$ for some k, and if r is infinite then every index of r is reached by some i_k; likewise for r' and (j_k).

Definition A.2 (Interaction, Interactive Run, First Interaction [28]). Let $Q \subset \Pi$; for a configuration c over Q and $R \subseteq Q$, write $c|_R$ for the restriction of c to R. A transition $e \to e'$ of $\mathcal{F}(Q)$ is an *interaction between $x \neq y \in Q$* if $e|_{\{x\}} \neq e'|_{\{x\}}$ and $e|_{\{y\}} \neq e'|_{\{y\}}$, and $e|_{\{x\}} \to e'|_{\{x\}}$ is not a transition of $\mathcal{F}(\{x\})$ or $e|_{\{y\}} \to e'|_{\{y\}}$ is not a transition of $\mathcal{F}(\{y\})$. It is an *interaction between* disjoint nonempty $P, P' \subseteq Q$ if it is an interaction between some $x \in P$ and some $y \in P'$. A run of $\mathcal{F}(Q)$ is *interactive between P and P'* if some transition of it is an interaction between them, and the *first interaction* of P and P' in it is the first such transition.

Definition A.3 (Oblivious, Interactive, Grassroots [28]). A protocol $\mathcal{F}$ is *oblivious* if for every $P, P' \subset \Pi$ disjoint and nonempty every interleaving of a correct run of $\mathcal{F}(P)$ and a correct run of $\mathcal{F}(P')$ is a correct run of $\mathcal{F}(P \cup P')$; *interactive* if for every $Q \subset \Pi$ and $P, P' \subseteq Q$ disjoint and nonempty, every prefix of a correct run of $\mathcal{F}(Q)$ containing no interaction between P and P' is a prefix of a correct run of $\mathcal{F}(Q)$ that contains one; and *grassroots* if it is oblivious and interactive.

A protocol specified by transactions over a local-states function S has, for each $P \subset \Pi$, the set $R(P)$ of its transactions over some $Q \subseteq P' \subseteq P$ and $S(P')$; the set is *contained* if for every $P \subset \Pi$ every transaction in it with participants $Q \subseteq P$ and source in $S(P)^Q$ is in $R(P)$ [28]. An agent may leave — a person by death or incapacity, an AI agent by retirement — and interactivity is then asked only of groups each holding an agent that has not left.

Definition A.4 (Left, Interactive Among the Agents That Have Not Left [28]). Let $\mathcal{F}$ be a protocol specified by volition-guarded transactions R, $Q \subset \Pi$, c a configuration of $\mathcal{F}(Q)$ and $x \in Q$. Then x *has left* in c if no $(t, Q') \in R(Q)$ with $x \in Q'$ is machine-enabled in any configuration c' of $\mathcal{F}(Q)$ with $c \xrightarrow{*} c'$. $\mathcal{F}$ is *interactive among the agents that have not left* if for every $Q \subset \Pi$ and disjoint nonempty $P, P' \subseteq Q$, every prefix of a correct run of $\mathcal{F}(Q)$ containing no interaction between P and P', in whose last configuration some agent of P and some agent of P' have not left, is a prefix of a correct run of $\mathcal{F}(Q)$ that contains one.

Definition A.5 (Volitionally Grassroots [28]). A protocol $\mathcal{F}$ specified by volition-guarded transactions is *volitionally grassroots* if it is oblivious, interactive among the agents that have not left, and, for every $P, P' \subset \Pi$ disjoint and nonempty and every safe run of $\mathcal{F}(P \cup P')$ interactive between P and P', the first interaction of P and P' is induced by a volition-guarded transaction (t, Q') with $Q' \cap P \neq \emptyset$ and $Q' \cap P' \neq \emptyset$.

Two groups thus become connected only by an act a person of each wills; an interactive protocol is interactive among the agents that have not left, so a grassroots protocol meeting the condition

on first interactions is volitionally grassroots. An else transaction (Definition 2.6) is over a single participant and so is never an interaction; a binary transaction with an else-branch would let an agent of another group, who never wills it, trigger it, and could be a first interaction willed by neither, which is why Definition 2.5 allows only unary ones.

Two results of [28] are used below and stated without proof. The first is that an agent's volitional state, in any configuration of $\mathcal{F}(P)$, holds only classes of transactions whose participants lie in P. The second reduces obliviousness to a condition on those classes.

Proposition A.6 ([28]). *A protocol specified by a contained set of volition-guarded transactions is oblivious provided that for every $P, P' \subset \Pi$ disjoint and nonempty, no equivalence class whose transactions have participants in both P and P' is ever enabled in any interleaving of correct runs of $\mathcal{F}(P)$ and $\mathcal{F}(P')$.*

A.1 An Example: Child-Safe Befriending

The example the Introduction summarises, as volition-guarded transactions.

Example: Volition-guarded atomic child-safe befriending. Child-safe befriending [54] extends the grassroots social graph [51], the foundational grassroots platform on which all other grassroots platforms are built. Each agent p holds: a set $c_p \subseteq P$ of its friends; sets $\rho_p, \kappa_p \subseteq P$ of the parties that declared themselves its parents and of those it declared itself a parent of; and natural numbers α_p and ω_p counting the acts by which a child of its gained, or lost, a friend or a group. All are initially empty or zero. A party is an *adult* when $\rho_p = \emptyset$ and a *child* otherwise. In each transaction below, all components not explicitly updated are preserved.

(1) **Child befriend**(r, s, p, q), where $p \in \rho_r$, $q \in \rho_s$, $q \in c_p$ or $q = p$, $s \notin c_r$ and $r \neq s$: $c'_r := c_r \cup \{s\}$, $c'_s := c_s \cup \{r\}$, $\alpha'_p := \alpha_p + 1$ and $\alpha'_q := \alpha_q + 1$, guarded by $\{r, s, p, q\}$.
(2) **Child unfriend**(r, s, p, q), where $p \in \rho_r$, $q \in \rho_s$, $s \in c_r$ and $r \neq s$: $c'_r := c_r \setminus \{s\}$, $c'_s := c_s \setminus \{r\}$, $\omega'_p := \omega_p + 1$ and $\omega'_q := \omega_q + 1$, guarded by any one of $\{r, s, p, q\}$.

Child befriending requires the two children and one parent of each to be willing, and requires the two named parents to be friends already. Child unfriending is licensed by any one of the four, all four being its participants: each parent's count grows by one and names nobody, a parent holds no record of whom its child befriended, and any parent of either child may end any friendship of that child. When $p = q$ — two children of one parent — the participants are $\{r, s, p\}$ and α_p grows by one. The transactions by which parties declare parenthood are omitted: the example assumes the parents declared.

B Proofs

This appendix proves the results of Sections 3, 3.2, 5, 5.4 and 6.

Proof of Proposition 3.6. Let $\mathcal{F}$ be the protocol of a CVA platform (Definition 3.4). We show first that in every configuration of a safe run of $\mathcal{F}(Q)$ and for every $p \in Q$, every message in o_p has sender p and a recipient in $known_p$. It holds initially, o_p being empty. An app transaction adds to o_p only messages message(p, r, x) with $r \in known_p$ and leaves $known_p$ as it is; Communicate removes a message from o_p; Discover and Introduce add to $known_p$; a volition change leaves both as they are. And an introduction intro(q) that an agent r sends has $q \in known_r$ (Definition 3.4), and, knowing never being undone, r knows both q and the recipient from then on.

Second, which transitions are interactions. Let $P, P' \subseteq Q$ be disjoint and nonempty. An interaction between P and P' is an interaction between some $p \in P$ and $q \in P'$, and changes the states of

both (Definition A.2); a unary transaction changes the machine state of its one participant, and a transaction taken removes its class from volitional states, which hold only classes of transactions their agent participates in (Definition 2.3); so an interaction is a transition of a binary transaction with a participant in each group, a Discover, an Introduce or a Communicate between some $p \in P$ and $q \in P'$. A Discover(p, q) or an Introduce(p, q) is an interaction: it changes the states of both, and it takes $known_p$ to a set containing $q \neq p$, which no machine state in $S(\{p\})$ holds, so $e|_{\{p\}} \rightarrow e'|_{\{p\}}$ is not a transition of $\mathcal{F}(\{p\})$. Hence in a prefix of a safe run of $\mathcal{F}(Q)$ with no interaction between P and P', $known_p \cap P' = \emptyset$ for every $p \in P$, since $known_p$ grows by Discovers and Introduces of p only and one of p with an agent of P' is an interaction, and likewise $known_q \cap P = \emptyset$ for every $q \in P'$.

Volitionally grassroots. Let $P, P' \subset \Pi$ be disjoint and nonempty, r a safe run of $\mathcal{F}(P \cup P')$ interactive between P and P', and its first interaction between $p \in P$ and $q \in P'$. A Communicate(p, q, m) requires $m = \mathsf{message}(p, q, x) \in o_p$, so $q \in known_p$ by the first part, where $known_p \cap P' = \emptyset$ by the second: it is not the first interaction, and nor is a Communicate(q, p, m). An Introduce(p, q) requires an introduction $\mathsf{intro}(q)$ in i_p, sent by an agent $r \in P \cup P'$ that by the first part knows q and p: if $r \in P$, $q \in known_r \cap P'$, and if $r \in P'$, $p \in known_r \cap P$, against the second part; so it is not the first interaction either. The first interaction is therefore a Discover(p, q), induced by the volition-guarded transaction guarded by $\{p, q\}$, which meets both P and P'.

Oblivious. The transactions of the platform are contained (Appendix A): a transaction with participants $Q \subseteq P$ and source in $S(P)^Q$ has its targets in $S(P)^Q$, since Discover and Introduce add to *known* only participants, an app transaction adds to the outbox only messages to known peers and leaves *known* as it is, and the app states are A whatever P is; so it is a transaction over Q and $S(P)$. By Proposition A.6 it then suffices that no class of transactions with participants in both P and P' is enabled in an interleaving of correct runs of $\mathcal{F}(P)$ and $\mathcal{F}(P')$. Such a class is of a Discover, an Introduce or a Communicate between some $p \in P$ and $q \in P'$, and in an interleaving the state of p is a state of $\mathcal{F}(P)$. Discover(p, q) is enabled only if p wills it, and the volitional state of p in $\mathcal{F}(P)$ holds only classes of transactions whose participants lie in P (Definition 2.3), which q does not. Communicate(p, q, m) is enabled only if $\mathsf{message}(p, q, x) \in o_p$, and $o_p \subseteq M(P)$ in $\mathcal{F}(P)$, whose messages have their recipients in P. Introduce(p, q) is enabled only if an introduction $\mathsf{intro}(q)$ is in i_p, sent in $\mathcal{F}(P)$ by an agent that knew q, and the known peers of a state of $\mathcal{F}(P)$ lie in P, which q does not. None is enabled, and the same holds with P and P' exchanged.

Interactive. Let $Q \subset \Pi$, $P, P' \subseteq Q$ disjoint and nonempty, and $\hat{r}$ a prefix of a correct run of $\mathcal{F}(Q)$ with no interaction between P and P', ending in e; take $p \in P$ and $q \in P'$. By the second part $q \notin known_p$ and $p \notin known_q$ in e, so Discover(p, q) changes both their states, and it is enabled once both will it. Extend $\hat{r}$ by a volition change of p adding [Discover(p, q)], one of q adding it, and the Discover, an interaction between P and P'. The result is a finite safe run of $\mathcal{F}(Q)$ with $\hat{r}$ as a prefix, and by the Extension Lemma of [28] a prefix of a correct run of $\mathcal{F}(Q)$.

$\mathcal{F}$ is thus oblivious and interactive, hence grassroots; interactive, it is interactive among the agents that have not left (Definition A.4), and with the first interaction willed on both sides it is volitionally grassroots. □

Six facts about the runs of the canonical CVA platform (Definition 3.17) are used in the proofs of Proposition 3.20 and Theorem 3.24.

(*i*) *Votes and outcomes.* No transaction removes a term from a record, and a participant votes at most once on an attempt: an initiator proposes an attempt once, its number growing, and votes on it as it proposes it, and tells start(α) once to each other participant, which reads it once, by **Start**; **Start** votes, records a question, or, the attempt closed, neither; **Answer** records one answer to a pending question; and **Vote** votes on it once it is answered, which ends it. So an attempt once

decided stays decided, and one on which a participant votes no is never decided. An attempt is closed at its initiator only by its **Propose**, where the initiator votes no on it, and otherwise by the first **Tally**, **Commit**, **Yield** or **Wound** that closes it, each requiring it not closed and telling each other participant one outcome; at any other participant it is closed only by that outcome's **Outcome**.

(*ii*) *Locks.* $L_p = \alpha$ exactly while p has recorded voted(α, yes) and not closed(α): a vote yes, and **Propose** voting yes, set $L := \alpha$ and require $L = \perp$ — **Propose** by its agent being free, **Start** by its start not waiting, **Vote** explicitly — and L returns to $\perp$ only as p closes α, by **Tally**, **Commit**, **Yield** or **Wound** at its initiator and by **Outcome** elsewhere. While $L_p = \alpha$, s_p does not change: s changes only by the **Commit** of an attempt at its initiator and by the **Outcome** that commits an attempt at another participant, an attempt that p, being decided, voted yes on, so that L_p is that attempt from that vote to that **Commit** or **Outcome**. A vote yes being cast at a state in the domain of its effect, $s_p \in dom(f_p^{[t]_j})$ while L_p is an attempt of $[t]_j$, so the settled state applies the effect of every decided lock.

(*iii*) *Questions.* At most one question of a class is pending at an agent; while a question of $[t]$ is pending at p, L_p is no attempt of $[t]$, and no approval of $[t]$ other than p's answer to it is pending at p; and only a guard of a part with no opener is asked a question. **Start** records a question of $[t]$ only where its start does not wait, so with p free for $[t]$: $L_p = \perp$ and none pending; only where no approval of $[t]$ is pending at p; and only for an attempt naming no request, of a part with no opener. While it is pending, **Propose** and **Start** lock p on no attempt of $[t]$ and record no second question of $[t]$, and **Request** makes no approval of $[t]$, each requiring p free for $[t]$; **Vote** on another question locks p on the attempt of that question, of another class; and **Vote** on it ends it.

(*iv*) *Peers.* Discover and Introduce are the only transactions that change *known*, and each adds each of two agents to the other's, so knowing is symmetric and never undone. **Propose** requires the initiator to know every participant, so every participant an attempt reaches knows its initiator, and the proviso that p knows the agents it sends to holds of every **Start**, **Vote**, **Tally**, **Yield** and **Wound** whose other conditions hold in a configuration of a run, **Wound** sending a yield to the initiator of an attempt whose start its agent has read.

(*v*) *Priorities.* For every attempt α, finitely many attempts have priority over α, and each attempt is the α of finitely many **Wound** transactions. The second first: an attempt γ is the α of a **Wound** at q only while its start waits at q, or its question, answered yes, is pending there, once each; while it does, q records yielded(β) or closes β at the first **Wound** for each β blocking it, and q comes to be blocked by an attempt γ has priority over only by **Vote** on a question pending before, finitely many: **Start** of such an attempt waits behind γ, so that no new question of it is raised, a question of it answered yes waits behind γ where γ is a question, and q, not clear, proposes nothing. Now the first. A rank is either the number of the **Propose** that sets it, *fresh*, or that of the initiator's last attempt of the same part, recorded wounded; an agent's numbers grow, so finitely many fresh ranks are at most the rank of α, and with P finite finitely many pairs of an initiator and a rank give priority over α. The attempts of such a pair are of one part, and each but the last is wounded, by a **Yield** or a **Wound** for an attempt with priority over it, of a pair with priority over it. Were some pair to have infinitely many attempts, the one of highest priority among them would be wounded infinitely often by the finitely many attempts of the pairs with priority over it, each wounding finitely often by the second part: so none has.

(*vi*) *Approvals.* Every member g of the guard set of an attempt α of $[t]$ that votes yes on α has an approval of $[t]$ pending as it votes, and it stays pending until α is decided or closed at its initiator. The first: the initiator, a guard where the part of α has no opener, proposes α with one pending; the guard of an attempt that names a request votes yes only while it is pending; and an asked guard

otherwise votes yes by **Start** only where one is pending, or by **Vote** on its answer yes, which is one. The second: an approval of $[t]$ pending at g ends only as g votes no on an attempt of a part of $[t]$ it guards, or closes an attempt of $[t]$ committed, or refused and either naming that approval or, naming none, one that g, a guard of its part, voted or answered yes on. From its vote yes on α until α closes at g, $L_g = \alpha$ (*ii*), so no other attempt of $[t]$ is decided, each needing g's yes, and g votes on no other attempt of $[t]$, a start of one waiting while g is not free for $[t]$ and no question of $[t]$ being pending at g (*iii*), any attempt it voted yes on before having closed at g before g voted on α. And an attempt naming the request that α names is refused only at their initiator, the opener of its part, which then deletes that request, so proposes no later attempt naming it, and proposes nothing while α is open.

Proposition 3.20 (Atomic Commitment). *In every safe run of $\mathcal{P}$: (*agreement*) no attempt is committed at one participant and aborted at another; and (*validity*) an attempt is committed at a participant only if it is decided, each participant having voted yes on it holding no other attempt prepared, at a machine state in the domain of its effect, and each member of its guard set having voted yes on it with an approval of its class pending, made by a **Request** or an answer yes, which its person willed. In every safe, live run of $\mathcal{P}$: (*termination*) every attempt is eventually closed at each of its participants.*

Proof. Fix a safe run of $\mathcal{P}$ and an attempt $\alpha = (i, k, \pi, [t]_j, r)$ proposed in it.

Agreement. By (*i*), α is closed at its initiator once, and at every other participant by the outcome the initiator tells it on closing α: committed everywhere if the initiator commits it, and aborted everywhere if it aborts it.

Validity. α is committed at its initiator by its **Commit**, which requires a yes of every other participant recorded, each recorded by its voter as it cast it; the initiator's own was recorded by its **Propose**. So α is decided, and by agreement it is committed elsewhere only if committed at its initiator. Every vote yes is cast with $L = \bot$ at a state in the domain of its voter's effect: **Propose** requires its agent free and votes yes only there, **Start** requires its start not waiting and checks the state, and **Vote** checks both. By (*vi*) every member of the guard set of α voted yes on it with an approval of $[t]$ pending, made by a **Request** or by the then-branch of **Answer**, each guarded by its agent, a member of its approval class or of its request class.

Termination, the run being safe and live. Once its initiator closes α, it tells every other participant the outcome, which reaches it by liveness and is read by **Outcome**, closing α there; so it suffices that every attempt is closed at its initiator. Call an attempt *stuck* if it is never closed at its initiator; suppose some is, and let $\alpha = (i, k, \pi, [t]_j, r)$ be a stuck attempt over which no stuck attempt has priority, one existing as finitely many attempts have priority over any (*v*). Its initiator voted yes on it, a no closing it at once. If α has one participant, **Commit** is enabled from its **Propose** until taken, and liveness takes it. Otherwise its initiator tells start(α) to every other participant q, and the Communicate of that message is enabled until taken, so by liveness it reaches q, which knows i (*iv*). There the start waits only while q's inbox holds the start of an attempt with priority over α, or a question with priority over α and answered yes is pending at q — finitely many such attempts, none stuck — or while q is blocked by an attempt β. If α has priority over β, **Wound** is enabled until taken or β is closed at q, and it closes β or sends a yield to its initiator, whose **Yield** closes β unless it is closed already; if not, β is not stuck; either way β is closed at its initiator and, its outcome reaching q, at q. While the start waits, q is blocked anew only by **Vote** on a question pending before, finitely often, as in (*v*); so from some configuration on the start does not wait, and **Start** on it is enabled until taken. So liveness takes it: q votes on α or records the question α. In the last case α is the one question of $[t]$ pending at q (*iii*), and while it is pending and unanswered, q's approval class on $[t]$ is enabled — by the then-branch of **Answer** where q wills that class and

by its else-branch where not — with no other member enabled, **Request** requiring q free for $[t]$; so liveness takes **Answer** unless α is closed at q first, and then **Vote**, the answer yes waiting, as the start did, only finitely long. Each vote reaches i likewise and is read by **Tally**. No participant votes no, and no **Yield** or **Wound** closes α, since each would close it; and no participant closes α, i telling no outcome; so every participant votes yes, the **Tally** of the last of these votes records a yes of every other participant, and **Commit**, enabled from then until taken, closes α, against the supposition. So no attempt is stuck. □

The mapping for Theorem 3.24 reads the specification's machine states off the settled states and its volitions off the volitional states and the approvals.

Definition B.1 (CVA Implementation). The *CVA implementation* of a local volition-guarded specification over $P \subset \Pi$ is the function σ taking a configuration c of its canonical CVA platform over P to the agent configuration over P with, at each $p \in P$, machine state the settled state of p in c, and volitional state the set of classes $[t]$ of which p is a guard such that L_p is not an attempt of $[t]$ decided in c and either (A) the volitional state of p holds its approval class on $[t]$, or its request class on a request holding $[t]$, or (B) an approval of $[t]$ is pending at p.

Theorem 3.24 (CVA Implementation). *The implementation of volition-guarded specifications by CVA is correct, and complete for strongly fair runs.*

Proof. Fix a local volition-guarded specification and $P \subset \Pi$, with $\mathcal{P}$ and $\mathcal{F}(P)$ as in Section 3.2, and let σ be the mapping of Definition B.1. The initial configuration has no locks, no approvals and no wills, so σ maps it to the initial agent configuration: every settled state is s_0 and every volitional state is empty; $(\mathcal{P}, \sigma)$ is an implementation of $\mathcal{F}(P)$ (Definition 2.14).

Safety, per transition $c \to c'$.

The deciding transition. Let the transition decide an attempt $\alpha = (i, k, \pi, [t]_j, r)$: the **Propose** of α voting yes, if $Q([t]) = \{i\}$, and otherwise the **Tally** at i that records the last of the other participants' yes votes. In the first case, before it $L_i = \perp$ and $s_i \in dom(f_i^{[t]_j})$, and after it $L_i = \alpha$, decided. In the second, before it every participant q of α has voted yes on α and has $L_q = \alpha$ by (*ii*), α not yet being closed at q — its initiator closes it, telling the outcome, only by a **Tally** of a no, which no participant of α casts (*i*), by a **Yield** or **Wound** before it records every yes, or by **Commit** after — so its settled state is s_q, in $dom(f_q^{[t]_j})$ by (*ii*). Either way, after it the settled state of every participant q is $f_q^{[t]_j}(s_q)$, and no other settled state changes; by locality $d \to d'$, with $d_q = s_q$ and $d'_q = f_q^{[t]_j}(s_q)$, is a member of the part $[t]_j$ of $[t]$, over $Q([t])$. Before it every member of G_α holds $[t]$ under σ by clause (B), an approval of $[t]$ being pending at it by (*vi*), with L_q undecided. After it no agent holds $[t]$: every participant has $L_q = \alpha$, decided, which fails both clauses; and the guards of $[t]$ are participants of it. No agent's holding of another class changes, the transition changing only the record and lock of i and the decidedness of α. So $\sigma(c) \to \sigma(c')$ is the volitional machine transaction induced by that member and G_α (Definition 2.4), (t', G_α) being in R for every member t' of $[t]_j$ (Definition 3.8), and it is enabled in $\sigma(c)$ (Definition 2.13).

Other votes yes. A **Propose** of an attempt of more than one participant voting yes, or a **Vote** or **Start** casting a yes on an attempt α of $[t]$, sets $L_p := \alpha$, undecided, decides no attempt, and fulfils no will, **Propose** being unguarded: no settled state changes, and an undecided lock leaves both clauses as they were: a stutter.

Approvals. A **Request** of p of ρ fulfils p's request class on ρ, or its approval class on $[t]$ where $\rho = \{[t]\}$, and the then-branch of **Answer** on a question of $[t]$ fulfils p's approval class on $[t]$; so p, which held each class the approval approves by (A) before — L_p being $\perp$ for a **Request**, which requires p free, and no attempt of $[t]$ for an **Answer** (*iii*) — holds it by (B) after. Nothing else σ

reads changes: a stutter. The else-branch of **Answer** fulfils no will and records the answer no, which σ does not read; its agent held $[t]$ by neither clause, the else-branch being enabled only where it does not hold its approval class on $[t]$, and no approval of $[t]$ being pending at it (*iii*): a stutter.

Votes no. A vote no, by **Start**, by **Vote**, or by a **Propose** that also closes its attempt, locks nothing, changes no settled state, its attempt never being decided (*i*), and fulfils no will; where its agent is a guard of the attempt's part it ends the agent's approvals of $[t]$, so that the agent may cease to hold $[t]$: a volition change of the agent, or a stutter, but for the approvals the closing ends, as below.

Closing. A **Propose** voting no, a **Tally**, a **Commit**, a **Yield**, a **Wound** of its agent's own attempt, or an **Outcome**, that closes α at p sets $L_p := \bot$ where $L_p = \alpha$, and applies p's effect where α is committed: if α is decided the effect moves from the settled state into s_p, and if not, α is aborted (Proposition 3.20) and the lock it releases carried no effect. No settled state changes. Where it ends approvals of p — on a commit, every approval of $[t]$, and on a refusal the request the attempt names or, naming none, every approval of $[t]$ where p voted or answered yes on the attempt — p may cease to hold their classes, and where it releases a decided lock, (*A*) may resume for $[t]$: a volition change of p, or a stutter.

The rest. A **Start** that records a question or reads the start of an attempt closed at its agent, a **Tally** that records a yes without deciding its attempt or reads the vote of an attempt closed at its agent, a **Wound** that sends a yield, and a **Yield** that reads the yield of an attempt closed or decided at its agent, change nothing σ reads, numbers and ranks included: stutters. Discover, Introduce, Communicate and **Introduce peers** change *known*, the messages and the record of introductions, which σ does not read: stutters. A volition change of p changes which approval and request classes p holds, hence at most which classes σ gives p, by (*A*): a volition change of p, or a stutter.

Every safe run of $\mathcal{P}$ therefore maps to a safe run of $\mathcal{F}(P)$, and a member of $[t]$ occurs in $\sigma(\rho)$ only at a transition deciding an attempt of $[t]$.

Liveness. Let ρ be a correct run of $\mathcal{P}$ — connected, unobstructed, steady and strongly fair, by its fairness condition — and suppose a class $[t]$ is enabled in every configuration of a suffix of $\sigma(\rho)$ in which no member of $[t]$ occurs. Where ρ is finite, *infinitely many configurations* below means its last configuration, as in its fairness condition (Definition 2.16). By the safety cases no attempt of $[t]$ is decided in the corresponding suffix of ρ, and enabledness gives, at each of its configurations, a part of $[t]$ that holds (Definition 3.21) and all of whose guards hold $[t]$ under σ, each of them for $[t]$: an enabled volition-guarded transaction (t', Q'), $t' \in [t]$, has a part of $[t]$ that contains t', and so holds, and whose guard set is contained in Q' (Definition 3.8). ρ being steady, one part $[t]_j$ of $[t]$ holds, and all its guards are for $[t]$, in every configuration of a suffix, which is the suffix from here on: the settled state of every $q \in Q([t])$ is in $dom(f_q^{[t]_j})$.

No refusals. A vote no on an attempt of $[t]_j$ cast in the suffix by **Start** or **Propose** on its agent's precondition, or by **Vote** on the answer yes, is cast with $L = \bot$ at a machine state outside the domain of its voter's effect, its settled state, which the holding of $[t]_j$ excludes; and a **Propose** by a guard of $[t]_j$ of another part of $[t]$ voting no requires its state in the domain of no guarded part of $[t]$ with no opener that it guards, which the holding of $[t]_j$ excludes too, $[t]_j$ having no opener where its guard proposes. A guard g answering no to a question of $[t]_j$ in the suffix is a guard of a part with no opener (*iii*), so its guarded parts of $[t]$ have no opener and no request class of g holds $[t]$; it holds no approval of $[t]$ (*iii*); being for $[t]$, it holds its approval class on $[t]$, and the else-branch is not enabled. The guard of an attempt of $[t]_j$ naming a request r votes no on it by **Start** only where r is not pending at it; r ended at a commit of an attempt of $[t]$, none being in the suffix, or at the refusal of an attempt naming r, at whose **Tally** the opener deleted r, so that the attempt was proposed before that refusal and closed at the opener before it, and the vote no refuses

nothing. So such refusals name requests ended before the suffix, finitely many, each named by one refused attempt, the opener deleting it at that refusal. So from some configuration c_1 of the suffix on, after the votes on the answers no recorded before the suffix, at most one at each participant (*iii*, *i*), no attempt of $[t]_j$ is refused.

Proposals. Infinitely many attempts of $[t]_j$ are proposed. First, every guard h of $[t]_j$ has an approval of $[t]$ pending in infinitely many configurations of the suffix. If not, from some configuration on none is pending at h, which, being for $[t]$, holds its approval class on $[t]$ or its request class on a request holding $[t]$ throughout, with its settled state in $dom(f_h^{[t]_j})$; ρ being unobstructed, h is free for $[t]$ and clear in infinitely many of these configurations, where its settled state is s_h and it is ready for $[t]_j$, and free for every class of the request, a guard of a part with an opener being asked no question (*iii*); there a **Request** of h of $\{[t]\}$, or of that request, is enabled, the request not pending at h, and h knowing the opener of $[t]_j$ where it has one, which, ρ being connected, knows every other participant of $[t]$ from some configuration on; so, ρ being strongly fair, h makes infinitely many requests holding $[t]$, each an approval of $[t]$ pending, against the supposition. Let g be the opener of $[t]_j$ if it has one, and otherwise a guard of $[t]_j$, ρ being connected, one that knows every other participant of $[t]$ from some configuration on. Where $[t]_j$ is unguarded, g has its settled state in $dom(f_g^{[t]_j})$ throughout the suffix and, ρ being unobstructed, is free for $[t]$, hence ready for $[t]_j$, and clear in infinitely many configurations; so a **Propose** of g for an attempt of $[t]_j$ is enabled in infinitely many, and, ρ being strongly fair, g takes infinitely many, no attempt of $[t]$ being decided. Where $[t]_j$ is guarded and has no opener, an approval of $[t]$ is pending at g in infinitely many configurations, by the first part, with its settled state in $dom(f_g^{[t]_j})$; ρ being unobstructed, g is free for $[t]$ and clear in infinitely many of them, where it is ready for $[t]_j$; so likewise g takes infinitely many **Propose** transactions for attempts of $[t]_j$. Where $[t]_j$ is guarded and has an opener, its guard h knows g, which knows it. A request of h, made with h knowing g, is delivered by liveness, and g holds it from then until it deletes it, at the refusal of an attempt naming it, which ends it at h as h reads the outcome, or at a commit, of which there is none in the suffix; and every approval of h of $[t]$ is such a request, h being asked no question (*iii*). So, by the first part, g holds a request of $[t]$ from h in infinitely many configurations; ρ being unobstructed, g is free for $[t]$ and clear in infinitely many of them; so a **Propose** of g for an attempt of $[t]_j$ naming such a request is enabled in infinitely many, and g takes infinitely many.

Decision. Every attempt of $[t]_j$ proposed after c_1 is closed at its initiator (Proposition 3.20), not by a refusal, of which there is none after c_1, nor by **Commit**, which follows its decision, against the hypothesis; so it is wounded, by a **Yield** or a **Wound**. Infinitely many being proposed, by the finitely many guards of $[t]_j$ or its opener, some agent g initiates infinitely many of them; each from its second on retries a wounded one, so all have one rank, and each is wounded in favour of an attempt with priority over it, of which there are finitely many, each the α of finitely many **Wound** transactions (v): so finitely many are wounded, a contradiction. Hence $\sigma(\rho)$ is live, and with safety, correct.

Completeness. Let $r = c^0, c^1, \ldots$ be a correct run of $\mathcal{F}(P)$ that is strongly fair. We build a correct run r' of $\mathcal{P}$ with $\sigma(r') = r$, in stages, keeping between stages the invariant that the current configuration e of $\mathcal{P}$ is *quiescent for* the current configuration c of r: every agent knows every other; every outbox is empty, every inbox holds introductions only, and no **Introduce peers** is enabled; every lock is $\perp$, every attempt proposed is closed at each of its participants, and no approval is pending; $s_p = c_p^m$ for every p; and the volitional state of p in e holds exactly p's approval classes on the members of c_p^v. A quiescent e has $\sigma(e) = c$: with no locks, the settled state of p is s_p, and clause (A) yields c_p^v; and every agent is clear in it, its inbox holding no start.

Start. From the initial configuration, for each pair $p \neq q$ of P in turn, a volition change of p and one of q add the class of Discover(p, q), and the Discover is taken; then each **Introduce peers** enabled is taken, with the Communicates of its introductions; each step is a stutter under σ, which reads neither *known*, the Discover classes, the messages nor the record of introductions, and the result is quiescent for c^0.

Step. Suppose e is quiescent for c^i and $c^i \longrightarrow c^{i+1}$ is a transition of r. If it is a volition change of p, take the volition change of p in $\mathcal{P}$ that replaces p's approval classes by those on the members of $c_p^{i+1,v}$: it maps to $c^i \longrightarrow c^{i+1}$, and the result is quiescent for c^{i+1}. If it is the volitional machine transaction induced by (t, Q'), $t = d \longrightarrow d'$ over Q, enabled in c^i, let $[t]_j$ be a part of $[t]$ that contains t and whose guard set $G = G([t]_j)$ is contained in Q' (Definition 3.8): every $q \in Q$ has $s_q = d_q \in dom(f_q^{[t]_j})$ and, being free, is ready for $[t]_j$, and clear; and every $g \in Q'$ holds $[t]$ in c^i, so it wills its approval class on $[t]$. Let i be the opener of $[t]_j$ if it has one and otherwise a member of G, and g the guard of $[t]_j$ where it has an opener; let $\alpha = (i, k, k, [t]_j, r)$, k one more than i's number after the request below and its rank, no attempt being wounded in r', whose guard set is G, and r the request of g below, or $\perp$. Take, in this order: where $[t]_j$ is guarded, the **Request** of $\{[t]\}$ by i to itself where i is a guard, or by g to i followed by the Communicate of the request; i's **Propose** of α, voting yes; if $Q \neq \{i\}$, for each other participant q in turn, the Communicate of start(α) to q, q's **Start**, which votes yes where $q = g$, its request pending, or $q \notin G$, and records the question where $q \in G$ and $q \neq g$, and, in that case, the then-branch of its **Answer** and its **Vote**, voting yes, q being ready, then the Communicate of the vote and i's **Tally** of it, the last of which decides α; a volition change of each participant $q \notin G$ that wills its approval class on $[t]$, removing it; i's **Commit**; and the Communicate of the outcome and of the introductions **Commit** sends to each $q \neq i$, and q's **Outcome**. Each step but one is a stutter, by the safety cases: the **Request** and the answers move their agents from clause (A) to (B), and the **Propose** and the votes before the last leave every agent's holding as it was; the Communicates and the earlier **Tally** transactions change nothing σ reads; the volition changes remove an approval class from an agent whose lock is on α, decided, so that it holds $[t]$ by neither clause before or after; and the **Commit** and the **Outcome**s move a decided effect into s and end the approvals of $[t]$, the agents they release willing no approval class on $[t]$. The one that is not is the **Tally** that decides α, of the last participant's yes, or **Propose** if $Q = \{i\}$, which by the safety case maps to the volitional machine transaction induced by $d \longrightarrow d'$ and G from $\sigma(e) = c^i$, which is $c^i \longrightarrow c^{i+1}$, the one induced by (t, Q'), G being contained in Q'. After the stage $s_q = c_q^{i+1,m}$ for every q, every lock is $\perp$, α is closed everywhere, no approval is pending and no request is held, and no agent holds an approval class on $[t]$, those of G having been fulfilled and the others removed, as c^{i+1} removes $[t]$ from every volitional state; and after each **Introduce peers** then enabled, taken with the Communicates of its introductions, stutters, the result is quiescent for c^{i+1}.

Idle. A guard g of a guarded part $[u]_j$ of a class $[u]$ not enabled in c^{i+1} that holds its approval class on $[u]$ and is ready for $[u]_j$ has a **Request** of $\{[u]\}$ enabled in a quiescent configuration, and the opener g of an unguarded part $[u]_j$ of such a class that is ready for $[u]_j$ has a **Propose**; neither may stay enabled forever. For each such pair $(g, [u]_j)$, taken in a fair order over the countably many of them — a pair enabled at infinitely many quiescent configurations being taken infinitely often — and interleaved with the stages above, run an *idle attempt.* Where $[u]_j$ is guarded, take first g's **Request** of $\{[u]\}$, to itself, or to the opener o of $[u]_j$ with the Communicate of the request, moving g from (A) to (B). Let the initiator i be g, or o where $[u]_j$ has an opener, and take its **Propose** of an attempt α of $[u]_j$, naming g's request where $i = o$; let $G = G_\alpha$. As $[u]$ is not enabled in c^{i+1}, some participant q refuses α with no will fulfilled: a participant not ready for $[u]_j$, its machine state outside the domain of its effect, votes no, by **Start**, or by **Propose** where it is o; and if every

participant is ready, a member of $[u]_j$ is machine-enabled, by locality, and some member q of G does not will its approval class on $[u]$, g willing its own and being the only guard where $[u]_j$ has an opener, so q, a guard of a part with no opener, is asked, and it records the question by **Start**, the answer no by the else-branch of **Answer**, and votes no by **Vote**. Where $q \neq i$, take the Communicate of start(α) to q, its transactions up to its vote, and the Communicate of the vote. Then, where $[u]_j$ is guarded, take a volition change of g adding its approval class on $[u]$ back; then, where $q \neq i$, i's **Tally** of the vote, which aborts α, $q = i$ having aborted it by its **Propose**; and, for every participant other than i, the Communicate of the outcome and its **Outcome**, and, for every one whose start is still undelivered, then the Communicate of start(α) and its **Start**, which does nothing. Every step is a stutter: g holds $[u]$ under σ throughout, by (B) from its **Request** and by (A) as well once its approval class is back, which is before the refusal ends its approval; no vote yes is cast but the initiator's; and no other agent's holding changes. The result is again quiescent for c^{i+1}. A pair whose class is enabled in c^{i+1} is not idled there.

Finite runs. If r is finite, the stages after its last configuration consist of idle attempts.

The run r' so built is safe, and $\sigma(r')$, with stutters removed, is r. Each stage is finite and begins and ends at a quiescent configuration, where every agent is free for every class and clear; within a stage an agent's settled state changes only at the deciding vote, and its approval classes only by fulfilment, by the removals that follow that vote and by an idle attempt's restoring one; so an agent whose settled state is in the domain of its effect in a part and which holds its approval class on the part's class, has an approval of it pending, or opens the part, at a configuration of a stage does so at the stage's first or last configuration, or the stage is an idle attempt of that part. A **Propose** or **Request** enabled at a configuration of a stage is likewise enabled at its first or last configuration, except a **Propose** that needs an approval or a request, which is enabled only within the stage that makes the approval, and is taken there. Every attempt is closed within its stage, every message sent but an introduction is read within it, an introduction is delivered within it, and every question raised is answered within it. So r' is connected, every agent knowing every other; unobstructed, its configurations of an agent at a state in the domain of its effect in a part that meet a condition of Definition 3.21 lying in infinitely many stages if they are infinitely many, each ending free and clear; steady, since at its quiescent configurations, where the settled states are the machine states of r and an agent is for a class exactly when it holds its approval class, a part of a class holds with all its guards for the class only where the class is enabled in r, so that a class for which this holds from some configuration on, with no attempt of it decided, would stay enabled in r with no member taken, against r being live; and strongly fair and live: a Discover class is enabled only at the start; an Introduce never is, every agent knowing every other; an **Introduce peers** class is taken in the stage that enables it; a **Request**, or the **Propose** of the opener of an unguarded part, enabled at infinitely many configurations is enabled at infinitely many quiescent ones, where it is taken by an idle attempt in its turn, at those where its class is not enabled in r, or its class is enabled in infinitely many configurations of r, which, being strongly fair, takes it in infinitely many, each a stage that decides an attempt of the class with its agent as a participant, whose lock keeps it disabled there; and a **Propose** that needs an approval or a request is taken in the stage that enables it. Hence r' is a correct run of $\mathcal{P}$ under its fairness condition with $\sigma(r') = r$, and σ is complete for strongly fair runs. □

Theorem 5.14 (Volitional Soundness). *Let $c^1 \to c^2 \to \cdots$ be a proper safe run of the vmaGLP transition system over P and M, $p \in P$, and X a question of p's person in c^i. Then the writer X occurs in c^i in H_p and nowhere else, and a transition $c^i \to c^{i+1}$ that adds an assignment $X? := t$ to a readers substitution is the Answer of p to X.*

Proof. We show first that in every configuration c^k of the run (*i*) every writer a person holds occurs in that person's H and nowhere else, and (*ii*) every writer in the term of an assignment in a readers substitution occurs in that term once and nowhere else. In c^1 no person holds a variable. Suppose both hold in c^k. An Ask introduces a writer occurring nowhere in c^k and gives its person one end of it, the other end going to the asked goal. An Answer to a question removes it from the variables its person holds, adds an assignment to its reader, which the person does not hold, and introduces variables occurring nowhere in c^k: writers the person holds, their readers in the answer, and writers in the answer for the program to write, their readers held by the person. A Communicate to a reader a person holds gives that person the variables of the term $\hat{T}$ it delivers and the assignment, shown; each writer of $\hat{T}$ occurred, by (*ii*), once in the term of one of the assignments used in forming $\hat{T}$ and nowhere else, and those assignments, communicated with it, are removed (Definition 4.10). A Reduce at an agent q renames its clause apart, and the run being proper, the variables of the renamed clause occur nowhere in c^k or earlier, at any agent; its writer mgu $\hat{\sigma}$ binds writers of the goal and of the renamed head, none of them held by a person, and a writer in a term it assigns is a variable of the renamed clause or of the goal, occurring in that term once and nowhere else, by SRSW and by the SO of the goal, which the reduction removes. Any other Communicate, and a Cold-call, put into a goal a term already in c^k, which by (*i*) holds no writer a person holds. Hence both hold in c^{k+1}, and (*i*) holds in c^i.

Now let $c^i \longrightarrow c^{i+1}$ add $X? := t$ to a readers substitution. An Ask, a Communicate and a Cold-call add no assignment. A Reduce adds $\hat{\sigma}?$, whose domain is the readers of the writers $\hat{\sigma}$ binds, writers of the goal and of the renamed head, and by (*i*) X is neither. So the transition is an Answer to a question Z, adding $Z? := t$; hence $Z = X$, and X being among the questions of p's person and of no other, it is the Answer of p to X. □

Theorem 5.19 (Compilation of the Canonical Platform). *For every effective local volition-guarded specification and $P \subset \Pi$, the platform mapping is a correct implementation of $\mathcal{P}$, the transition system of its canonical platform with a computable introduction rule or none over P under its fairness condition (Definition 3.22), by the vmaGLP transition system over P and $\lceil\Theta\rceil$, under the fairness condition that a run is proper, no approval question stays open in it, its calendar does not stop where the specification requires the day, and its image under the platform mapping is connected, unobstructed and steady.*

Proof. Fix an effective local volition-guarded specification, its canonical platform Θ and $P \subset \Pi$; write TS' for the vmaGLP transition system over P and $\lceil\Theta\rceil$, and σ for the platform mapping. σ maps the initial configuration of TS' to that of $\mathcal{P}$: each agent's resolvent is the initial goal, so no `hello` is sent, no message is on a stream, nothing is read and nothing is willed, and the agent goal, once called, holds $(s_0, 0, \bot, \emptyset)$.

Four facts about the configurations of a safe run of TS'. First, a request, a message and an answer each pass from where they are written — the person's answer, the network input stream, the answer to an approval question — through the merge to the head of the agent goal's stream, where the clause of their kind reads them and computes in its body the condition of the transaction they put up, on the agent state the clause was called with; the agent goal is one goal, so no dispatch intervenes before the dispatch procedure's reduction, and the app state and the starts held waiting do not change between the reading of a trigger and its dispatch. Second, where a hello is `hello` or `ihello`, the two of an exchange being of one kind, a peer an agent has sent a hello to and received one from is one with which it has exchanged hellos in σ, a hello received having been sent, and not conversely until the peer's hello arrives: a condition's test that a peer is known, true at the agent, is true in σ. Third, an agent reads the start of an attempt only after it has sent its sender a hello and read the sender's: the sender sends only to a peer it has received a hello from and sent its own

to, which precedes the start on its output stream, and the messages of one stream are delivered and read in order; so an agent knows the sender of every start it reads, and, knowing being never undone, every agent **Start**, **Vote** and **Tally** send to. Fourth, an agent sends a cargo to a recipient at most once, by fact (*i*) of the proofs for Section 3.2 and its records of yields and introductions, so no two messages on the streams are one message of $\mathcal{P}$.

Safety, per transition of TS'. An Ask, a Communicate to a reader the person holds or of an assignment along a stream, a Reduce of a merge goal, the agent's guarded clause, the clause that asks an approval question, the goal it hands its answer to, or a goal computing a condition or an effect, and a Cold-call, an app message it delivers staying in o_p until its recipient's agent goal reads it, change nothing σ reads: stutters. An Answer of p to its request question is a stutter, the request being unread, except with a request of a acquaintance part with $q \in P$: then it adds [Discover(p, q)], a volition change, unless $q \in known_p$ or the class is already there; the person's yes to the question an arriving `hello` asks is such a request, and their no a stutter. An Answer to an approval question is a stutter, the answer being unread.

A Reduce of the agent goal's clause reading a request `propose(`N`)`, N naming a guarded part that p guards, adds to p's volitional state its approval class on $[t]$ where the request puts up the **Request** of $\{[t]\}$ and its request class on ρ_N where it puts up the **Request** of ρ_N of two or more classes, and one reading the answer yes to the approval question of an attempt of $[t]$ adds p's approval class on $[t]$ where the condition of the **Answer**, but for its test of known peers, holds at p's machine state: a volition change of p, the agent goal reading one trigger at a time; otherwise it is a stutter, and so is a dispatch that holds a request or puts one up again. Reading an app message m that q sent, it is Communicate(q, p, m), m leaving o_q and entering i_p; reading a hello, an answer no or an own trigger, it is a stutter.

A Reduce of a dispatch clause taking a transaction is that transaction of $\mathcal{P}$ at the configuration σ reads: its condition holds there, by the first fact on the app state and the inbox and by the second and the third on the peers it requires known. **Request** and the then-branch of **Answer** are guarded by p, and p's approval class on $[t]$, or its request class on ρ_N, is in its volitional state, read with its condition holding, so the transaction is enabled, and the reduction removes the class, fulfilled (Definition 2.4). The else-branch of **Answer**, taken on the answer no, is enabled, p holding no approval class while the agent goal reads an answer no. **Start**, **Tally**, **Yield** and **Outcome** are unguarded and take a message in i_p, which they remove; **Propose**, **Vote**, **Commit** and **Wound**, taken as its own transactions, are unguarded, **Wound** leaving the start it reads in i_p, and a **Propose** naming a request it holds, in i_p, deleting it where it votes no. The new app state and the messages the effect yields, addressed to peers known at the agent and so in σ, are the transaction's effect. A Reduce of a dispatch clause holding a start waiting, or a request, is a stutter, the message staying in i_p. A Reduce of a dropping clause on a request or an answer yes is a volition change of p removing the class its reading added, and a stutter where it added none; on an answer no it is a stutter, and so is the agent goal's finding none of its own transactions holding. A Reduce of the request clause on a request of a acquaintance part with q sending `hello` is a stutter where q has not sent `hello` to p or $q \in known_p$ already, and otherwise Discover(p, q): both will it, each having a pending request of the acquaintance with the other, q's made by its person or by their yes to p's hello, with $q \notin known_p$, and after it $known_p$ and $known_q$ each gain the other and both volitional states lose the class. A Reduce of the clause reading an introduction `intro(`q`)` and sending `ihello` is a stutter where q has not sent `ihello` to p or $q \in known_p$ already, and otherwise Introduce(p, q): each has read an introduction of the other, both of which leave the inboxes, and after it $known_p$ and $known_q$ each gain the other; the reduction recording an arriving `ihello` is a stutter. A Reduce of a dispatch clause taking **Introduce peers** is that transaction, its introductions addressed to known peers. Every safe run of TS' therefore maps to a safe run of $\mathcal{P}$.

Liveness. Let r' be a proper correct run of TS' in which no approval question stays open and whose image $r = \sigma(r')$ is connected, unobstructed and steady. In r' every element written to a stream is read by the agent goal's clause of its kind: the Communicate of its assignment and the reductions of the merge goals are enabled until taken and the merge is fair, and once the element heads the agent goal's stream the clause of its kind reduces, by Monotonicity and the clause order as for the requests of Section 5; its condition is then bound, being computable and terminating, and the dispatch clause reduces. Own triggers are elements of that stream too, so the agent goal reads every other trigger however many own triggers it writes. It reads an own trigger after each other trigger it reads, the one it writes then or one it wrote before and has not read, and after each own transaction it takes, and on each it takes one of its own transactions whose condition holds, if one does, by a reduction of a dispatch clause, enabled until taken. Suppose a class of $\mathcal{P}$ is enabled in every configuration of a suffix of r and no member of it occurs there.

Communicate(p, q, m): m is on p's output stream, the Cold-calls of the messages before it and then of m are each enabled until taken, so m is delivered and q's agent goal reads it. The class of Discover(p, q) and Introduce(p, q): enabled through Discover, both will it, so each has a pending request of a acquaintance part with the other, and each request is read and its `hello` sent, if not before, q not being known at p nor p at q, the second sending being a Discover(p, q); enabled through Introduce, each has read an introduction of the other and so sent `ihello`, if not before, the second sending being an Introduce(p, q). **Tally**, **Yield** or **Outcome** on a message $m \in i_p$: m is read and not yet dispatched, and its dispatch takes it. **Start** on m: read and not yet dispatched, the dispatch takes it, m not waiting; held waiting, its waiting held at the state at which it was last put up and does not hold now, so a dispatch has since changed the app state or consumed a start, after which it was put up again and taken.

A **Propose**, a **Vote** on a question pending at p, a **Commit**, a **Wound**, or an **Introduce peers**: at each configuration of the suffix an instance of the class has its condition holding at the agent, where it tests a known peer once the peer's `hello` has arrived, which it does, the peer being known in r. Only the agent goal's reading of a trigger and its dispatches change that condition, and after each it reads an own trigger, so from the suffix's first configuration and the arrival of that `hello` on it reads own triggers without end and takes an own transaction on each, an instance of the class that has waited longest; a class taken waits from then on less long than this one, and the classes that have waited longer than it are finitely many — those taken, or found holding, before it began to wait, finitely many at each step, the unguarded parts an agent opens at a state, the approvals pending at it, the requests it holds, the questions pending at it, the attempts it initiated and the starts it holds being finitely many — so within finitely many of these steps the dispatch takes an instance of it.

p's approval class on $[t]$, whose members are p's **Request** of $\{[t]\}$ and its **Answer** on the questions of $[t]$, or its request class on a request ρ, whose members are its **Request** transactions of ρ. If **Request** is enabled in infinitely many configurations of the suffix, p holds the class in them, so in infinitely many a request `propose(`N`)` that puts up the **Request** of $\{[t]\}$ or of ρ is read or held at p, p being free for its classes and ready for a part it names; a held request is put up again after every dispatch, so it is dispatched at such a configuration within finitely many steps, and that dispatch takes the transaction unless a peer its condition tests is known in r and not yet at the agent, which, each hello arriving, happens finitely often. Otherwise, from some configuration on, **Answer** is enabled throughout, on a question α of $[t]$ pending at p and unanswered, at most one being pending, by fact (*iii*) of the proofs for Section 3.2; α is never closed at p, the configuration after its closing enabling neither **Request** of $\{[t]\}$, by the supposition, nor **Answer**, no **Start** having yet raised another question. So the approval question **Start** raised on α is neither answered and read

nor closed: it is asked, and is open until answered; it is answered, no approval question staying open; the answer is read; and the dispatch takes the then-branch on yes and the else-branch on no.

In every case a member of the class occurs in the suffix: r is live.

Fairness. r is connected, unobstructed and steady by hypothesis, and strongly fair. A **Request** of p to o of a request holding $[t]$ is enabled in r only while p holds its approval class on $[t]$ or its request class on the request, which by the safety case is while a request `propose(`N`)` that puts up a **Request** to o of a request holding $[t]$ is read or held at p; a held request is put up again after every dispatch, and its dispatch at a configuration where p is free for its classes and ready for a part it names takes, among the **Request** transactions it puts up, finitely many, the one it has taken least recently; so if such a **Request** is enabled in infinitely many configurations of r, the agent takes one at infinitely many of these dispatches, unless a peer its condition tests is known in r and not yet at the agent, which happens finitely often. A **Propose** is one of the agent's own transactions, and its condition's value at the agent changes only as the agent goal reads a trigger or takes a transaction, after each of which it reads an own trigger; so if a **Propose** of p for an attempt of a part $[t]_j$ is enabled in infinitely many configurations of r, its condition holds at infinitely many own triggers, and it is taken in infinitely many, each own trigger at which it holds taking it or one of the finitely many classes that have waited longer. So r is a correct run of $\mathcal{P}$ under its fairness condition. □

Proposition 5.20 (Grassroots Compilation). *For every effective local volition-guarded specification with acquaintance parts guarded by every two agents, and computable introduction rule or none, the protocol of the compilation of its canonical platform is grassroots (Definition A.3).*

Proof. Fix the specification, the rule, its canonical platform Θ and $M = \lceil\Theta\rceil$, and write $\mathcal{F}$ for the protocol of M. It is a transactions-based protocol [53]: its transactions are those of vmaGLP over M (Definition 5.8), each over its participants, and its local states over P grow with P. In a run of $\mathcal{F}(P)$ every variable at an agent of P is a variable of an agent of P: those of the initial goals are, a Reduce, an Ask and an Answer at p take new ones from $\hat{\mathcal{V}}_p$ (Definitions 4.7, 5.6 and 5.7), and Communicate and Cold-call move variables among the agents of P.

Oblivious. Let $P, P' \subset \Pi$ be disjoint and nonempty, and e an interleaving of a correct run of $\mathcal{F}(P)$ and a correct run of $\mathcal{F}(P')$. Reduce, Ask and Answer are unary. A Communicate between $p \in P$ and $q \in P'$ requires a variable pair shared by the two, whose variables would be of an agent of P and of an agent of P', and these sets are disjoint; and a Cold-call from p to q requires a message to q on the network output stream of p, an identifier of an agent not of P, and likewise from q to p. So no class whose transactions have participants in both P and P' is enabled in e. And e is proper, the two runs being proper and a clause renamed at p into $\hat{\mathcal{V}}_p$ having no variable of an agent of P'. By the obliviousness condition of [53], e is a correct run of $\mathcal{F}(P \cup P')$.

Interactive. Let $Q \subset \Pi$, $P, P' \subseteq Q$ disjoint and nonempty, $p \in P$ and $q \in P'$, and let $\hat{r}$ be a prefix of a correct run of $\mathcal{F}(Q)$ with no interaction between P and P' (Definition A.2). Extend $\hat{r}$ by the reductions and the Ask at p by which the person of p comes to hold an open request question, the agent goal asking one when it starts and the next when it reads the answer to the last (Section 5.4); by the person's Answer to it, a request of a acquaintance part with q, one being guarded by p and q; and by the reductions by which the agent goal reads that answer and sends q `hello`, unless it has sent one, which is then on its network output stream, undelivered, $\hat{r}$ having no interaction between P and P'. The Cold-call of that message from p to q changes the states of both, and no transaction of $\mathcal{F}(P)$ advances the network output stream of p past a message to an agent not of P, so the Cold-call is an interaction between P and P'. The result is a finite proper safe run, and taking the classes enabled in it in turn, in a fair order over the countably many, each agent taking new variables from its own set, extends it to a correct run of $\mathcal{F}(Q)$, which contains the interaction:

a Reduce class stays enabled until taken (Proposition 5.12), and so does a Communicate, an Ask and a Cold-call class.

$\mathcal{F}$ is thus oblivious and interactive, hence grassroots. □

Proposition 6.12 (Grassroots Cascade). *For every effective local volition-guarded specification with acquaintance parts guarded by every two agents, and computable introduction rule or none, the protocol of GLP with persons over $\lceil\lceil\Theta\rceil\rceil$ is grassroots.*

Proof. As that of Proposition 5.20, with GLP with persons over $\lceil\lceil\Theta\rceil\rceil$ in place of vmaGLP over $\lceil\Theta\rceil$: a grant is unary and its new writer is a variable of its agent (Definition 6.8), and the person of p answers the request question with a request of a acquaintance part with q by granting it on its construct, which the dispatcher routes and the construct process binds (Section 6.3). □

Theorem 6.3 (Elicitation Completeness). *In every configuration of a safe run of the vmaGLP transition system over P and M and every $p \in P$, a term offered for a question X at p is an answer to X that the Answer transition allows; and if the widget of the construct of X is complete, every answer the Answer transition allows for X is offered.*

Proof. A construct is presented at p only for a question X that p's person holds (Definition 6.1). A submission t it forms is a term of the type of X: a widget forms only terms of its type, the forming procedure of each construct process being typed (Section 6.3). The person's input fills the positions the person writes, but for the stream and list positions, at which the widget places new readers or, for a list the person ends, `[]`; and the positions the program writes hold new writers, whose readers the construct holds. So t is ground at the positions the person writes but for the stream and list positions, which is an answer the Answer transition allows for X (Definition 5.7). Conversely, if the widget of the construct of X is complete, every answer the Answer transition allows for X is formed by some input: it is offered. □

The mapping for Theorem 6.10 rests on what the dispatcher and the construct processes do. Here is the dispatcher, as the repository holds it in `programs/vglp/dispatcher.glp`, which the compilation emits into every compiled program under fresh names. `dispatch` reads the ask stream and the person channel; `fair` merges the asks, the person's grants and the constructs' draws into one stream, taking them in turn, and `serve` reads that stream and numbers the questions from 1. A program that declares `Output` is dispatched by the second `dispatch`, with the draws of construct `0`, on which the compiled program draws its output to its person; `merge_draws` merges them with the constructs' draws, and the merged stream never ends. On an ask, `serve` writes on its `Spawn` stream the question with a fresh identifier `N`, the reader of a new grant stream and the writer of a new draw stream, and the compiled program's `constructs/1` reads that stream and calls the construct process of the question's type; it routes each grant `input(Id, P, R)` to the construct with identifier `Id`; it passes the constructs' draws on; and it moves a withdrawn construct's grant writer to `Deads`, which it never reads, since the person channel never closes.

```
Inputs(T) ::= [Input(T) | Inputs(T)].
Route(T) ::= route(Integer, Inputs(T)?).
Routes(T) ::= [] ; [Route(T) | Routes(T)].
Deads(T) ::= [] ; [Inputs(T)? | Deads(T)].
Ask(Q) ::= ask(Constant, Q).
Spawn(Q, T) ::= spawn(Integer, Q, Inputs(T), Stream(Draw(T))?).

exported procedure(Q, T) dispatch(Stream(Ask(Q))?, PersonChannel(T)?,
    Channel(Stream(_), Stream(_)), Stream(Spawn(Q, T))).
dispatch(Asks, ch(PIn, POut?), ch(MIn?, _), Ss?) :-
    split_grants(PIn?, Inputs, MIn),
    fair(ask, Asks?, Inputs?, COut?, Es),
    serve(Es?, COut, POut, [], [], 1, Ss).
```

```
exported procedure(Q, T) dispatch(Stream(Ask(Q))?, PersonChannel(T)?,
    Stream(_), Stream(Draw(T))?, Stream(Spawn(Q, T))).
dispatch(Asks, ch(PIn, POut?), MIn?, ODs, Ss?) :-
    split_grants(PIn?, Inputs, MIn),
    merge_draws(ODs?, DOut?, POut),
    fair(ask, Asks?, Inputs?, COut?, Es),
    serve(Es?, COut, DOut, [], [], 1, Ss).

procedure(T) merge_draws(Stream(Draw(T))?, PersonOut(T)?, PersonOut(T)).
merge_draws([X | Xs], Ys, [X? | Zs?]) :- merge_draws_(Ys?, Xs?, Zs).
merge_draws(Xs, [Y | Ys], [Y? | Zs?]) :- merge_draws(Xs?, Ys?, Zs).
merge_draws([], Ys, Ys?).

procedure(T) merge_draws_(PersonOut(T)?, Stream(Draw(T))?, PersonOut(T)).
merge_draws_([X | Xs], Ys, [X? | Zs?]) :- merge_draws(Ys?, Xs?, Zs).
merge_draws_(Xs, [Y | Ys], [Y? | Zs?]) :- merge_draws_(Xs?, Ys?, Zs).
merge_draws_(Xs, [], Xs?).

procedure(T) split_grants(PersonIn(T)?, Inputs(T), Stream(_)).
split_grants([input(Id, P, R) | Gs], [input(Id?, P?, R?) | Is?], Ms?) :-
    split_grants(Gs?, Is, Ms).

Event(Q, T) ::= Ask(Q) ; Input(T) ; Draw(T).
Events(Q, T) ::= [Event(Q, T) | Events(Q, T)].
Turn ::= ask ; grant ; draw.

procedure(Q, T) fair(Turn?, Stream(Ask(Q))?, Inputs(T)?, Stream(Draw(T))?,
    Events(Q, T)).
fair(ask, [A | As], Is, Ds, Es?) :- Es = [A? | Es1?], fair(grant, As?, Is?, Ds?, Es1).
fair(grant, As, [I | Is], Ds, Es?) :- Es = [I? | Es1?], fair(draw, As?, Is?, Ds?, Es1).
fair(draw, As, Is, [D | Ds], Es?) :- Es = [D? | Es1?], fair(ask, As?, Is?, Ds?, Es1).
fair(ask, As, [I | Is], Ds, Es?) :- Es = [I? | Es1?], fair(draw, As?, Is?, Ds?, Es1).
fair(grant, As, Is, [D | Ds], Es?) :- Es = [D? | Es1?], fair(ask, As?, Is?, Ds?, Es1).
fair(draw, [A | As], Is, Ds, Es?) :- Es = [A? | Es1?], fair(grant, As?, Is?, Ds?, Es1).
fair(ask, As, Is, [D | Ds], Es?) :- Es = [D? | Es1?], fair(ask, As?, Is?, Ds?, Es1).
fair(grant, [A | As], Is, Ds, Es?) :- Es = [A? | Es1?], fair(grant, As?, Is?, Ds?, Es1).
fair(draw, As, [I | Is], Ds, Es?) :- Es = [I? | Es1?], fair(draw, As?, Is?, Ds?, Es1).

procedure(Q, T) serve(Events(Q, T)?, Stream(Draw(T)), PersonOut(T),
    Routes(T)?, Deads(T)?, Integer?, Stream(Spawn(Q, T))).
serve([ask(_, Q) | Es], CInto?, DOut?, Rs, Xs, N,
      [spawn(N?, Q?, Gs?, Ds) | Ss?]) :-
    ground(N?) |
    merge(Ds?, CInto1?, CInto),
    N1 := N? + 1,
    serve(Es?, CInto1, DOut, [route(N?, Gs) | Rs?], Xs?, N1?, Ss).
serve([input(Id, P, R) | Es], CInto?, DOut?, Rs, Xs, N, Ss?) :-
    route_grant(input(Id?, P?, R?), Rs?, Rs1),
    serve(Es?, CInto, DOut, Rs1?, Xs?, N?, Ss).
serve([draw(Id, W, V) | Es], CInto?, [draw(Id?, W?, V?) | DOut?], Rs, Xs, N,
      Ss?) :-
    serve(Es?, CInto, DOut, Rs?, Xs?, N?, Ss).
serve([withdraw(Id) | Es], CInto?, [withdraw(Id?) | DOut?], Rs, Xs, N, Ss?) :-
    ground(Id?) |
    remove_route(Id?, Rs?, Rs1, Xs?, Xs1),
    serve(Es?, CInto, DOut, Rs1?, Xs1?, N?, Ss).
```

```
procedure(T) route_grant(Input(T)?, Routes(T)?, Routes(T)).
route_grant(input(Id, P, R), [route(Id1, [input(Id?, P?, R?) | Gs1?]) | Rs],
            [route(Id1?, Gs1) | Rs?]) :-
    Id? =?= Id1? | true.
route_grant(G, [route(Id1, Gs?) | Rs], [route(Id1?, Gs) | Rs1?]) :-
    otherwise |
    route_grant(G?, Rs?, Rs1).
route_grant(_, [], []).

procedure(T) remove_route(Integer?, Routes(T)?, Routes(T), Deads(T)?,
    Deads(T)).
remove_route(Id, [route(Id1, Gs?) | Rs], Rs?, Xs, [Gs | Xs?]) :-
    Id? =?= Id1? | true.
remove_route(Id, [route(Id1, Gs?) | Rs], [route(Id1?, Gs) | Rs1?], Xs,
             Xs1?) :-
    otherwise |
    remove_route(Id?, Rs?, Rs1, Xs?, Xs1).
remove_route(_, [], [], Xs, Xs?).
```

The dispatcher binds only variables of its own streams.

The construct process. The *construct process* of an ask `ask(`T`, `$t(Y)$`)` with identifier Id is a goal holding Y, the person's end of the interactive variable — its writer where T is in reader mode, and its reader where T is in writer mode — a stream of grants and a stream of draws. It holds a set of *questions*, each at its path from the root of the interactive variable's term: initially Y at the root where T is in reader mode, and none otherwise; as the program's output arrives on a reader it holds, each writer inside it at a position the person writes is added to its questions.

On a grant `input(`Id`, `Pos`, `R`)` for a question W it holds and has drawn, at path Pos, of type T', the construct process forms R by the forming procedure of T', which yields `formed(`u`)` or `refused`; u is a term of type T' with R at the positions the person writes and with fresh variables where Answer has them (Definition 5.7). On `formed(`u`)` it binds W to u, removes W from its questions and adds to them the writers of the new variables at the stream and list positions of u, and it holds the readers of the new variables at the positions the program writes. On `refused`, and on a grant for a path at which it holds no question, it binds nothing. It binds no variable but its questions, and those only on a grant.

A construct process has the bridge draw the widget of T once every reader in the output it holds is assigned, with that output and its questions, and again as output arrives inside the person's answers, each position there not yet written whole drawn as pending; it withdraws once it holds no question and no reader at a position whose output may hold one. The output, construct `0`, is drawn element by element as it arrives, its unwritten tail not drawn. It draws `draw(`Id`, `W`, `V`)`, W the widget of T, `named(`W'`, `D`)` where a declaration names it W', D the default widget of T, and V the output it has drawn, with `input(`Pos`)` at each question and `pending(`Pos`)` at each widget position not yet written whole, Pos the position's path, and with V under the functor t of its ask; the output thread's value is under the functor `output`. V is of the *view type* of T: T with the type S of each position the person writes replaced by S `; input(Path)`, and of each position the program writes inside the person's answers by S `; pending(Path)`, so no value drawn is typed by a wildcard.

So the values a program draws are of one type, the union of these, and the grants' raw inputs are of the types of its questions; the compilation forms the union of both, the parameter `T` of the program's person channel (Definition 6.8). The union is formed over the alternatives of these types, an alternative common to two of them taken once, and the compilation refuses a program two of

whose alternatives have one functor and arity and differ, since the alternatives of a union have distinct functors [58].

The mapping for Theorem 6.10 reads the source configuration off the compiled goals and the construct processes. An asking clause passes each argument of the goal it reduces to the goal it calls through a variable pair of its own, S_l and $S_l?$, and the mapping reads the pair as the argument: at an input position the writer mgu binds S_l to the goal's term u and the call receives $S_l?$, with $S_l? := u$ pending, and at an output position it binds the goal's writer W to $S_l?$ and the call receives S_l, with $W? := S_l?$ pending.

Definition B.2 (Compilation Mapping: Goals). Given a configuration of GLP with persons over P and $\lceil M \rceil$, the mapping σ yields the vmaGLP configuration over P and M with, at each agent, the goals G of this definition and the held variables H and the readers substitution σ_r of the next two: G is the multiset of the agent's goals, each read back to its source with its ask stream dropped: $q(S_1, \ldots, S_n, D)$ as the n-ary goal $q(S_1, \ldots, S_n)$, $q_1(S_1, \ldots, S_n, X?, D)$ and $q_1(S_1, \ldots, S_n, X, D)$ as the asked goal, with the identifier of the goal its asking clause reduced, and any other compiled goal as itself; the construct processes, the dispatcher and the `merge` goals the compilation adds contributing nothing; and in every goal an asking clause's pair for an argument read as that argument: at an input position $S_l?$ as the term u, until the assignment is communicated, and at an output position S_l as W and $S_l?$ as $W?$.

Definition B.3 (Compilation Mapping: Held Variables). H is the set of variables held for the person, each with its type: by an ask the dispatcher has not yet taken, and by a construct process that has not drawn its question, the person's end of the interactive variable; by a construct process that has drawn it, the writers of the positions the person writes, and the readers of the positions the program writes inside the person's answers whose output it has not drawn; and the output the construct processes have drawn, shown, each reader in it replaced by its value.

Definition B.4 (Compilation Mapping: Substitution). σ_r is the restriction of the readers substitution to the readers of *source variables* — the variables of the initial goal, of the compiled clauses of M as renamed, but for their ask streams, the interactive variables, and the variables of the terms construct processes bind questions with — less the assignments the asking clauses' reductions make, but for the assignment $W? := S_l?$ of an output position, which it reads through, absent while $S_l?$ is unassigned and $W? := u$ once $S_l? := u$ is; and with each assignment to a reader a construct process holds that a Communicate has delivered to it and the construct process has not yet drawn, Communicate removing the assignment it delivers (Definition 4.10).

Theorem 6.10 (Compilation). *The implementation of vGLP by GLP is correct, and complete for every program whose interactive types have complete widgets.*

Proof. Fix a vGLP program M and $P \subset \Pi$, and let σ be the mapping of Definition B.2. σ maps the initial configuration to the initial vmaGLP configuration: the goals read back to the initial goal of M, and there is no construct process.

Safety, per transition of GLP with persons over $\lceil M \rceil$. A grant, a reduction of the dispatcher or of a `merge` goal the compilation adds, a reduction of a construct process that neither binds a question nor draws output, and a Communicate on the person channel, on an ask stream, or between the dispatcher and the construct processes change nothing σ reads: stutters. A reduction of the asking clause of q replaces $q(S_1, \ldots, S_n, D)$ by the goal $q_1(S'_1, \ldots, S'_n, X?, D')$ or $q_1(S'_1, \ldots, S'_n, X, D')$, holding one end of a fresh interactive variable of the interactive type of q, which σ reads, through the pairs of its arguments and with the identifier of the goal reduced, as $q(S_1, \ldots, S_n, X?)$ or $q(S_1, \ldots, S_n, X)$, the assignments the reduction makes being outside σ_r, and writes on its ask stream an ask holding the other end: the Ask of the source goal. A Communicate of an assignment an asking clause's

reduction made replaces a pair by what σ already read it as: a stutter. A construct process binding a question X with the term t its person's input forms, a term of the type of X ground at the positions the person writes but for the stream and list positions, which hold readers of new writers it holds or, for a list the person ends, `[]`, with new writers at the positions the program writes, whose readers it holds: the Answer of p to X with t. A reduction of a compiled clause of M: its guard and body are those of the source clause, with calls of volitional procedures read back, ask streams dropped and the `merge` goals the compilation adds contributing nothing; its added head argument, the ask stream, matches the goal's writer whatever it is; and the earlier compiled clauses of the procedure, the source's in the same order, fail or suspend on the goal: a clause that reads an argument whose pair's assignment, made by the asking clause, is not yet communicated suspends on it and is followed by no clause that does not read it (Definition 5.2), so neither the reducing clause nor an earlier one reads such an argument, and each earlier clause fails or suspends on the source goal too; so it maps to the Reduce of the source goal with the source clause. A Communicate to a goal of the program, within the agent or from another, maps to the source Communicate of the same assignment, and a Cold-call is carried unchanged. A Communicate to a construct process is a stutter, σ_r keeping the assignment it delivers until the construct process draws it, and a construct process's draw of output it holds, every reader in it assigned, is the source Communicate of that output to the person (Definition 5.5), which removes from σ_r the assignments it draws.

Every safe run therefore maps to a safe run. *Correctness*: let ρ be a proper correct run of GLP with persons over $\lceil M \rceil$ and suppose a class in the domain of the equivalence is enabled in every configuration of a suffix of $\sigma(\rho)$ with no member taken. An Ask class of an n-ary goal corresponds to its goal $q(S_1, \ldots, S_n, D)$, whose asking clause succeeds throughout, the goal holding a writer at each output position (Definition 5.2); a Reduce class of an asked goal or of an ordinary goal corresponds to the compiled clause, which succeeds throughout by the correspondence of the safety case once the Communicates of the assignments its asking clause made, each enabled until taken, are taken; a Communicate class of an assignment to a reader the person holds corresponds to the construct process's draw of it, enabled once the Communicates to it of that assignment and of those read through it, each enabled until taken, are taken, and any other Communicate or Cold-call class to itself. Each is enabled in the corresponding suffix of ρ, whose liveness takes it, and its image is a member. Hence $\sigma(\rho)$ is correct.

Completeness, for a program whose interactive types have complete widgets. Let r be a correct run of the vmaGLP transition system over P and M. We build a proper correct run r' of GLP with persons over $\lceil M \rceil$ with $\sigma(r') = r$, taking the transitions of r in turn. For an Ask, reduce the goal's asking clause, let the `merge` goals carry its ask to the dispatcher, and let the dispatcher spawn its construct process, stutters but the first. For an Answer to X with t, the widget being complete, grant the input that forms t, let the dispatcher route it, and let the construct process bind X with t, stutters but the last. For a Reduce, take the compiled clause. For a Communicate to the person, take the Communicates to the construct process of its assignment and of those read through it, stutters, and the construct process's draw of the output. Any other Communicate, and a Cold-call, of r is taken as it is, followed by the Communicates on the channels of the dispatcher that it enables. The run r' is proper, its variables fresh, and live: a construct process awaiting input is suspended, the dispatcher and the `merge` goals are reduced in the stage that enables them, and every other class enabled from some configuration on is of a source class that r takes, r being correct. Hence σ is complete. □

Theorem 6.11 (Cascade). *For every effective local volition-guarded specification, computable introduction rule or none, and $P \subset \Pi$, $\sigma_1 \circ \sigma_2 \circ \sigma_3$ is a correct implementation of $\mathcal{F}(P)$ by GLP with persons over P and $\lceil\lceil\Theta\rceil\rceil$, under the fairness condition that a run ρ is proper, that no approval question*

stays open in $\sigma_3(\rho)$ and its calendar does not stop where the specification requires the day, and that $\sigma_2(\sigma_3(\rho))$ is connected, unobstructed and steady.

Proof. Fix an effective local volition-guarded specification and $P \subset \Pi$, and let ρ be a correct run of GLP with persons over P and $\lceil\lceil\Theta\rceil\rceil$ under the fairness condition. By Theorem 6.10, $\sigma_3(\rho)$ is a correct, proper run of the vmaGLP transition system over P and $\lceil\Theta\rceil$, in which no approval question stays open; by Theorem 5.19, $\sigma_2(\sigma_3(\rho))$, being connected, unobstructed and steady, is a correct run of $\mathcal{P}$ under its fairness condition; and by Theorem 3.24, $\sigma_1(\sigma_2(\sigma_3(\rho)))$ is a correct run of $\mathcal{F}(P)$. $\sigma_1 \circ \sigma_2 \circ \sigma_3$ takes the initial configuration to the initial configuration, as each of its parts does, so it is a correct implementation. □