Modular population protocols

Michael Raskin
LaBRI, University of Bordeaux, CNRS UMR 5800
mraskin@u-bordeaux.fr

Abstract. Population protocols are a model of distributed computation intended for the study of networks of independent computing agents with dynamic communication structure. Each agent has a finite number of states, and communication opportunities occur nondeterministically, allowing the agents involved to change their states based on each other's states. Population protocols are often studied in terms of reaching a consensus on whether the input configuration satisfied some predicate. A desirable property of a computation model is modularity, the ability to combine existing simpler computations in a straightforward way. In the present paper we present a more general notion of functionality implemented by a population protocol in terms of multisets of inputs and outputs. This notion allows to design multiphase protocols as combinations of independently defined phases. The additional generality also increases the range of behaviours that can be captured in applications (e.g. maintaining the role distribution in a fleet of servers).

We show that composition of protocols can be performed in a uniform mechanical way, and that the expressive power is essentially semilinear, similar to the predicate expressive power in the original population protocol setting.

Keywords: Population protocols · Protocol verification · Modularity.

1 Introduction

1.1 General context of population protocols

Population protocols have been introduced in [2,3] as a restricted yet useful subclass of general distributed protocols. Each agent in a population protocol has a fixed amount of local storage, and an execution consists of selecting pairs of agents and letting them update their states based on an interaction. The choice of pairs is assumed to be performed by an adversary subject to a fairness condition. The fairness condition ensures that the adversary must allow the protocol to progress.

Typically, population protocols are studied from the point of view of recognising some properties of an input configuration. In this context population protocols and their subclasses have been studied, for example, from the point of view of expressive power [5], verification complexity [9,19,18], time to convergence [4,15], necessary state count [8], etc.

The original target application of population protocols and related models is modelling networks of restricted sensors, starting from the original paper [2]...
on population protocols. Of course, in the modern applications the cheapest microcontrollers typically have thousands of bits of volatile memory permitting the use of simpler and faster algorithms for recognising properties of an input configuration. So on the one hand, the original motivation for the restrictions in the population protocol model seems to have less relevance. On the other hand, verifying distributed systems benefits from access to a variety of restricted models with a wide range of trade-offs between the expressive power and verification complexity, as most problems are undecidable in the unrestricted case. Complex, unrestricted, and impossible to verify distributed deployments lead to undesirable and hard to predict and sometimes even diagnose situations such as so called gray failures [22] and similar.

From the theoretical point of view, population protocols provide a model of distributed computing with some of verification problems still decidable[17] although non-elementary[11]. This property puts them near the edge of decidability of verification, which is interesting on its own.

1.2 Composition

Unfortunately, as the standard approach considers protocols that compute predicates, composing protocols directly is limited to boolean combinations. Whenever a multiphase protocol needs to be constructed, the interaction between phases needs to be described and proven in an ad hoc way. We find it desirable to have better modularity in protocol design.

Naturally, for the composition to be useful, we need the protocols to achieve more complex output distributions than consensus. Moreover, some constructions in the literature [10,20] have more or less sub-protocols executed in parallel with different agents participating in different sub-protocols based on the output of yet another «previous» sub-protocol. From the other point of view, desirable behaviours of distributed systems go beyond consensus about a single property of initial configuration. For example, one might want to maintain task allocation across a fleet of servers. Task allocation usually depends on the server models, and servers are sometimes taken out of service for maintenance. From this point of view, we also find it interesting to study a more general notion of expressive power.

1.3 Related work

Many approaches to composition of distributed protocols depend on a fixed communication structure [17], relatively rich local capabilities with infinite state space [24], or specific limits on how often each agents must be scheduled [12]. As there is no natural translation of such approaches to the model of population protocols, and the possibilities are less restricted in the richer models, we do not aim to classify such approaches.

Many composition methods rely on the notion of self-stabilisation introduced by Dijkstra [13], informally meaning that the agents will reach a correct (collective) configuration no matter what are their initial local states. Self-stabilising
protocols compose naturally. Indeed, whatever effects on the consumer protocol happen because of the changes in the output of the producer protocol, the end results can be treated as the initial configuration before the self-stabilisation in the consumer protocol starts. This has been long used to design composition formalisms [21,14]. Self-stabilisation has been applied to population protocols [6]. There, a general definition of behaviours as sets of series of output configurations is used, and a subclass behaviours with eventually unchanging outputs is defined. However, self-stabilisation is a very restrictive requirement for a population protocol where each agent may talk to each other (i.e. the communication graph is a clique). It is shown [6] that one cannot do leader election in a self-stabilising way, and the same proof applies to reaching the consensus whether there is currently a leader.

Some papers on self-stabilisation use lemmas about composing completely generic behaviours [14], but this is done in a low-level way.

The limitations of self-stabilisation for population protocols are not shared by the protocols with stabilising inputs [1]. The idea is that each agent gets an input that can change finitely many times. The authors mention that this approach is useful for composition, but only study reaching consensus on the value of some predicates. Moreover, only a conference version of that article exists with proofs deferred to a full version that has never been finished, so for some statements it is unclear whether they should be considered proven.

One of the works on majority [20] explicitly describes the most complicated protocol via a sequential composition of protocols with eventually-stabilising inputs, while solving a problem that is impossible for general self-stabilising protocols. However, the proofs are specific to the protocols used and no general statement (nor a general definition) is given. In a sense, our approach is a broad and reusable generalisation of the approach there.

1.4 Contribution

In the present paper we introduce an alternative general notion of a protocol implementing an input-output specification. We consider each agent to have an input, and as usual there is an output function. An input-output specification is a relation between multisets of inputs and multisets of outputs, plus a compatibility relation between individual inputs and outputs. A protocol implements the specification, if in each fair execution from an input configuration there is a step after which the specification is always satisfied, both in terms of the multisets of inputs and outputs, and in terms of each agent’s output being compatible with the input. The standard notion of computing predicates by stable consensus corresponds to a natural class of specifications where output is a single bit, only consensus output configurations are acceptable, and for each input configuration only one of the two consensus options is acceptable.

Additionally, we define computations with shutdown requests. We allow the scheduler to add new agents, or to tell existing agents to leave the computation. Each leaving agent can take some time to hand over the information, but must
eventually terminate (in a fair execution). Once there is no turnover, an input-output specification must be eventually stably satisfied.

We show a natural completely mechanical way of combining two population protocols with shutdown requests implementing two relations to obtain a protocol with shutdown requests implementing the composition of these relations.

In terms of handling the inputs, we add the possibility of shut down request to the approach of stabilising inputs. In terms of the specifications implemented, we study what multisets of outputs can get obtained from what multisets of inputs, while previous work for stabilising inputs only considered computing predicates.

The rest of the present paper is organised as follows. We start with presenting the standard definitions for population protocols, then in the next section we define our extension to the model to speak about specifications beyond predicate evaluation and provide some examples. In the section after that we define and construct sequential composition of protocols. We then proceed to characterise the expressive power of population protocols under our definition. The paper ends with a small conclusion with some future directions outlined.

For better alignment with the conference version, the proofs missing in the main text have been placed in the appendix.

2 Basic definitions

In this section we recall the standard definitions and facts related to population protocols. We use the definition where agents have identities for the purposes of analysis but cannot distinguish each other. This will also be relevant in the generalisations. First we define the population protocols. We start by describing what information we need to specify a protocol.

**Definition 1.** A population protocol is defined by a finite set of states $Q$ and a step relation $\text{Step} \subseteq Q^2 \times Q^2$. When there is no ambiguity about the protocol, we abbreviate $((q_1, q_2), (q_1', q_2')) \in \text{Step}$ as $(q_1, q_2) \mapsto (q_1', q_2')$ and call the quadruple $(q_1, q_2) \mapsto (q_1', q_2')$ a transition.

We use the following notation to work with function on agents.

**Definition 2.** For a function $f$ let $\text{Dom}(f)$ denote the domain of the function. For a function $f$ and $x \not\in \text{Dom}(f)$ let $f \cup \{x \mapsto y\}$ denote the function $g$ defined on $\text{Dom}(f) \cup \{x\}$ such that $g |_{\text{Dom}(f)} = f$ and $g(x) = y$. For $u \in \text{Dom}(f)$ let $f[u \mapsto v]$ denote the function $h$ defined on $\text{Dom}(f)$ such that $h |_{\text{Dom}(f) \setminus \{u\}} = f |_{\text{Dom}(f) \setminus \{u\}}$ and $h(u) = v$. For symmetry, if $w = f(u)$ let $f \setminus \{u \mapsto w\}$ denote restriction $f |_{\text{Dom}(f) \setminus \{u\}}$.

Use of this notation implies an assertion of correctness, i.e. $x \not\in \text{Dom}(f)$, $u \in \text{Dom}(f)$, and $w = f(u)$.

**Definition 3.** A configuration of a population protocol is a set $A$ of agent identities, and a state function $C : A \to Q$, assigning states to agents. We identify the configuration with the state function as $A = \text{Dom}(C)$. The size of a configuration $C$ is the number of agents. An execution of a population protocol is a finite
or infinite sequence \((C_0, C_1, \ldots)\) of configurations such that the set of agents is the same for each \(C_j\), and for each \(j\) between 1 and the execution length we have some two agents \(a_1, a_2\) that interact according to the rules, i.e. \(C_j(a_1) = q_1, C_j(a_2) = q_2, C_{j+1} = C_j[a_1 \mapsto q'_1][a_2 \mapsto q'_2]\) where \(((q_1, q_2), (q'_1, q'_2)) \in \text{Step}\). In other words, we let two agents with states \(q_1\) and \(q_2\) interact in some way permitted by the \text{Step} relation.

**Example 1.** Consider the set of states \(\{q_0, q_1, q_2, q_3\}\). The step relation is described by

\[
\begin{align*}
(q_1, q_1) &\mapsto (q_0, q_2), \\
(q_2, q_1) &\mapsto (q_0, q_3), \\
(q_2, q_2) &\mapsto (q_1, q_3), \\
(q_0, q_3) &\mapsto (q_3, q_3), \\
(q_1, q_3) &\mapsto (q_3, q_3), \\
(q_2, q_3) &\mapsto (q_3, q_3).
\end{align*}
\]

Using agent identities 1, 2, and 3, and denoting a function from 1, 2, 3 as a tuple, we can consider the following execution: \((q_1, q_1, q_1), (q_2, q_1, q_0), (q_0, q_3, q_0), (q_3, q_3, q_0), (q_3, q_3, q_3)\). Here we use the first two steps, then twice the fourth step.

**Remark 1.** Note that all the configurations in an execution have the same size.

We often consider executions with the steps chosen by an adversary. However, we need to restrict adversary to ensure that some useful computation remains possible. To prevent the adversary from e.g. only letting one pair of agents to interact, we require the executions to be fair. The fairness condition can also be described by comparison with random choice of steps to perform: fairness is a combinatorial way to exclude a zero-probability set of bad executions.

**Definition 4.** Consider a population protocol \((Q, \text{Step})\).

A configuration \(C'\) is reachable from configuration \(C\) iff there is a finite execution with the initial configuration \(C\) and the final configuration \(C'\).

A finite execution is **fair** if it cannot be extended, i.e. it is not a prefix of any longer execution.

An infinite execution \(C_0, C_1, \ldots\) is **fair** if for every configuration \(C'\) either \(C'\) is not reachable from some \(C_j\) (and all the following configurations), or \(C'\) occurs among \(C_j\) infinitely many times.

**Example 2.** The finite executions in the example are fair.

**Remark 2.** As long as the number of reachable configurations is finite, fairness implies that every set of configurations either becomes unreachable, or gets reached infinitely many times.

The most popular notion of expressive power for population protocols is computing predicates, defined in the following way.
Definition 5. Consider a population protocol \((Q, \text{Step})\) with additionally defined non-empty set of input states \(I_s \subset Q\), output alphabet \(O\) and output function \(o : Q \rightarrow O\).

A state is inhabited in the configuration if there is at least one agent with this state. Considering a configuration as a function from agents to states, we can say that a state \(q\) is inhabited if \(q \in \text{Im}(C)\).

A configuration \(C\) is an input configuration if all the inhabited states are among the input states, \(\text{Im}(C) \subset I_s\).

A configuration \(C\) is a \(b\)-consensus for some \(b \in O\) if the output function yields \(b\) for all the inhabited states, \(\text{Im}(o \circ C) = \{b\}\). A configuration is a stable \(b\)-consensus if it is a \(b\)-consensus together with all the configurations reachable from it. A configuration is called just a consensus or a stable consensus if it is a \(b\)-consensus (respectively stable \(b\)-consensus) for some \(b\).

A protocol computes a function \(\varphi : \mathbb{N}^I_s \rightarrow O\) iff for each input configuration \(C\), every fair execution with initial configuration \(C\) contains a stable \(\varphi(C)\)-consensus. We usually use the protocols computing predicates, which corresponds to \(O = \{\text{true}, \text{false}\}\).

Example 3. If we define the set of input states \(I_s = \{q_1\}\), output set \(O = \{\text{true}, \text{false}\}\) and the output function \(o(q) = (q = q_3)\), the protocol in the example \(\square\) computes the predicate \(\varphi(C) = (C(q_1) \geq 3)\).

3 Model extensions

In this section we describe our extensions.

Informally speaking, each agent knows its input (either a value or a shutdown request) as a part of the state, but the agent cannot change it. For convenience, all input state have the same «empty» internal component of the state. An agent whose input becomes a shutdown request and doesn’t change should eventually change the internal component of the state to empty; agents with a shutdown request and empty internal component of the state do not participate in any interactions (and can be removed from the configuration). The scheduler can change the inputs and add agents in the input states, but only finitely many times.

A specification consists of two relations: the main relation on multisets of inputs and multisets of outputs; and a secondary relation on the input language and the output language. The first relation describes the global-level requirements, and the second describes compatibility of inputs and outputs (e.g. a server needs to have enough storage if it is a backup server). For every multiset of inputs there must be at least one acceptable multiset of input-output pairs (i.e. the task is never completely impossible). A protocol implements a specification \((\varphi, \psi)\) if any finite sequence of scheduler actions from an input configuration followed by a fair execution without input changes or agent addition/removal eventually has a stable output for each agent and the multisets of inputs and outputs satisfy the relation \(\varphi\), while each agent’s input and output satisfy \(\psi\).
So we could say something like «the input being amount of storage for a server, we consider as many webservers as there are servers without sufficient storage for a database server, as many database servers as webservers, and the remaining servers as backup servers — and only servers with enough storage can be database servers or backup servers». Then after decommissioning some servers, deploying some new ones, and reconfiguring a few of the remaining ones, the fleet settles into a role assignment compliant with the policy. Formally, we also need some policy what to do when no servers have sufficient storage, accepting something that would be a misconfiguration otherwise.

**Definition 6.** We use \( \bot \) to denote lack of value (in the sense of a shutdown request or in the sense of the initial empty internal state).

A protocol is input-saving if there is a set of input values \( I \) such that the set of states \( Q \) is the Cartesian product \( (I \cup \{ \bot \}) \times (Q_{int} \cup \{ \bot \}) \), the set of input states \( I_s \) is equal to \( I \times \{ \bot \} \), the output function on \((\bot, \bot)\) is equal to \( \bot \), and the following conditions on the step relation hold:

1. the first component of the state (the input) doesn’t change;
2. if at least one agent has both components of the state equal to \( \bot \), the states don’t change at all (the agents don’t interact while shut down).

We call the first component of the state the **input**, and the second component of the state the **internal memory**.

A reconfiguration of an input-saving population protocol is one of the following changes to the configuration:

1. removing an agent in the shutdown state \((\bot, \bot)\);
2. adding an agent in the shutdown state \((\bot, \bot)\);
3. changing an agent’s input, i.e. replacing a state \((i, q)\) with a state \((i', q)\) that differs only in the first component.

An execution with reconfiguration of an input-saving protocol is a finite or infinite sequence of steps and reconfigurations. An execution with reconfiguration is fair if it has a finite number of reconfigurations, and its suffix after the last reconfiguration is a fair execution.

An input-saving population protocol respects shutdown requests, if in each fair execution with reconfiguration after some time either all agents have the input \( \bot \) or all the agents with the input \( \bot \) are in the shutdown state \((\bot, \bot)\).

An input-saving population protocol implements a specification \((\varphi, \psi)\) if the following conditions all hold:

1. the protocol respects shutdown requests;
2. in each fair execution with reconfiguration each agent’s output only changes a finite number of times;
3. \( \varphi \) is satisfied by the multiset of inputs and the multiset of outputs after stabilisation of outputs;
4. for each agent, \( \psi \) is satisfied by the agent’s input and output after stabilisation.
Remark 3. If we want an execution with reconfiguration to start with a non-empty configuration, we can instead use reconfiguration to add the necessary agents in the very beginning of the execution.

Remark 4. After the last reconfiguration in a fair execution with reconfigurations the number of reachable configurations becomes finite, so by fairness any set of configurations has to become unreachable or be reached infinitely many times.

Remark 5. When verifying a specific protocol, $\psi$ is usually trivially ensured by the output function.

Example 4. Consider the following input-saving protocol.

The input language has two elements, «Yes» and «Maybe». The output language has three elements, «Yes» and «No», and $\perp$. The internal memory language has four elements «Mes», «Yes», «No», and $\perp$.

The output function returns $\perp$ when the input is $\perp$, «Yes» when the input is «Yes» or the internal memory is «Yes», and «No» otherwise.

In a pattern form:

\[
(\perp, *) \rightarrow \perp \\
(Yes, *) \rightarrow Yes \\
(Maybe, Yes) \rightarrow Yes \\
(Maybe, Me/No/\perp) \rightarrow No
\]

The step function works as follows (right-hand-side $*$ keeps the value from the left-hand-side; same interactions are also possible in the reverse order).

\[
(\perp, *), (Yes, *) \rightarrow (\perp, \perp), (Yes, Me) \\
(\perp, Me), (Maybe, *) \rightarrow (\perp, \perp), (Maybe, Me) \\
(\perp, Yes/No), (*, *) \rightarrow (\perp, \perp), (*, *) \\
(Yes, *), (Yes, *) \rightarrow (Yes, Me), (Yes, Me) \\
(Yes, *), (Maybe, *) \rightarrow (Yes, Me), (Maybe, Yes) \\
(Maybe, Me), (Maybe, *) \rightarrow (Maybe, Me), (Maybe, No) \\
(Maybe, Me), (Maybe, *) \rightarrow (Maybe, Me), (Maybe, *)
\]

Here $\mathcal{M}e$ means any value allowed in this position except $Me$.

Informally, internal memory «Me» is set by an own «Yes» input and gets replaced with «Yes» by observing another «Yes» input while not having a «Yes» input; it is also «handed over» before the shutdown; internal memory «Yes» is set by observing a «Yes» input of another agent and removed by observing a «Me» internal memory of an agent without «Yes» input.

We observe the following.

1. The protocol respects shutdown requests.
2. The protocol implements the specification «all inputs are “Maybe” and all outputs are “No”, or there is a “Yes” input and all outputs are “Yes”»
The basic idea is that «Yes» in the memory appears when «Me» appears somewhere else, eventually all «Me» are for the same input, and «Me» with a given input spread either «Yes» or «No» output.

Leader election can be implemented in a similar way, and we provide the details in the appendix. We only promise that eventually there will be a leader if after reconfigurations there are at least two non-shutdown agents.

Example 5. Consider the following protocol, parametrised by a positive integer \( m \). The input language is \( \{-m, -m + 1, \ldots, m - 1, m\} \). The output language is the same as the input language. The internal memory language is \( \{\{-m, -m + 1, \ldots, m - 1, m\} \times \{-2m, -2m + 1, \ldots, 2m - 1, 2m\}\} \). We call the two sub-components of the internal memory «previous input» and «current balance».

We will call \( k \)-clamping of a number \( x \) the number \( x \) itself if it is at most \( k \) by absolute value (i.e. \( |x| \leq k \)) and \( k \) or \( -k \) according to the sign of \( x \) otherwise (i.e. \( k \cdot \text{sign}(x) \)). We write \( \text{clamp}_k x = \min(k, \max(-k, x)) \).

The output function is the \( m \)-clamping of the current balance.

The step function is as follows.

First, for each of the agents we consider the current input \( i \), the previous input \( p \) and the current balance \( b \). We try to add the difference between the previous and the current input to the current balance and to the previous input, but make sure the values stay in the permitted range. More precisely, we consider \( b' = \text{clamp}_{2m}(b+i-p) \), \( p' = p+b'-b \), and update the current balance to become \( b' \) and the previous input to become \( p' \). (As \( p' \) is between \( p \) and \( i \), it cannot cause range problems).

Then we consider the two updated current balances, \( b'_1 \leq b'_2 \) (possibly swapping the agents to ensure that the first balance is larger). We replace them with \( b'_1 = \text{clamp}_{2m}(b'_1 + b'_2) \) and \( b'_2 = b'_1 + b'_2 - b'_1 \).

Then if exactly one of the agents has the current input \( \bot \) and the other agent has the current input \( 0 \), we swap the current balances. Afterwards each agent with the current input \( \bot \), the previous input \( 0 \), and the current balance \( 0 \) gets the internal memory set to \( \bot \) to shut it down.

Observe that this protocol implements the specification that either there is at most one agent, or the following holds.

- Either there are no agents with a negative output or there are not agents with a positive output.
- The largest output by absolute value is the (clamped) sum of all the current inputs \( \text{clamp}_m \sum_j i_j \).

The basic idea is that the current balances cancel out when possible and accumulate otherwise, while the unused difference between current input and previous input in a stable situation can only be in the same direction as the sign of the global sum of inputs.

Remark 6. If we replace clamping with taking a remainder modulo \( m \), and all the sums in the specification and the proof with sums modulo \( m \), we obtain a very similar protocol for computing the sum of the inputs modulo \( m \) as the only non-zero agent output.
3.1 Parallel composition

In this section we briefly mention the simpler composition methods for input-saving population protocols.

If we have two protocols with the same input language, we can take the Cartesian product of their output languages and internal memory languages, and execute the two protocols in parallel on the same population. For the internal memory we identify $(\bot, \bot)$ with $\bot$. Clearly, the product protocol will respect the shutdown requests if the original ones did. If the protocols implemented specifications $(\varphi_1, \psi_1)$ and $(\varphi_2, \psi_2)$, the product protocol will implement the specification that accepts a multiset of inputs and a multiset of outputs if replacing outputs with their first components makes $\varphi_1$ accept the pair of multisets and $\psi_1$ accept each agent’s input-output pair, while replacing outputs with their second components satisfies $\varphi_2$ globally and $\psi_2$ locally.

It is possible that the protocols we wanted to combine had different inputs, and we want to combine the outputs in some way other than building a pair. For the output side, we can replace the output function with a composition of the old output function and an output-translating function. For the input side, given an input-translating function, we need to define the new transitions as the old transitions, where inputs are the images of the actual inputs. The specification will be composed with the translation functions in a natural way.

Example 6. We have a protocol (example 3) to check whether any agent has input «Yes». If we want to verify that with the input language \{0, 1, 2\} some agents have 0 and some agents have 1, we can ask each agent to participate in two independent copies of the «Yes»-checking protocol. One copy would interpret 0 as «Yes» (other inputs as «Maybe»), and the other copy would interpret 1 as «Yes». The output of naive Cartesian product protocol would be two «Yes»/«No» values, but we use their conjunction as the top-level output.

4 Sequential composition

In this section we show how to perform sequential composition, using the outputs of one protocol (producer) as input of the following one (consumer). This is similar to the general notion of a computational pipeline.

Definition 7. Consider finite sets $I$, $M$, $O$ (informally: global input, language in the middle, and global output, the protocols working from $I$ to $M$ and from $M$ to $O$). Consider also finite sets $Q_1$ and $Q_2$ (to be used as internal memory languages for the two protocols). Consider two input-saving population protocols $(I \times Q_1, \text{Step}_1, o_1)$ and $(M \times Q_2, \text{Step}_2, o_2)$ (the producer and the consumer protocols, correspondingly).

Then the sequential composition of these two protocols is the following input-saving population protocol. The input language is $I$. The output language is $O$. The internal memory language is $(Q_1 \times M \times Q_2)$; we identify $\bot$ with $(\bot, \bot, \bot)$ in the internal memory. The output function is $o(q_1, m, q_2) = o_2(q_2)$. 
The step relation is defined by the following procedure. The procedure is deterministic if the two original protocols had functions as step relations. The procedure takes two states (saved input included), \((i_1, q_1^1, m_1, q_1^2)\) and \((i_2, q_2^1, m_2, q_2^2)\).

- We compute the new internal memory for the first (producer) protocol based on the global inputs and the old internal memory states. In other words, we non-deterministically pick an arbitrary fitting transition \(((i_1, q_1^1), (i_2, q_2^1)) \in \text{Step}_1\) if there is any, and otherwise we do nothing and take \((q_1^1', q_2^1') = (q_1^1, q_2^1)\). Note that the inputs cannot change during a step.
- We update the intermediate output-input as the output of the first (producer) protocol. In other words, we set \(m_1' = o_1(i_1, q_1^1')\) and \(m_2' = o_1(i_2, q_2^1')\).
- We compute the new internal memory for the second (consumer) protocol, based on the intermediate output-input as the input. In other words, we non-deterministically pick an arbitrary fitting transition \(((m_1', q_1^2), (m_2', q_2^2)) \in \text{Step}_2\) if there is any, and otherwise we do nothing and take \((q_1^2', q_2^2') = (q_1^2, q_2^2)\). Note that the input cannot change but we use its updated value.
- The new states are \((i_1, q_1^1', m_1', q_1^2')\) and \((i_2, q_2^1', m_2', q_2^2')\). (The input doesn’t change, as required for an input-saving protocol).

We also want to combine specifications.

**Definition 8.** The composition of relations \(\varphi_1 \subseteq I \times M\) and \(\varphi_2 \subseteq M \times O\), is the relation \(\varphi \subseteq I \times O = \{(I, O) \mid \exists M: \varphi_1(I, M) \land \varphi_2(M, O)\}\). This is compatible with composition of relations as (multi-valued) functions.

The composition of specifications composes the two components correspondingly.

**Theorem 1.** Sequential composition of input-saving population protocols implementing two specifications implements the composition of the specifications as relations.

The basic idea is that things that are allowed to happen during a fair execution with reconfigurations of the composition protocol correspond to things that are allowed to happen during each of the two protocols executed independently; and once things stabilise, we can recover the intermediate values from the intermediate output-input components of the internal memory.

### 5 Expressive power

In this section we study which specifications can be implemented by input-saving population protocols. We show that these specifications are the semilinear (Presburger-definable) ones.

We use both equivalent definitions of these sets: they are definable as finite unions of linear sets, where a linear set is the set of points obtained from a base vector by repeated addition of the vectors from a fixed finite set of periods;
and they are definable by boolean combination of equalities, inequalities, and fixed-modulo modular equalities of integer linear combinations of coordinates.

Strictly speaking, a protocol implementing a specification \((\varphi, \psi)\) implements a disjunction \((\varphi \lor \varphi', \psi)\) for any relation \(\varphi'\). We will call the former specification stronger than the latter. We will prove that for each specification implemented by a protocol there is a stronger semilinear specification implemented by some other protocol.

**Definition 9.** A specification \((\varphi, \psi)\) is **semilinear** if \(\varphi\) is semilinear when considered as a predicate on a tuple of multiplicities of possible input and output values.

**Theorem 2.** Every semilinear specification which can in principle be satisfied for each multiset of inputs is implemented by some input-saving population protocol.

*Here a specification \((\varphi, \psi)\) can be satisfied for a multiset \(I\) of inputs if there exists a multiset \(P\) of pairs satisfying \(\psi\), such that the multiset of the first components of all pairs in \(P\) is \(I\), the multiset of the second components of pairs in \(P\) is some multiset \(O\), and the multisets \(I\) and \(O\) of the first and the second components satisfy \(\varphi\).*

The basic (and very inefficient) idea is that we have already seen protocols that are enough to compute semilinear predicates, so we can guess the output assignment and verify that it is suitable.

**Theorem 3.** Every specification implemented by an input-saving population protocol is weaker (accepts more multisets of input-output pairs) than some implementable semilinear specification.

The basic idea is to consider two notions of reachability: with and without reconfiguration. Due to respect of shutdown request, execution with reconfiguration can always return to a configuration of size 1. Consider a bottom strongly connected component (BSCC) for reachability between size-1 configurations. Configurations of arbitrary size reachable from this BSCC via executions with reconfiguration have mutual reachability with this BSCC and thus form a semilinear set \([23]\). BSCCs of executions without reconfiguration form a semilinear set \([16]\); those of them that are reachable (with reconfiguration) from the chosen size-1 reconfiguration BSCC form a semilinear set as an intersection of two semilinear sets. It turns out that this set has to be allowed by the specification and also have at least one configuration for each multiset of inputs.

**6 Conclusion and future directions**

We have presented an extension of the population protocols model with a better support for modular design and synthesis of the protocols, as well as for a wider range of applications.
We have also established that the specifications that can be implemented this way are essentially the semilinear ones.

However, the constructions used are quite inefficient; it might be of interest to adapt to our model (and possibly extend beyond computing predicates) the existing fast and succinct protocols [10]. However, even a single-exponential-state-count polynomial-convergence-time construction for an arbitrary semilinear specification would be of interest.

Acknowledgments. I would like to thank Javier Esparza, Roland Guttenberg, Jérôme Leroux and Chana Weil-Kennedy for useful discussions. I am also grateful to the anonymous reviewers of this and previous versions for their valuable feedback and advice on presentation.

The project has been supported by a European Research Council (ERC) project under the European Union’s Horizon 2020 research and innovation programme grant agreement No 787367 (PaVeS). This work has been supported by France National Research Agency (ANR) grant ANR-23-CE48-0005 (PaVeDyS).

Disclosure of Interests. The author(s) have no competing interests to declare that are relevant to the content of this article.

References

1. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In: International Conference on Distributed Computing in Sensor Systems (2005),
   [https://api.semanticscholar.org/CorpusID:16310485](https://api.semanticscholar.org/CorpusID:16310485)
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Chaudhuri, S., Kutten, S. (eds.) PODC. pp. 290–299. ACM (2004).
   [https://doi.org/10.1145/1011767.1011810](https://doi.org/10.1145/1011767.1011810)
3. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing 18(4), 235–253 (2006)
4. Angluin, D., Aspnes, J., Eisenstat, D.: Fast computation by population protocols with a leader. In: Dolev, S. (ed.) IN DISTRIBUTED COMPUTING: 20TH INTERNATIONAL SYMPOSIUM, DISC 2006. Lecture Notes in Computer Science, vol. 4167, pp. 61–75. Springer (2006)
5. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007),
   [https://arxiv.org/abs/cs/0608084](https://arxiv.org/abs/cs/0608084)
6. Angluin, D., Aspnes, J., Fischer, M.J., Jiang, H.: Self-stabilizing population protocols. In: TAAS (2005), [https://api.semanticscholar.org/CorpusID:63153](https://api.semanticscholar.org/CorpusID:63153)
7. Austin, M.A., Johnson, J.: Compositional approach to distributed system behavior modeling and formal validation of infrastructure operations with finite state automata: Application to viewpoint-driven verification of functionality in waterways. Syst. 6, 2 (2018), [https://api.semanticscholar.org/CorpusID:4786185](https://api.semanticscholar.org/CorpusID:4786185)
8. Blondin, M., Esparza, J., Genest, B., Helfrich, M., Jaax, S.: Succinct population protocols for presburger arithmetic. in submission (2019),
   [http://arxiv.org/abs/1910.04600](http://arxiv.org/abs/1910.04600)
9. Blondin, M., Esparza, J., Jaax, S., Meyer, P.J.: Towards efficient verification of population protocols. In: Schiller, E.M., Schwarzmann, A.A. (eds.) PODC. pp. 423–430. ACM (2017)

10. Czerner, P., Guttenberg, R., Helfrich, M., Esparza, J.: Fast and succinct population protocols for presburger arithmetic. J. Comput. Syst. Sci. 140, 103481 (2024). https://doi.org/10.1016/j.jcss.2023.103481

11. Czerwiński, W., Lasota, S., Lazić, R., Leroux, J., Mazowiecki, F.: The reachability problem for Petri nets is not elementary. In: Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC). pp. 24–33 (2019). https://doi.org/10.1145/3313276.3316369

12. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. In: Safety-critical Systems Symposium (2007), https://api.semanticscholar.org/CorpusID:2179550

13. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17, 643–644 (1974), https://api.semanticscholar.org/CorpusID:11101426

14. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Comput. 7(1), 3–16 (1993). https://doi.org/10.1007/BF02278851

15. Doty, D., Soloveichik, D.: Stable leader election in population protocols requires linear time. In: Moses, Y. (ed.) DISC. Lecture Notes in Computer Science, vol. 9363, pp. 602–616. Springer (2015), https://arxiv.org/abs/1502.04246

16. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Verification of Population Protocols. In: Aceto, L., de Frutos Escrig, D. (eds.) 26th International Conference on Concurrency Theory (CONCUR 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 42, pp. 470–482. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015). https://doi.org/10.4230/LIPICS.CFC.2015.470

17. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Model Checking Population Protocols. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 65, pp. 27.1–27.14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPICS.FSTTCS.2016.27

18. Esparza, J., Jaax, S., Raskin, M.A., Weil-Kennedy, C.: The complexity of verifying population protocols. Distributed Comput. 34(2), 133–177 (2021). https://doi.org/10.1007/s00446-021-00390-x

19. Esparza, J., Raskin, M., Weil-Kennedy, C.: Parameterized analysis of immediate observation petri nets (2019), http://arxiv.org/abs/1902.03025

20. Gasieniec, L., Hamilton, D.D., Martin, R., Spirakis, P.G., Stachowiak, G.: Deterministic population protocols for exact majority and plurality. In: Fatourou, P., Jiménez, E., Pedone, F. (eds.) 20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain. LIPIcs, vol. 70, pp. 14:1–14:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPICS.OPODIS.2016.14
21. Gouda, M., Herman, T.: Adaptive programming. IEEE Transactions on Software Engineering 17(9), 911–921 (1991). https://doi.org/10.1109/32.92911
22. Huang, P., Guo, C., Zhou, L., Lorch, J.R., Dang, Y., Chintalapati, M., Yao, R.: Gray failure. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS '17. ACM Press (2017). https://doi.org/10.1145/3102980.3103005
23. Leroux, J.: Vector addition system reversible reachability problem. Log. Methods Comput. Sci. 9(1) (2013). https://doi.org/10.2168/LMCS-9(1:5)2013
24. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Transactions on Modeling and Computer Simulation (TOMACS) 28, 1 – 28 (2017). https://api.semanticscholar.org/CorpusID:4302945
A Proofs for Section 3 (Model extensions)

Proof (of example 4).

1. Any interaction of an agent with the input $\bot$ with an agent with a non-shutdown input leads to the agent with the input $\bot$ shutting down. Any fair execution will eventually need to include such an interaction given that there is an agent with $\bot$ input and an agent with a non-$\bot$ input.

2. First, observe that as long as there is a $(\text{«Maybe»}, \text{«Yes»})$ agent there is an agent with $\text{«Me»}$ in the internal memory. Indeed, any creation of a state $(\text{«Maybe»}, \text{«Yes»})$ also ensures existence of $(\text{«Yes»}, \text{«Me»})$; while reducing the number of $\text{«Me»}$ is only possible by replacing two of them with one. As the protocol cannot create a $(\text{«Yes»}, \text{«Yes»})$ with a step, reconfigurations also preserve this condition.

Second, observe that as long as there is a $\text{«Yes»}$ input after the last reconfiguration, interacting with this agent reduces the number of $(\text{«Maybe»}, \text{«Me»})$. Therefore eventually there will be no $\text{«Yes»}$ inputs or no $(\text{«Maybe»}, \text{«Me»})$.

Third, $\text{«Yes»}$ inputs convert $\text{«Maybe»}$ inputs to $(\text{«Maybe»}, \text{«Yes»})$, while $(\text{«Maybe»}, \text{«Me»})$ converts $\text{«Maybe»}$ inputs with non-$\text{«Me»}$ internal memory to $(\text{«Maybe»}, \text{«No»})$. Therefore either there is never a $\text{«Me»}$ internal memory, or eventually everyone has output $\text{«Yes»}$ forever, or eventually everyone has output $\text{«No»}$ forever. In particular, the outputs of all agents stabilise.

Example 7. Consider the input-saving protocol with input language of one symbol $\top$ (the shutdown request $\bot$ being added to the input language), output language $\{\text{Leader}, \text{Follower}\}$, and the same internal memory language (plus $\bot$).

The transitions are:

\[
(\bot, \ast), (\top, \text{Leader}) \to (\bot, \bot), (\top, \text{Leader}) \tag{12}
\]

\[
(\bot, \text{Leader}), (\top, \ast) \to (\bot, \bot), (\top, \text{Leader}) \tag{13}
\]

\[
(\bot, \text{Follower}), (\top, \text{Leader}) \to (\bot, \bot), (\top, \text{Follower}) \tag{14}
\]

\[
(\bot, \text{Leader}), (\bot, \text{Leader}) \to (\bot, \bot), (\bot, \bot) \tag{15}
\]

\[
(\bot, \text{Leader}), (\bot, \ast) \to (\bot, \text{Leader}), (\bot, \bot) \tag{16}
\]

\[
(\top, \text{Leader}), (\top, \ast) \to (\top, \text{Leader}), (\top, \text{Follower}) \tag{17}
\]

\[
(\top, \text{Follower}), (\top, \text{Leader}) \to (\top, \text{Follower}), (\top, \text{Follower}) \tag{18}
\]

\[
(\top, \bot), (\top, \bot) \to (\top, \text{Leader}), (\top, \text{Follower}) \tag{19}
\]

Informally, Follower knows there is a Leader, one of two Leaders becomes a Follower on collision, and when there is no information one of the agents opportunistically declares itself a leader. During the shutdown, the agent makes sure to pass the Leader status to another agent.

Proof (of example 7). An agent with a shutdown request as the input will shutdown after any interaction with an agent without a shutdown request. A Follower can only exist if there is a Leader, the number of Leaders cannot go to zero from
positive, but the number of Leaders can only decrease once there are no \((\top, \bot)\) agents and no reconfigurations happen. Thus the protocol implements leader election.

**Proof (of example 5).** First of all, the sum of the current balances over all the agents is always equal to the sum of the saved previous inputs, as reconfigurations cannot change either value and steps can only change them by the same amount.

Observe that once the reconfigurations cease, the sum of absolute values of differences between the current input and the stored previous input can only decrease. Thus when analysing a fair execution with reconfiguration we can assume that the sum of absolute values of these differences has achieved its minimum at some point and consider the execution afterwards.

Note that the difference between the current and the previous inputs is always of the same sign as the current balance from this point on. Indeed, otherwise the agent would decrease the difference at the next interaction.

Now consider the sum of the absolute values of the current balances. It can only grow due to updating the balance and the previous input, decreasing the sum of absolute values of differences between the current and the previous inputs. Once the latter sum has stabilised, the former sum can only decrease, so we can consider the execution once it has achieved its minimum. As an interaction between the agents with the balances of the opposite signs will decrease that sum, the agents’ current balances, and thus outputs, have to be all non-negative or all non-positive from this point on.

Now consider the number of agents that have non-zero non-maximum current balance. As by now the current balances can only shift around between the agents, the number of such agents can only decrease. Moreover, at this point interaction of two such agents leaves only one such agent (merging the balances either creates the maximum value or a zero value). Thus eventually there will be at most one such agent and we can consider the execution starting from this point.

Observe that the number non-shut-down agents with the current input \(\bot\) can only decrease. Note that the sum of all current balances is always the sum of the stored previous inputs, and by the current moment in the execution it has the absolute value at most the sum of the absolute values of current inputs (we have already shown that the difference can only go in one direction by this point). As the current input \(\bot\) counts as zero, and each agent can hold more as the current balance than as the previous input, as long as there is a non-shut-down agent with a shutdown request, there is an agent with the current balance \(0\). Interaction of these two agents lets the agent with the shutdown request to get current balance to \(0\); if the previous input were non-zero, the difference could decrease but we are in the part of the execution where this is impossible thus the agent can be shut down completely, decreasing the number of such agents. Thus as long as not all agents have shut down requests, all the shutdown requests will be honoured.

Now observe that either we have an agent with the current balance at least \(m\) by absolute value and thus output \(\pm m\), which has to coincide with the clamped
sum of the current inputs; or we have a single non-zero-balance agent (and then its current balance has to be equal to the sum of the current inputs).

Finally, once the number of agents with neither zero nor extreme internal memory stabilises at zero or one, and there are no outstanding shutdown requests, even the internal memory states stop changing. A fortiori, the outputs of the agents stabilise by that point.

B Proofs for Section 4 (Sequential composition)

Proof (of theorem 4). Let the original protocols be \((Q_1, \text{Step}_1)\) and \((Q_2, \text{Step}_2)\) with \(Q_1 = I \times Q_{1i}\) and \(Q_2 = M \times Q_{2i}\), and let the output functions be \(o_1 : Q_1 \to M\) and \(o_2 : Q_2 \to O\). Then the composition \((Q, \text{Step})\) has \(Q = Q_1 \times Q_2\) and the output function \(o : Q \to O\) mapping \((i, q_1, m, q_2)\) to \(o_2(m, q_2)\). Let the original specifications be \(\langle \varphi_1, \psi_1 \rangle\) and \(\langle \varphi_2, \psi_2 \rangle\) with composition \(\langle \varphi, \psi \rangle\).

Consider a fair execution with reconfiguration \((C_n)\) of the composition protocol \((Q, \text{Step})\). Consider the projections \(\pi_{Q_1} : (i, q_1, m, q_2) \mapsto (i, q_1)\), \(\pi_{Q_2} : (i, q_1, m, q_2) \mapsto (m, q_2)\), \(\pi_{1 \text{MO}} : (i, q_1, m, q_2) \mapsto (i, m, o(i, q_1, m, q_2) = o_2(m, q_2))\). We also apply these projections to configurations, steps, and multisets of states in a natural way.

Let us check that if we apply the first projection \(\pi_{Q_1}\) to the execution \((C_n)\), we obtain a fair execution with reconfiguration \((\pi_{Q_1}(C_n))\) of the first original protocol \((Q_1, \text{Step}_1)\).

The reconfigurations correspond to similar reconfigurations after the projection, as the projection of the shutdown state \(\pi_{Q_1}(\bot, \bot, \bot, \bot) = (\bot, \bot)\) is the shutdown state, and the input of the projection is the same as the input of the original state. Any step of the composition protocol \((Q, \text{Step})\) first performs a change that is projected to a step of the first original protocol \((Q_1, \text{Step}_1)\) or to preserving all the states, then proceeds to do changes to the components ignored by the projection \(\pi_{Q_1}\).

Moreover, at each moment every possible step of the first original protocol \((Q_1, \text{Step}_1)\) can be implemented as a projection via \(\pi_{Q_1}\) of a possible step of the composition protocol \((Q, \text{Step})\). Therefore the projection of a fair execution is also fair, and the projection of a fair execution with reconfiguration is a fair execution with reconfiguration.

As the projection \((\pi_{Q_1}(C_n))\) is a fair execution with reconfiguration of the first original protocol, the outputs according to the first original protocol eventually stabilise. Moreover, if the input of an agent stabilises at \(\bot\), so do its internal memory and output, as the first original protocol \((Q_1, \text{Step}_1)\) respects shutdown requests.

Note also that after each step of the composition protocol \((Q, \text{Step})\), if the state is \((i, q_1, m, q_2)\), then its third component \(m\) is the output of the first protocol for the projection \(m = o_1((i, q_1))\). As the first original protocol \((Q_1, \text{Step}_1)\) implements the first specification \(\varphi_1\), the stabilised multiset of inputs and the stabilised multiset of third components will be accepted by the relation \(\varphi_1\).
We now consider the result of applying the first projection $\pi_{Q_2}$ to the execution $(C_n)$. The sequence $(\pi_{Q_2}(C_n))$ is not an execution with reconfiguration of the second original protocol $(Q_2, \text{Step}_2)$, but we are going to check that it is a subsequence of a fair execution with reconfiguration. The reconfigurations adding or removing shutdown agents do the same after projection. The reconfigurations changing the inputs (in terms of the composition protocol $Q, \text{Step}$) do not change the projection at all. Now consider normal execution steps. The first part of a step (simulating the first protocol) does not change anything in terms of $(\pi_{Q_2}(C_n))$. The second part, updating the intermediate output-input, is an input-changing reconfiguration after projection. The third part of a step (simulation of the second protocol) is projected to a normal execution step. Thus the entire execution step is projected to a reconfiguration step followed by an execution step.

Furthermore, we have previously shown that the third components (the outputs of the first protocol) stabilise. Therefore after some point only normal execution steps happen in the projection. As every possible step of the second protocol $(Q_2, \text{Step}_2)$ can be simulated by a step of the composition protocol $(Q, \text{Step})$, the suffix of the projection after the stabilisation of inputs is fair. As the second protocol $(Q_2, \text{Step}_2)$ implements a specification, each agent’s output in $(\pi_{Q_2}(C_n))$ only changes a finite number of times.

Consider the sequence of some agent’s states $(i^t, q^t_1, m^t, q^t_2)$.

If the eventual stable value of $i^t$ is the shutdown request $\perp$, then after some point $q^t_1$ will always be $\perp$ as the first protocol $(Q_1, \text{Step}_1)$ respects shutdown requests, and $\pi_{Q_1}((i^t, q^t_1, m^t, q^t_2)) = (i^t, q^t_1)$ is the sequence of states of an agent in a fair execution with reconfiguration of the first protocol. From this point on, $i^t = q^t_1 = \perp$ thus $m^t = o_1((i^t, q^t_1)) = o_1((\perp, \perp)) = \perp$. As the second protocol $(Q_2, \text{Step}_2)$ respects shutdown requests and $\pi_{Q_2}((i^t, q^t_1, m^t, q^t_2)) = (m^t, q^t_2)$ is a subsequence of the sequence of states of an agent in a fair execution with reconfiguration of the second protocol, from some point on $q^t_2 = \perp$, and so the entire state $(i^t, q^t_1, m^t, q^t_2) = (\perp, \perp, \perp, \perp)$ is the shutdown state. Therefore the composition protocol $(Q, \text{Step})$ respects shutdown requests.

Otherwise, the output of the second projection (according to the second protocol), $o_2(\pi_{Q_2}((i^t, q^t_1, m^t, q^t_2))) = o_2((m^t, q^t_2))$ stabilises as shown above, thus the output according to the composition protocol also stabilises as it is the same: $o((i^t, q^t_1, m^t, q^t_2)) = o_2((m^t, q^t_2))$.

Finally, consider the sequence $(\pi_{IMO}(C_n))$. We know that eventually it stops changing forever. The multisets of the first two components are accepted by $\varphi_1$, the multisets of the last two components are accepted by $\varphi_2$, thus the multisets of the first and the last components are accepted by the composition $\varphi$ of $\varphi_1$ and $\varphi_2$. Furthermore, for each agent the first two components satisfy $\psi_1$ while the last two component satisfy $\psi_2$ thus the first and the last component satisfy the composition $\psi$.

Hence the composition protocol $(Q, \text{Step})$ respects shutdown requests; has each agent’s output change only a finite number of times in a fair execution with reconfiguration; has each agent’s output eventually be compatible with its input.
with respect to the composition of $\psi_1$ and $\psi_2$; and has the eventual multisets of inputs and outputs satisfy the composition of the relations $\varphi_1$ and $\varphi_2$. Thus it implements the composition of specifications.

C Proofs for Section 5 (Expressive power)

Proof (of theorem 2).

Note that the example protocols from example 5 (or their modulo variants) in composition with example 4 can reach consensus on the value of atomic predicates from the definition of Presburger sets, while parallel composition allows to compute the boolean combinations.

Now we can run the following protocol: the internal memory language is the Cartesian product $(O \cup \{\bot\}) \times Q_{\text{check}}$ of the output language $O$ demanded by the specification and the internal memory language $Q_{\text{check}}$ of a protocol that verifies that a specific assignment of inputs from the input language $I$ and the outputs from the output language $O$ satisfies the given specification $(\varphi, \psi)$. The intuition is that the choice of the value from $O$ is the guessed eventual output.

The output function is $o(o_{\text{guess}}, q_{\text{check}}) = o_{\text{guess}}$. As usual we identify $(\bot, \bot)$ with $\bot$.

The transitions are non-deterministically chosen between doing nothing, doing the same as for the verification (without changing the guessed outputs), and when both agents have output «No» for the verification, setting the guessed outputs to arbitrary values from $O$; also, if there is a shutdown request in the input, the guessed output component of the internal memory is changed to $\bot$.

For the verification protocol an execution with reconfiguration is simulated (the changes of guessed outputs correspond to input changes). Once the global inputs stop changing, and shutdown requests are processed, it is always possible to not change the guessed outputs long enough that the verification process converges, then change all the outputs to suitable values if the verification rejects the previous assignment, then run the verification process again without changing the guesses, then observe that the guesses can no longer be changed and the verification has stabilised.

Thus the protocol can always reach a configuration with good outputs, and in a fair execution it will do so.

Proof (of theorem 3). Consider a protocol $(Q, \text{Step})$ with input language $I$ and output language $O$, the output function being $o: Q \rightarrow O$.

Consider the set $C^\bot$ of all configurations with 1 agent and its input $\bot$, and the relation $R^*$ for reachability between such configurations via executions with reconfiguration. Consider any bottom strongly connected component $B^\bot$ of the relation $R^*$ on $C^\bot$ that is reachable from $(\bot, \bot)$, and some configuration in it, $B \in B^\bot$. Consider also the relation $R^0$ denoting reachability without reconfiguration (on all configurations). Let $S$ be the union of all of its bottom strongly connected components (BSCCs). It is a semilinear set \cite{16}.
Every configuration reachable from $B$ is reachable from empty configuration via an execution with reconfiguration, as $(\bot, \bot)$ can be obtained by an agent-adding reconfiguration, and by assumption $B^\perp$ and in particular $B$ is reachable from $(\bot, \bot)$. Moreover, if a configuration $C$ is reachable from $B$, reconfiguration replacing every input with a shutdown request $\bot$ followed by a fair execution without reconfiguration until all the agents (except at most one shut down) and agent-removal reconfigurations allows to return to some configuration $B'$ in $C^\perp$, i.e. a configuration with one agent and its input $\bot$. Note that $B'$ is reachable (with reconfiguration) from $B$ thus it has to be in the BSCC $B$, and then $B$ is reachable from $B'$. By transitivity, $B$ and $C$ are mutually reachable: $B \xrightarrow{R'} C \xrightarrow{R'} B' \xrightarrow{R'} B$. Then the set $C$ of all configurations reachable from $B$ is the set of all configurations mutually reachable with $B$ via executions with reconfiguration. As both the steps and the reconfigurations can be modelled in the standard way via vector addition systems, we can apply the result [23] that $C$ is a semilinear set.

Consider now any configuration $C^S$ from the intersection $C \cap S$ of the configurations reachable from $B$ with reconfigurations, and the configurations in BSCCs with respect to executions without reconfigurations. There is a fair execution with reconfiguration that reaches $C^S$ infinitely often (by reaching it once and then never doing any further reconfiguration). Thus $C^S$ has to satisfy the specification $(\varphi, \psi)$.

Now observe that $C \cap S$ is a semilinear set as an intersection of two semilinear sets. It has to contain configurations with all possible multisets of inputs, as we can obtain the necessary multiset of inputs via reconfiguration from $B$, and then a fair execution without reconfiguration would not change the inputs but would need to reach some BSCC of $R^0\{16\}$. That BSCC would be a part of $S$, and the overall execution with reconfiguration would certify that it would also be in $C$.

Thus we have proven that $C \cap S$ is a semilinear set of some of the configurations satisfying the specification $(\varphi, \psi)$ with at least one configuration for every possible multiset of inputs. The relation between multisets of inputs and outputs observed in $C \cap S$ is an implementable semilinear specification stronger than (or equivalent to) $(\varphi, \psi)$, as required.