Compositional Design, Implementation, and Verification of Swarms

Florian Furbach
University of Surrey, Guildford, UK
Technical University of Denmark, Kgs. Lyngby, Denmark

Lucas Clorius
Technical University of Denmark, Kgs. Lyngby, Denmark

Roland Kuhn
RKSW UG, Kassel, Germany

Hernán Melgratti
University of Buenos Aires, Argentina
Conicet, Buenos Aires, Argentina

Alceste Scalas
Technical University of Denmark, Kgs. Lyngby, Denmark

Emilio Tuosto
Gran Sasso Science Institute, L'Aquila, Italy

Abstract

Swarm protocols are a recently introduced formalism for specifying, implementing, and verifying peer-to-peer systems called *swarms*. A swarm consists of distributed agents called *machines* that communicate by asynchronous event propagation. Following a *local-first* model, each machine can progress without requiring continuous connectivity to other machines. Existing models of swarms are not compositional, making the modular development of large and complex swarm applications as well as the reuse of code difficult. We address these issues by presenting novel theory and techniques for the compositional specification, verification, and implementation of swarms. These results enable the correct compositional reuse of pre-existing swarm protocols and machine implementations. We implement these contributions in a companion software artifact which enables the automatic integration of independently designed and verified swarm components.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Distributed programming languages; Software and its engineering → Formal software verification

Keywords and phrases Swarms, Swarm Protocols, Concurrency, Distributed Coordination, Local-first Software, Behavioural Types, Publish-Subscribe, Asynchronous Communication

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.7

Related Version *Full Version*: `https://arxiv.org/abs/2604.16097` [25]

Supplementary Material *Software (ECOOP 2026 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.12.1.18`

Funding Research partially supported by the Horizon Europe grant 101093006 (TaRDIS) and by the Italian Ministero dell'Università e della Ricerca "Dipartimenti di eccellenza" initiative.

Acknowledgements This work was inspired by the discussions at the Dagstuhl Seminar 24051 "Next Generation Protocols for Heterogeneous Systems" [8]. We thank the organisers of the meeting and Schloss Dagstuhl – Leibniz Center for Informatics for making this work possible.

40th European Conference on Object-Oriented Programming (ECOOP 2026).
Editors: Robbert Krebbers and Alexandra Silva; Article No. 7; pp. 7:1–7:30

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Modern distributed systems face significant challenges in ensuring reliability, availability, and scalability. Recent work on *swarm protocols* [40, 39] addresses these challenges by formally specifying the behavior of interacting agents capable of independent yet coordinated progress. In this approach, a *swarm* is modelled as an ensemble of distributed interacting agents, called *machines*. Each machine can coordinate with others by emitting *events* (i.e., messages) that propagate asynchronously through the swarm, and by *subscribing* to (some of) the events emitted by other machines. Essentially, swarms are type-based publish/subscribe systems with distributed asynchronous delivery [24]. Unlike most publish/subscribe systems that aim for causal-consistent delivery (i.e., machines share a consistent view of the global system state), swarms are designed to operate over inconsistent views, often corresponding to inconsistent cuts of the global state [6]. This design choice prioritises availability over consistency [28], following a local-first paradigm [38, 37]: each machine can make progress independently, without requiring up-to-date global information or a continuously available network connection. Such an approach enables decentralized decision-making, reducing reliance on central coordination and improving robustness. However, it also introduces potential inconsistencies: machines may take decisions based on stale or divergent state, which may later require reconciliation [43].

The approach in [40, 39] advocates the usage of behavioral types [34, 26, 5] to ensure that a swarm correctly reconciles inconsistencies. Roughly, the intended global behaviour of the swarm (i.e., the expected interactions among machines) is formalised as a *swarm protocol*: a specification that provides a bird's-eye view of how machines playing different *roles* should interact at runtime, without causing irreconcilable inconsistencies in the overall state of the swarm. A swarm protocol G can then be *projected* onto the different roles to obtain the corresponding machine specifications. When an ensemble of machines, each running according to a projection of G, is deployed to form a swarm S, then (under *well-formedness* conditions discussed later) the swarm S is guaranteed to execute in accordance with G.

A Use Case of Swarm Protocols and Swarms. To illustrate the theory and practice of swarm protocols, we present a simplified use case in factory automation ("Industry 4.0").[1] The intended global behaviour of the swarm is specified by the Warehouse protocol illustrated in Fig. 1. A machine playing the role of a transport vehicle `T` requests a part by emitting an event of type `partReq`. A machine playing the forklift role `FL` responds by delivering the requested part to the specified pick-up position `pos`. A machine playing role `T` then picks up the part and emits an event of type `partOK`. Between consecutive requests, the door `D` may close, emitting an event of type `closingTime`.

The swarm protocol Warehouse is then *projected* onto its roles `T`, `FL`, and `D`, yielding a machine specification for each role. For example, the projection of Warehouse onto the role `D` (door) results in the state machine shown in Fig. 2. For each event type, say `closingTime`, a transition labelled `closingTime!` indicates that the machine *emits* an event of type `closingTime`, while a transition labelled `closingTime?` indicates that the machine *accepts* an event of that type. During the execution of a swarm, when a machine emits an event, the event is appended to the emitting machine's *local event log*; then, the event asynchronously propagates to the logs of other machines. A machine changes its state only

[1] We borrow the use case of an automatic warehouse operated by a swarm which has been implemented in the open source Actyx toolkit [3].

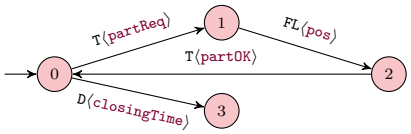

Figure 1 The swarm protocol Warehouse, involving the roles `T` (transport), `FL` (forklift), and `D` (door).

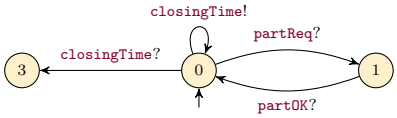

Figure 2 Machine M_D obtained by projecting the swarm protocol Warehouse (Fig. 1) onto role `D` (door).

```
1  // Machine implementation using the Actyx toolkit
2  const door = Warehouse.makeMachine('D')
3
4  // States
5  const s0 = door.designEmpty('s0')
6      .command('close', [closingTime], () => {
7          ...
8          return [closingTime.make({ time: new Date() })]
9      })
10     .finish()
11 const s1 = door.designEmpty('s1').finish()
12 const s3 = door.designEmpty('s3').finish()
13
14 // Accept events and change states
15 s0.react([partReq], s1, () => { ...; return s1.make() })
16 s0.react([closingTime], s3, () => { ...; return s3.make() })
17 s1.react([partOK], s0, () => { ...; return s0.make() })
```

Figure 3 Implementation of a Door Machine in Actyx.

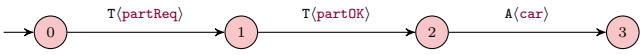

Figure 4 A Factory protocol that can interface with the Warehouse (Fig. 1) using the role `T`.

when it inspects its log and accepts an event from it; an event emission does not directly cause a state change – rather, after an event is emitted, its acceptance triggers a state transition. For example, in Fig. 2, state 0 of machine M_D has a self-loop emitting an event of type `closingTime`, as well as a transition that accepts the same event type. The non-atomic nature of event emission (_!) and acceptance (_?) implies that events from different machines may interleave: i.e., after a machine emits an event, and before that event is accepted, other events from other machines may be emitted, propagated, and accepted.

In practice, machines can be implemented using the open source Actyx toolkit [3], which provides a TypeScript API for defining swarm protocols, implementing machines, and deploying swarms. Fig. 3 shows the TypeScript implementation of the state machine M_D in Fig. 2: the machine is called `door` and plays the role `'D'` (line 2). Lines 5–12 define the states `s0`, `s1`, `s3` that map to states 0, 1, and 3 in M_D, respectively, as follows. Line 5 introduces the state named `s0` while lines 6–9 define the transition that emits an event of type `closingTime`, which is declared using the `command` method. The elided code on line 7 implements the business logic associated with the event emission, while line 8 specifies the information attached to the emitted event (in this case, a timestamp). Lines 11 and 12 introduce the states `s1` and `s3`. Finally, lines 15–17 define the accepting transitions. In particular, line 16 specifies that when the machine is in state `s0` and receives an event of type `closingTime`, it accepts the event and transitions to state `s3`. The omitted code implements the corresponding application-specific business logic.

Once the code is written, the Actyx toolkit enables static verification to check that a machine's implementation adheres to an intended behaviour. Specifically, it provides APIs for defining swarm protocols, tools for projecting protocols onto roles, a type extraction tool that reconstructs machines from TypeScript code, and a type-checking tool to check whether the projected and extracted machines are consistent [3]. For instance, the Actyx toolkit can verify that the code in Fig. 3 conforms to the machine in Fig. 2, that is projected from Fig. 1.

Going back to the theory, the use of swarm protocols ensures a key correctness property: if a swarm S consists of machines projected onto the roles of the same swarm protocol G (such as Warehouse in Fig. 1), then the swarm S enjoys *eventual fidelity* to G [40]. Intuitively, this means that, once every event propagates to every machine in the swarm S, all machines will reach a consensus on which events reflect a transition specified in G, and how they conform to an execution of G. This property also holds if, to increase reliability, the swarm includes multiple replicas of machines projected from the same role (e.g., in the example above, multiple machines may play the forklift role `FL`). The formulation and proof of eventual fidelity are quite challenging, as they must take into account the shape of G, the swarm event propagation mechanics, and the *subscriptions* of each machine (i.e., which events it sees).

Our Objective: Compositional Design and Verification of Swarm Protocols. The previously-published theory [40] and implementation [3, 39] of swarm protocols only support *monolithic* swarm design and implementation: i.e., they do not support developing a large and complex swarm by combining modular, reusable swarm protocols and swarms that are designed, implemented, and verified independently.

Consider the (simplified) protocol Factory shown in Fig. 4 where a transport (`T`) requests and picks a part, which is then utilised by an assembly robot (`A`) to build a car (emitting an event of type `car`). Notably, the Factory swarm protocol only specifies `T`'s events `partReq` and `partOK`, and does not specify how the part is obtained (i.e., what happens "in between" `T`'s events); such a detail could be part of another swarm protocol – such as Warehouse (Fig. 1), where `T` gets the part from a forklift `FL`. Intuitively, the Factory protocol is a high-level view of a production process, whereas the composition of Factory and Warehouse describes the complete process where both protocols run concurrently with the transport role acting as an interface between them.

Now, suppose that the Factory protocol changes into an updated swarm protocol Factory′ (where, e.g., the assembly robot `A` coordinates with other factory devices): it would be desirable to compose Factory′ and Warehouse (unchanged), obtaining a complete, updated production process protocol. Correspondingly, the machines projected from the Warehouse swarm protocol should be *reusable:* for example, it should be possible to deploy the implementation of the door role `D` from Warehouse (shown in Fig. 3) into any swarm that is "compatible" with Warehouse, without requiring manual changes depending on whether other machines implement the protocol Factory or Factory′.

Unfortunately, this scenario cannot be addressed with swarm protocol results in literature. There is no method to design Factory and Warehouse as separate protocols and compose them at a later stage: a whole FactoryWithWarehouse swarm protocol must be designed at once, and then used to project swarm machines. Moreover, if a detail of the "factory" part of the FactoryWithWarehouse swarm protocol changes and leads to an updated protocol FactoryWithWarehouse′, the change may also impact the machines of the "warehouse" part – e.g., the machine obtained by projecting FactoryWithWarehouse′ onto role `D` may differ from the projection of FactoryWithWarehouse onto `D` in non-trivial ways, meaning that the implementation of the `Door` machine in Fig. 3 may not be reusable and must be manually revised. Therefore, developing new swarms, or adapting existing swarms to new requirements, is a challenging and time-consuming endeavour. It would be highly desirable to have methods and tools to safely compose swarm protocols and corresponding machines from [40, 39], enabling their reuse in new scenarios. In this work, we introduce such methods and tools.

Contributions and Outline of the Paper. We present a novel theory and implementation of compositional swarm design, development, and verification. Our approach is based on *interfacing roles*: given n swarm protocols $\mathsf{G}_1, \ldots, \mathsf{G}_n$ (each one being sequential, as required by [40, 39]) with suitable roles acting as interfaces, we introduce their composition $\mathsf{G}_1 \parallel \ldots \parallel \mathsf{G}_n$. Also, given n swarms $\mathsf{S}_1, \ldots, \mathsf{S}_n$ whose machines are projected from $\mathsf{G}_1, \ldots, \mathsf{G}_n$, we introduce a method to compose $\mathsf{S}_1, \ldots, \mathsf{S}_n$. This allows for creating large and complex swarms by reusing and composing machines that were previously projected from the individual swarm protocols $\mathsf{G}_1, \ldots, \mathsf{G}_n$. The details behind this broad intuition are quite subtle, due to the highly dynamic nature of swarm protocols, swarms, and their properties – and to achieve these results, we develop novel swarm-specific notions of *well-formedness* and *compositionality*.

In § 2 we provide an overview of our approach. § 3 presents our new swarm protocol composition and well-formedness, with a method for computing well-formed subscriptions. § 4 defines a projection of swarm protocols onto machines, a *branch-tracking* semantics for such machines, and proves their eventual fidelity to well-formed swarm protocols. § 5 presents our results on correctly composing swarms which realise different swarm protocols. In § 6 we discuss the companion software artifact of this paper: an extension of the open source Actyx toolkit that implements the theory of §§ 3–5 to allow the practical compositional development of swarms; we compare experimental results on different strategies for computing the subscriptions of composed swarms. We discuss related work in § 7 and conclude in § 8. Due to space limits, we provide proofs and further details in a separate technical report [25].

2 Overview and Main Results

Before presenting our development, in § 2.1 we give an intuitive account of the swarm framework in [40]. Then, in § 2.2 we outline our approach and summarise our main results.

2.1 Swarms and Swarm Protocols

As mentioned in § 1, a *swarm* is an ensemble of *machines* that can make progress independently without requiring up-to-date global information or an always-active network connection. Each machine M maintains a *local log* l_{M} of *events* received from the surrounding swarm. These events are unique and have a globally agreed-upon total order [14] (usually defined in a coordination-free manner using timestamps, which we leave implicit) and are used by M to update its state when newly arriving events alter l_{M}.

A *swarm protocol* G provides a global view of the intended interactions that should take place during the swarm execution. To ensure correct interaction at run-time, each swarm machine plays a specific *role* `R` in G. To this end, a machine is obtained by *projecting* G onto `R`; the projection operation is based on the *subscription* of role `R`, i.e., which event types a machine playing role `R` observes while running. E.g., the machine in Fig. 2 is obtained by projecting the Warehouse protocol in Fig. 1 onto role `D` using a subscription that associates `D` with all the event types of the protocol except `pos` (emitted by `FL`). The event emission `closingTime`! in state 0 adds an event of type `closingTime` to the machine's local log and leaves the machine in state 0, whereas accepting an event of type `closingTime` from the log, denoted as `closingTime`?, transitions the machine to state 3.

For an intuitive presentation of the execution of a swarm realising the Warehouse protocol, let us consider a possible run of a swarm of three machines – $\mathsf{M}_{\mathtt{T}}$, $\mathsf{M}_{\mathtt{FL}}$, and $\mathsf{M}_{\mathtt{D}}$ – respectively implementing roles `T`, `FL`, and `D`:

1. Machine $\mathsf{M}_{\mathtt{T}}$ performs a request and emits the event $\mathit{partReq}_1$ (of type `partReq`), which it adds to its local log $l_{\mathtt{T}}$.

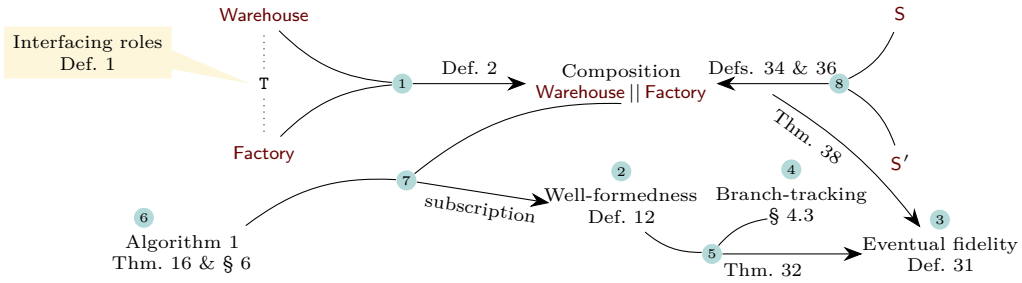

Figure 5 A synoptic view of our approach to swarm composition, with the main constructions and their dependencies.

2. This event is asynchronously propagated to the local log $l_{\mathtt{FL}} = \mathit{partReq}_1$ of $\mathsf{M}_{\mathtt{FL}}$, thus making $\mathsf{M}_{\mathtt{FL}}$ reach state 1 from where it delivers the part and emits pos_X (of type `pos`).
3. The event pos_X reaches $\mathsf{M}_{\mathtt{T}}$, which picks up the part and emits the event partOK_1 (of type `partOK`).
4. These events propagate to machine $\mathsf{M}_{\mathtt{D}}$, who adds them to its local log $l_{\mathtt{D}} = \mathit{partReq}_1 \cdot \mathit{pos}_X \cdot \mathit{partOK}_1$. It then closes the door emitting the event $\mathit{8PM}$ (of type `closingTime`).

Remarkably, when deploying a swarm, multiple machines may assume the same role in G, in a way that generalises *replicated state machines* [46]. In our Warehouse protocol for instance, other machines may play role `FL` in addition to $\mathsf{M}_{\mathtt{FL}}$. Such a machine may concurrently deliver another part by emitting an event, say pos_Y; eventually, all events propagate to all machines, and all machines reach the same log. For instance, the log $\mathit{partReq}_1 \cdot \mathit{pos}_X \cdot \mathit{pos}_Y \cdot \mathit{partOK}_1 \cdot \mathit{8PM} \cdot$ represents a case where event pos_Y is emitted after event pos_X; therefore pos_Y will be ignored by all machines. The business logic driving the machine that emitted pos_Y can detect and compensate for this reconciliation.

Following [40], the key correctness property of a swarm S with respect to a protocol G is *eventual fidelity*. Intuitively, eventual fidelity means that once the events emitted by the machines in S propagate to the whole swarm, all machines in S reach a consensus corresponding to a valid execution of G. Notably, [40] proves that eventual fidelity is guaranteed by construction whenever G is *well-formed* and the machines in S are correct projections of the roles in G; in other words, the swarm S does not need any runtime orchestration to be eventually faithful to G.

2.2 Achieving Compositional Swarms

Unfortunately, as discussed in § 1, the theory and results in [40] do not feature any form of compositionality and do not support modular swarm development. In this work, our key contribution is a new approach enabling the compositional design, implementation and verification of swarms. This is a challenging endeavour that we tackle by introducing a brand new notion of well-formedness that (*i*) ensures eventual fidelity of a swarm to a protocol, and (*ii*) is preserved by composition (unlike the well-formedness in [40]). In Fig. 5 we provide an overview of our approach showing how its main constructions are related and how they interact; the overview leverages the Warehouse and Factory composition use cases in § 1.

The pivotal notion upon which we develop our theory is *interfacing roles*, i.e., roles shared by protocols which "agree" on common event types. For instance, the transport role `T` is an interface between the Warehouse and Factory. Our composition operator $\_ \,||\, \_$ (1 in Fig. 5) essentially acts as a synchronised product of the behaviour of the protocols involved in the composition. For instance, consider the composition Warehouse || Factory of Warehouse and

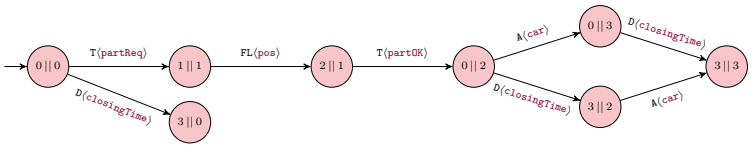

Figure 6 The composition Warehouse || Factory of Warehouse and Factory.

Factory shown in Fig. 6. All actions performed by the interfacing role T (i.e., T⟨partReq⟩ and T⟨partOK⟩) are synchronised between Warehouse and Factory while actions of non-interfacing roles (i.e., A⟨car⟩ and D⟨closingTime⟩) are interleaved. Note that (*i*) the interleaving of A⟨car⟩ and D⟨closingTime⟩ shows that they are concurrent and (*ii*) the loop in Warehouse disappears from the composed protocol since Factory requires the delivery of just one part.

We develop a novel *compositional* notion of well-formedness for swarm protocols (2). This notion provides a sufficient condition for statically verifying that a swarm is eventually faithful to a protocol (3). As part of this development, we refine the semantics of machines w.r.t. [40] by introducing a new *branch-tracking* mechanism. Specifically, in our new machine semantics, each emitted event contains a pointer to the most recent *branching*, *joining*, or *looping* event (Def. 6) that precedes it. This pointer establishes an *event causality chain* and allows machines to ignore events which are invalid because they are not caused by the expected branching/joining/looping event (this is illustrated in Example 29). Such a causality chain was not present nor needed in [40] – but it is necessary for swarms to work correctly with our compositional well-formedness, which removes restrictions that were assumed in [40]. Therefore, the new branch-tracking semantics allows us generalise the correctness results in [40]; moreover, it allows us to reduce the size of subscriptions needed to correctly track protocol choices (4), thus increasing the efficiency of the swarm. Crucially, well-formedness and branch tracking ensure that a swarm of machines projected from a protocol G is eventually faithful to G (5).

Another contribution of this paper is the definition and implementation of Algorithm 1 (6) to compute well-formed subscriptions for the composition of a set of input swarm protocols; Algorithm 1 computes the subscriptions based on the individual structures of the input protocols (e.g., Warehouse and Factory) without computing their composition (e.g., Warehouse || Factory) as it can be exponentially larger than the input protocols (7).

Finally, we develop a composition and adaptation of machines in a swarm (8), ensuring they are eventually faithful to the corresponding composition of swarm protocols.

3 Composing Swarm Protocols

In this section we introduce the composition operation on swarm protocols (§ 3.1) and our new notion of *well-formedness* (§ 3.2). We then provide an effective method for computing subscriptions that guarantee the well-formedness of a protocol with respect to those subscriptions (§ 3.3 and Theorem 16). First, as in [40], we formalise a *swarm protocol* (or simply *protocol*) G as a regular term from the following coinductive grammar:[2]

$$\mathsf{G} \quad ::\overset{\text{co}}{=} \quad \sum_{i \in I} \mathtt{R}_i\langle \mathtt{t}_i \rangle \cdot \mathsf{G}_i \qquad \text{where } \mathtt{R}_1, \mathtt{R}_2, \text{ etc. range over roles}$$

[2] A term is regular if it consists of finitely many *distinct* subterms. This condition ensures that the language generated by the co-inductive grammar is finitely representable either using the so-called "μ notation" [45] or as solutions of finite sets of equations [18]. Also, there is a correspondence between these structures and finite-state automata; the interested reader is referred to [18].

The protocol G above specifies that (any machine implementing) role $\mathtt{R}_i$ is expected to emit an event of type $\mathtt{t}_i$ (for some $i \in I$) and let the protocol continue as G_i. The summation operator $\sum$ represents a nondeterministic choice, so the order of summands is immaterial. We write $\mathsf{G} = \mathbf{0}$ if I is empty. For readability we use the infix notation for summations; for instance, for the protocol Warehouse in Fig. 1 we write:

$$\mathsf{Warehouse} = \mathtt{T}\langle\mathtt{partReq}\rangle \cdot \mathtt{FL}\langle\mathtt{pos}\rangle \cdot \mathtt{T}\langle\mathtt{partOK}\rangle \cdot \mathsf{Warehouse} \;+\; \mathtt{D}\langle\mathtt{closingTime}\rangle \cdot \mathbf{0}$$

3.1 Protocol Interface and Composition

We introduce some convenient notation: given a protocol $\mathsf{G} = \sum_{i\in I} \mathtt{R}_i\langle\mathtt{t}_i\rangle \cdot \mathsf{G}_i$, we write $\mathtt{R}\langle\mathtt{t}\rangle \in \mathsf{G}$ if $\mathtt{R}\langle\mathtt{t}\rangle$ occurs anywhere in G (and likewise for $\mathtt{t} \in \mathsf{G}$, and $\mathtt{R} \in \mathsf{G}$).

In Def. 1 below we formalise when two protocols *interface*; intuitively, this happens when event types common to the protocols are emitted by common roles only.

▶ **Definition 1** (Protocol interface). *Two protocols* G *and* G' *are* interfacing *iff for any* $\mathtt{R}\langle\mathtt{t}\rangle \in \mathsf{G}$ *and* $\mathtt{R}'\langle\mathtt{t}\rangle \in \mathsf{G}'$*, it holds that* $\mathtt{R} = \mathtt{R}'$*. The set of* interfacing roles *is* $\{\mathtt{R} \mid \mathtt{R} \in \mathsf{G} \wedge \mathtt{R} \in \mathsf{G}'\}$*. When* $\mathtt{R}\langle\mathtt{t}\rangle \in \mathsf{G}$ *and* $\mathtt{R}$ *is an interfacing role, we say* $\mathtt{t}$ *is an* interfacing event type*.*

Def. 2 below formalises a protocol composition operator $\_ \,||\, \_$ that allows two protocols to run independently (items 1 and 2) – except for the *synchronisations* imposed by the events from their interfacing roles, if any (item 3).

▶ **Definition 2** (Protocol composition). *Given protocols* $\mathsf{G} = \sum_{i\in I} \mathtt{R}_i\langle\mathtt{t}_i\rangle \cdot \mathsf{G}_i$ *and* $\mathsf{G}' = \sum_{j\in J} \mathtt{R}'_j\langle\mathtt{t}'_j\rangle \cdot \mathsf{G}'_j$ *as well as a set of roles* $\mathcal{R}$*, we coinductively define:*

$$\begin{aligned}
\mathsf{G} \,||_{\mathcal{R}}\, \mathsf{G}' \;\overset{co}{:=}\;& \textstyle\sum\{\mathtt{R}_i\langle\mathtt{t}_i\rangle \cdot (\mathsf{G}_i \,||_{\mathcal{R}}\, \mathsf{G}') \;\big|\; i \in I, \mathtt{R}_i \notin \mathcal{R}\} && (1)\\
+\;& \textstyle\sum\{\mathtt{R}'_j\langle\mathtt{t}'_j\rangle \cdot (\mathsf{G} \,||_{\mathcal{R}}\, \mathsf{G}'_j) \;\big|\; j \in J, \mathtt{R}'_j \notin \mathcal{R}\} && (2)\\
+\;& \textstyle\sum\{\mathtt{R}_i\langle\mathtt{t}_i\rangle \cdot (\mathsf{G}_i \,||_{\mathcal{R}}\, \mathsf{G}'_j) \;\big|\; i \in I, j \in J, \mathtt{R}_i\langle\mathtt{t}_i\rangle = \mathtt{R}'_j\langle\mathtt{t}'_j\rangle\} && (3)
\end{aligned}$$

We let $\_ \,||\, \_$ *be left-associative and simply write* $\mathsf{G} \,||\, \mathsf{G}'$ *instead of* $\mathsf{G} \,||_{\mathcal{R}}\, \mathsf{G}'$ *when* $\mathcal{R}$ *is the set of roles that* G *and* G' *interface on.*

By construction, $\_ \,||\, \_$ is an idempotent and commutative internal operation on protocols. Note that we do not introduce new protocol-level syntax for composed protocols: more precisely, the result of $\mathsf{G} \,||\, \mathsf{G}'$ is a protocol obtained by interleaving event types that G and G' do not share while synchronising shared event types.[3]

▶ **Example 3.** The swarm protocol composition Warehouse || Factory informally depicted in Fig. 6 is obtained by expanding Warehouse $||_{\{\mathtt{T}\}}$ Factory:

$$\begin{aligned}
\mathsf{Warehouse} \,||\, \mathsf{Factory} \;=\; & \mathtt{T}\langle\mathtt{partReq}\rangle \cdot \mathtt{FL}\langle\mathtt{pos}\rangle \cdot \mathtt{T}\langle\mathtt{partOK}\rangle \cdot \\
& \quad \big(\mathtt{A}\langle\mathtt{car}\rangle \cdot \mathtt{D}\langle\mathtt{closingTime}\rangle \cdot \mathbf{0} + \mathtt{D}\langle\mathtt{closingTime}\rangle \cdot \mathtt{A}\langle\mathtt{car}\rangle \cdot \mathbf{0}\big) \\
& \quad + \mathtt{D}\langle\mathtt{closingTime}\rangle \cdot \mathbf{0}
\end{aligned}$$

⌟

In general, when two swarm protocols are composed, only those interfacing event types emitted in both protocols in the same order are included in the composition; if interfacing event types are missing in a component protocol or emitted in different orders, then the composition excludes behaviour present in the component protocols, as shown in Example 4.[4]

[3] This design decision allows us us to maintain backward compatibility with existing tools (see § 6).
[4] Such behaviour restrictions are not consequential for our theoretical development. If undesired, they can be identified by checking whether the relevant transitions of G and G' remain present in $\mathsf{G} \,||\, \mathsf{G}'$.

▶ **Example 4** (Protocol composition and behaviour restrictions). Consider the swarm protocol Warehouse in Fig. 1, and $\mathsf{Factory}' = \mathtt{T}\langle\mathtt{partOK}\rangle \cdot \mathtt{T}\langle\mathtt{partReq}\rangle \cdot \mathtt{A}\langle\mathtt{car}\rangle \cdot \mathbf{0}$.

By Def. 2, the composition $\mathsf{Warehouse} \,||\, \mathsf{Factory}'$ yields $\mathtt{D}\langle\mathtt{closingTime}\rangle \cdot \mathbf{0}$ because the interfacing role T in $\mathsf{Factory}'$ emits an event of type partOK before those of type partReq – whereas Warehouse follows the opposite order. As another example, letting $\mathcal{R} = \{\mathtt{T}\}$, we have $\mathsf{Warehouse} \,||_{\mathcal{R}}\, \mathbf{0} = \mathtt{D}\langle\mathtt{closingTime}\rangle \cdot \mathbf{0}$ – because, by Def. 2, $\mathcal{R}$ "blocks" the actions of role T in Warehouse as it cannot synchronise with $\mathbf{0}$. ⌟

3.2 Well-Formedness of Swarm Protocols and Subscriptions

We now introduce our new notion of *well-formedness* (Def. 12), which builds upon the following three properties: *confusion-freeness* (Def. 5), *causal consistency* (Def. 8), and *determinacy* (Def. 10). Although these three properties have the same name as those in [40], their definitions deviate significantly to accommodate compositionality. Crucially, our well-formedness is more general than that of [40] (except in some minor corner cases discussed later in Footnote 11), hence we do not lose expressiveness.

Before proceeding, we introduce some additional notation. Given $\mathsf{G} = \sum_{i\in I} \mathtt{R}_i\langle\mathtt{t}_i\rangle \cdot \mathsf{G}_i$, the predicate $\mathsf{G} \xrightarrow{\mathtt{R}_i\langle\mathtt{t}_i\rangle} \mathsf{G}_i$ holds for all $i \in I$ and indicates that some role $\mathtt{R}_i$ can emit an event of type $\mathtt{t}_i$ causing the swarm protocol G to continue as G_i. We write $\mathsf{G} \xrightarrow{\mathtt{t}} \mathsf{G}$ instead of $\mathsf{G} \xrightarrow{\mathtt{R}\langle\mathtt{t}\rangle} \mathsf{G}$ when the role R is immaterial. We sometimes write $\mathsf{G} \xrightarrow{\mathtt{R}\langle\mathtt{t}\rangle}$ when there is G' such that $\mathsf{G} \xrightarrow{\mathtt{R}\langle\mathtt{t}\rangle} \mathsf{G}'$, and similarly for $\mathsf{G} \xrightarrow{\mathtt{t}}$. We write $\mathsf{G} \sqsubseteq \mathsf{G}'$ if G is a *subterm* of G'. We shorten $\mathsf{G} \xrightarrow{\mathtt{t}_1} \cdots \xrightarrow{\mathtt{t}_n} \mathsf{G}'$ to $\mathsf{G} \xrightarrow{\mathtt{t}_1\ldots\mathtt{t}_n} \mathsf{G}'$; therefore, given a sequence of event types $\mathtt{l} = \mathtt{t}_1 \ldots \mathtt{t}_n$, we write $\mathsf{G} \xrightarrow{\mathtt{l}} \mathsf{G}'$ if $\mathsf{G} \xrightarrow{\mathtt{t}_1\ldots\mathtt{t}_n} \mathsf{G}'$.

By Def. 5 a protocol is *confusion-free* if: (1) every event type is emitted by only one role; (2) event types are deterministic, i.e., an event emision leads to only one possible continuation; and (3) the same event type can be emitted by different subterms only due to concurrency, i.e., any subterm accounts for a different interleaving of independent event types.[5]

▶ **Definition 5** (Confusion-Freeness). *A protocol* G *is* confusion-free *if:*

1. *For every* $\mathtt{t} \in \mathsf{G}$*, if* $\mathtt{R}\langle\mathtt{t}\rangle, \mathtt{R}'\langle\mathtt{t}\rangle \in \mathsf{G}$*, then* $\mathtt{R} = \mathtt{R}'$*.*
2. *For every* $\mathsf{G}_1, \mathsf{G}_2, \mathsf{G}_3 \sqsubseteq \mathsf{G}$ *and* $\mathtt{t} \in \mathsf{G}$*, if* $\mathsf{G}_1 \xrightarrow{\mathtt{t}} \mathsf{G}_2$ *and* $\mathsf{G}_1 \xrightarrow{\mathtt{t}} \mathsf{G}_3$ *then* $\mathsf{G}_2 = \mathsf{G}_3$*.*
3. *There are* $\mathsf{G}_1, \ldots, \mathsf{G}_n$ *for* $n \geq 1$ *such that* $\mathsf{G} = \mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ *and for every* $\mathtt{t} \in \mathsf{G}$ *and* $i \in \{1, \ldots, n\}$*, there is at most one subterm* $\mathsf{G}' \sqsubseteq \mathsf{G}_i$ *such that* $\mathsf{G}' \xrightarrow{\mathtt{t}}$*.*

In Def. 6 below we characterise four categories of event types in a confusion-free protocol: (1) *concurrent* event types (i.e., those that are causally independent, see Fig. 7); (2) *branching* event types (i.e., those that introduce decision points, see Fig. 8); (3) *joining* event types (i.e., those that synchronize multiple concurrent branches, see Fig. 7); (4) *looping* event types (i.e., those that are repeated due to loops, see Fig. 9). Notice that these categories are not mutually exclusive: for instance, an event type can be both joining and branching.

▶ **Definition 6** (Concurrent, Branching, Joining, and Looping Event Types). *Let* G *be a confusion-free protocol. An event type* t *is:*

1. Concurrent with a $\mathtt{t}' \neq \mathtt{t}$ *if there exist* $\mathsf{G}_1, \mathsf{G}_2 \sqsubseteq \mathsf{G}$ *such that* $\mathsf{G}_1 \xrightarrow{\mathtt{t}}\xrightarrow{\mathtt{t}'} \mathsf{G}_2$ *and* $\mathsf{G}_1 \xrightarrow{\mathtt{t}'}\xrightarrow{\mathtt{t}} \mathsf{G}_2$.[6]

[5] Item 3 of Def. 5 can be expensive to check on an arbitrary G, but is immediate when the appropriate $\mathsf{G}_1, \ldots, \mathsf{G}_n$ are given upfront (which is the common case when composing protocols).

[6] Note that our definition of "concurrent events" together with confusion-freeness (Def. 5) ensure that for all $\mathsf{G}_a, \mathsf{G}_b \sqsubseteq \mathsf{G}$ and $\mathtt{t}_a, \mathtt{t}_b \in \mathsf{G}$ where $\mathtt{t}_a, \mathtt{t}_b$ are concurrent and $\mathsf{G}_a \xrightarrow{\mathtt{t}_a}\xrightarrow{\mathtt{t}_b} \mathsf{G}_b$, we also have $\mathsf{G}_a \xrightarrow{\mathtt{t}_b}\xrightarrow{\mathtt{t}_a} \mathsf{G}_b$.

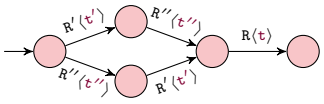

Figure 7 $\mathtt{t}'$ and $\mathtt{t}''$ are concurrent; $\mathtt{t}$ is joining for $\mathtt{t}'$ and $\mathtt{t}''$.

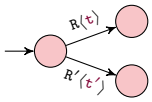

Figure 8 $\mathtt{t}$ is branching with $\mathtt{t}'$.

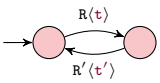

Figure 9 $\mathtt{t}$ and $\mathtt{t}'$ are looping.

2. Branching with a $\mathtt{t}' \neq \mathtt{t}$ at $\mathsf{G}_1 \sqsubseteq \mathsf{G}$ *if there are* $\mathsf{G}_2, \mathsf{G}_3 \sqsubseteq \mathsf{G}$ *with* $\mathsf{G}_1 \xrightarrow{\mathtt{t}} \mathsf{G}_2$, $\mathsf{G}_1 \xrightarrow{\mathtt{t}'} \mathsf{G}_3$ *such that* $\mathsf{G}_2 \neq \mathsf{G}_3$ *and* $\mathtt{t}, \mathtt{t}'$ *are not concurrent.*
3. Joining for $\mathtt{t}'$ and $\mathtt{t}''$ at G_3 *if there are* $\mathsf{G}_1, \mathsf{G}_2, \sqsubseteq \mathsf{G}$ *such that* $\mathsf{G}_1 \xrightarrow{\mathtt{t}'} \mathsf{G}_3$, $\mathsf{G}_2 \xrightarrow{\mathtt{t}''} \mathsf{G}_3$, *and* $\mathsf{G}_3 \xrightarrow{\mathtt{t}}$ *where* $\mathtt{t}'$ *and* $\mathtt{t}''$ *are concurrent, but neither of them is concurrent with* $\mathtt{t}$.
4. Looping *if there are* $\mathsf{G}' \sqsubseteq \mathsf{G}$ *and* $\mathtt{l}$ *such that* $\mathsf{G}' \xrightarrow{\mathtt{l}} \xrightarrow{\mathtt{t}} \mathsf{G}'$.

A subscription σ (Def. 7 below) specifies which roles observe which event types; Def. 8 then defines when a swarm protocol G is causally consistent w.r.t. a subscription σ. Intuitively, causal consistency enforces two key requirements. First, each role must subscribe to the event types it emits: this will be necessary later, when we define how to *project* machines from G based on σ.[7] Second, to follow the causal dependencies specified in the protocol, roles must subscribe to the event types directly preceding the events they emit.

▶ **Definition 7** (Subscription). *A* subscription σ *is a mapping from roles to sets of event types; i.e.,* $\sigma(\mathtt{R})$ *is the set of event types that role* $\mathtt{R}$ *subscribes to.*

▶ **Definition 8** (Causal-consistent swarm protocol). *A protocol* G is causal-consistent for a subscription σ *if, for all* $\mathtt{t}', \mathtt{R}\langle\mathtt{t}\rangle \in \mathsf{G}$*:*

1. $\mathtt{t} \in \sigma(\mathtt{R})$ *and*
2. $\mathtt{t}' \in \sigma(\mathtt{R})$ *if* $\mathtt{t}'$ *and* $\mathtt{t}$ *are not concurrent and there is* $\mathsf{G}_1 \sqsubseteq \mathsf{G}$ *such that* $\mathsf{G}_1 \xrightarrow{\mathtt{t}'} \xrightarrow{\mathtt{R}\langle\mathtt{t}\rangle}$.

▶ **Example 9** (Causal consistency). Consider the Factory protocol in Fig. 4. By causal consistency, the subscription $\sigma(\mathtt{A})$ of the Assembly robot must include event types `car` and `partOK` – respectively by the first and second condition of Def. 8. ⌟

Def. 10 below characterises subscriptions of *determinate* protocols, namely those where each role $\mathtt{R}$ subscribes to the events that are crucial for tracking causality in a swarm protocol G. Intuitively, depending on the category of an event type $\mathtt{t}$ in G (branching, joining, or looping), Def. 10 require that (*i*) role $\mathtt{R}$ must subscribe to any event types that are branching with $\mathtt{t}$ (clause *Branching*), (*ii*) $\mathtt{R}$ must subscribe to any concurrent event types that immediatly precede $\mathtt{t}$ (clause *Joining*), and (*iii*) each role $\mathtt{R}$ must observe each loop iteration, hence there must be at least one event type in each loop that all roles subscribe to (clause *Looping*). Def. 10 relies on the set $\mathit{roles}(\mathtt{t}, \mathsf{G}, \sigma)$, which is the set of roles that subscribe to event types in G that causally depend on $\mathtt{t}$ according to σ. Formally,

$$\begin{aligned} \mathit{roles}(\mathtt{t}, \mathsf{G}, \sigma) \quad = \quad \{\mathtt{R} \mid\ & \text{there are } n \geq 0,\ \mathtt{t}_0, \ldots, \mathtt{t}_n,\ \mathtt{l}_1, \ldots, \mathtt{l}_n \text{ such that:} \\ & \mathtt{t}_0 = \mathtt{t},\ \mathsf{G} \xrightarrow{\mathtt{t}_0} \xrightarrow{\mathtt{l}_1} \xrightarrow{\mathtt{t}_1} \cdots \xrightarrow{\mathtt{l}_n} \xrightarrow{\mathtt{t}_n} \text{ and} \\ & \mathtt{t}_n \in \sigma(\mathtt{R}),\ \text{and } \mathtt{t}_i, \mathtt{t}_{i+1} \text{ not concurrent for all } 0 \leq i < n\} \end{aligned}$$

[7] As mentioned in § 2.1, and later formalised in Def. 25, if a machine playing role $\mathtt{R}$ emits an event of type $\mathtt{t}$, that machine will *not* change state unless it has an accepting transition for $\mathtt{t}$. Subscribing $\mathtt{R}$ to $\mathtt{t}$ ensures that the projection of a swarm protocol onto its roles (using Def. 21) will produce machines that can change state by accepting every event type they emit.

▶ **Definition 10** (Determinate protocol subscriptions). *A protocol* G is determinate for subscription σ *if, for all* $\mathsf{G}' \sqsubseteq \mathsf{G}$ *and* $\mathtt{t}, \mathtt{R} \in \mathsf{G}$

Branching: *If* $\mathtt{t}$ *is branching with* $\mathtt{t}'$ *at* G' *and* $\mathtt{R} \in \mathit{roles}(\mathtt{t}, \mathsf{G}', \sigma)$*, then* $\mathtt{t}, \mathtt{t}' \in \sigma(\mathtt{R})$.

Joining: *If* $\mathtt{t}$ *is joining for* $\mathtt{t}'$ *and* $\mathtt{t}''$ *at* G' *and* $\mathtt{R} \in \mathit{roles}(\mathtt{t}, \mathsf{G}', \sigma)$*, then* $\mathtt{t}, \mathtt{t}', \mathtt{t}'' \in \sigma(\mathtt{R})$.

Looping: *If* $\mathsf{G}' \xrightarrow{\mathtt{l}} \mathsf{G}'$ *for some non-empty* $\mathtt{l}$ *then there are* $\mathtt{t}' \in \mathtt{l}$ *and* $\mathsf{G}'' \sqsubseteq \mathsf{G}'$ *such that* $\mathsf{G}'' \xrightarrow{\mathtt{t}'}$ *and* $\mathtt{t}' \in \sigma(\mathtt{R})$ *for all* $\mathtt{R} \in \mathit{roles}(\mathtt{t}', \mathsf{G}'', \sigma)$*. We call* $\mathtt{t}'$ *looping in* σ.

▶ **Example 11** (Branching vs. looping events). In Fig. 1, the loop $\mathsf{Warehouse} \xrightarrow{\mathtt{partReq}} \xrightarrow{\mathtt{pos}} \xrightarrow{\mathtt{partOK}} \mathsf{Warehouse}$ can be exited by event type `closingTime`, which branches with `partReq`. Consequently, subscribing to `partReq` and `closingTime` is required by clause *branching* of Def. 10 – and the subscription to `partReq` also satisfies the clause *looping*. In general, if a protocol contains a loop with a branching that exits the loop, then each branching event type that continues the loop is also a looping event type in the subscription (by Def. 6) and subscribing to it satisfies clause *looping* of Def. 10. ⌟

We now have all the ingredients to formalise when a swarm protocol is well-formed.

▶ **Definition 12** (Well-formed swarm protocols). *A protocol* G *is* σ-well-formed *if it is confusion-free, causal-consistent in* σ*, and determinate for* σ *(Definitions 5, 8, and 10).*

▶ **Example 13** (Well-formed protocol and subscription). The protocol $\mathsf{Warehouse} \,||\, \mathsf{Factory}$ in Example 3 and Fig. 6 is confusion-free (by Def. 5). Now, take the assembly robot `A` and a subscription σ such that $\mathsf{Warehouse} \,||\, \mathsf{Factory}$ is σ-well-formed (by Def. 12): rule (1) of causal consistency (Def. 8) requires $\mathtt{car} \in \sigma(\mathtt{A})$, implying $\mathtt{A} \in \mathit{roles}(\mathtt{partReq}, \mathsf{Warehouse} \,||\, \mathsf{Factory}, \sigma)$; therefore, from the *branching* rule of determinacy (Def. 10), we have $\mathtt{partReq}, \mathtt{closingTime} \in \sigma(\mathtt{A})$. The *joining* and *looping* rules of Def. 10 hold vacuously, as there are no joining event types or loops in $\mathsf{Warehouse} \,||\, \mathsf{Factory}$. ⌟

3.3 On Computing Subscriptions

This section introduces Algorithm 1 for computing subscriptions that guarantee well-formedness (by Def. 12) for a composition $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ of given swarm protocols $\mathsf{G}_1 \ldots \mathsf{G}_n$. In general, for any confusion-free swarm protocol G there is at least one subscription σ such that G is σ-well-formed: it is the "total" subscription where each role in G subscribes to all event types in G. However, if such a "total" subscription is used to project machines from G (using Def. 21 later on), then the resulting swarm would consist of complex machines that await the emission of all events emitted by all other machines, leading to inefficient executions. It is hence worthwhile to compute smaller subscriptions that preserve well-formedness. Importantly, for a composition $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ of n *composable* protocols (by Def. 14 below), our Algorithm 1 operates on the input protocols without expanding their composition (Def. 2), as the size of the expanded protocol may grow exponentially with n (as shown in § 6.2).

▶ **Definition 14** (Sequential and composable swarm protocols). *A protocol is* sequential *if it does not contain concurrent events. A set of protocols is* composable *if they are pairwise interfacing and each protocol is sequential and confusion-free.*

Algorithm 1 takes a set $\{\mathsf{G}_i\}_{i \in I}$ of composable protocols together with a set $\{\sigma_i\}_{i \in I}$ of subscriptions, and generates a subscription σ such that $\mathsf{G} = \mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ is σ-well-formed and $\sigma_i \subseteq \sigma$ holds for all $i \in I$. The algorithm initialises three sets:

- σ, initially approximating the result as the union of the given σ_i. Each σ_i can be understood as the specification of the event types that each role in G_i is interested in.

Algorithm 1 Computing a subscription for a composition of protocols.

Input: A set $\{\mathsf{G}_1, \ldots, \mathsf{G}_n\}$ of composable protocols and a set of subscriptions $\{\sigma_1, \ldots, \sigma_n\}$
Output: A subscription $\sigma \supseteq \sigma_1, \ldots, \sigma_n$ such that $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ is σ-well-formed.

Let $\mathit{subscribers}(\mathsf{G}, \sigma)$ denote the roles that subscribe to some event type of G in σ.
$\sigma := \bigcup_{i \leq n} \sigma_i$
$\mathtt{ifr} := \{\mathtt{R} \mid \exists i, j \leq n : i \neq j \wedge \mathtt{R} \in \mathsf{G}_i \wedge \mathtt{R} \in \mathsf{G}_j\}$
$\mathtt{conc} := \{\{\mathtt{t}_i, \mathtt{t}_j\} \mid i, j \leq n, \mathtt{R}_i\langle \mathtt{t}_i \rangle \in \mathsf{G}_i, \mathtt{R}_j\langle \mathtt{t}_j \rangle \in \mathsf{G}_j, \mathtt{R}_i \notin \mathsf{G}_j, \mathtt{R}_j \notin \mathsf{G}_i\}$
repeat
  for all $i \leq n$, G'_i, $\mathsf{G}''_i \sqsubseteq \mathsf{G}_i$, and $\mathtt{R}$, $\mathtt{R}'$, $\mathtt{t}$, $\mathtt{t}' \in \mathsf{G}_i$ **do**
    Causal Consistency: If $\mathsf{G}'_i \xrightarrow{\mathtt{R}\langle \mathtt{t} \rangle}$, then add $\mathtt{t}$ to $\sigma(\mathtt{R})$; If $\mathsf{G}'_i \xrightarrow{\mathtt{t}'} \xrightarrow{\mathtt{R}\langle \mathtt{t} \rangle}$, then add $\mathtt{t}'$ to $\sigma(\mathtt{R})$.
    Branching: If $\mathtt{t}$ is branching with $\mathtt{t}'$ at G'_i and $\mathtt{R} \in \mathit{subscribers}(\mathsf{G}'_i, \sigma)$, then add $\mathtt{t}, \mathtt{t}'$ to $\sigma(\mathtt{R})$.
    Joining: For all $j \leq n$, $\mathsf{G}'_j \sqsubseteq \mathsf{G}_j$, and $\mathtt{t}'' \in \mathsf{G}_j$ such that $\mathsf{G}'_i \xrightarrow{\mathtt{t}'} \xrightarrow{\mathtt{t}}$, $\mathsf{G}'_j \xrightarrow{\mathtt{t}''} \xrightarrow{\mathtt{t}}$, $\{\mathtt{t}', \mathtt{t}''\} \in \mathtt{conc}$, $\{\mathtt{t}', \mathtt{t}\} \notin \mathtt{conc}$, $\{\mathtt{t}'', \mathtt{t}\} \notin \mathtt{conc}$, and $\mathtt{R} \in \mathit{subscribers}(\mathsf{G}'_i, \sigma)$: Add $\mathtt{t}'$, $\mathtt{t}''$, $\mathtt{t}$ to $\sigma(\mathtt{R})$.
    Interfacing: If $\mathsf{G}'_i \xrightarrow{\mathtt{R}\langle \mathtt{t} \rangle} \mathsf{G}''_i$ with $\mathtt{R} \in \mathtt{ifr}$ and $\mathtt{R}' \in \mathit{subscribers}(\mathsf{G}''_i, \sigma)$, then add $\mathtt{t}$ to $\sigma(\mathtt{R}')$.
  end for
until σ no longer changes
for all $i \leq n$, $\mathsf{G}'_i \sqsubseteq \mathsf{G}_i$, such that $\mathsf{G}'_i \xrightarrow{\mathtt{t} \cdot \mathtt{l}} \mathsf{G}'_i$ for some $\mathtt{t}$ and $\mathtt{l}$, and $\mathtt{R} \in \mathit{subscribers}(\mathsf{G}'_i, \sigma)$ **do**
  Looping: If G'_i has no looping event type in σ, then add $\mathtt{t}$ to $\sigma(\mathtt{R})$.
end for

- `ifr` containing all interfacing roles.
- `conc` containing all potentially concurrent event types, i.e., the pairs of event types belonging to different components that are emited by non-interfacing roles.

The algorithm iteratively enlarges σ by applying monotone operations corresponding to the definitions of causal-consistency (Def. 8) and *branching* and *joining* of determinacy (Def. 10), together with the new operator *interfacing*, which is crucial for ensuring the correct synchronisation of the protocols (as will be discussed below). The iteration terminates when a fixed point is reached. The final phase ensures that every role that is involved in a term that contains a loop $\mathtt{l}$ is subscribed to (at least) one looping event of $\mathtt{l}$.

On the need of the *interfacing* step in Algorithm 1. One of the requirements of well-formedness (Def. 12) is *determinacy* (Def. 10). Checking the determinacy of a composed swarm protocol for a subscription requires the set $\mathit{roles}(\mathtt{t}, \mathsf{G}', \sigma)$ for subterms $\mathsf{G}' = \mathsf{G}'_1 \,||\, \ldots \,||\, \mathsf{G}'_n$ – and computing that set would require the explicit computation of the expanded protocol G' (using Def. 2); however, the size of the expanded G' can be exponential in n. To avoid this exponential blow-up, Algorithm 1 uses the set $\mathit{subscribers}(\mathsf{G}'_i, \sigma)$ (where G'_i is a subterm of one of the input protocols G_i) and its *interfacing* step to over-approximate the set $\mathit{roles}(\mathtt{t}, \mathsf{G}, \sigma)$ without explicitly expanding $\mathsf{G} = \mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$. To see why this approach yields the desired over-approximation, recall that $\mathtt{R} \in \mathit{roles}(\mathtt{t}, \mathsf{G}', \sigma)$ holds if there exists a path $\mathsf{G}' \xrightarrow{\mathtt{t}_0} \xrightarrow{\mathtt{l}_1} \xrightarrow{\mathtt{t}_1} \cdots \xrightarrow{\mathtt{l}_n} \xrightarrow{\mathtt{t}_n}$ such that $\mathtt{t}_0 = \mathtt{t}$, $\mathtt{t}_n \in \sigma(\mathtt{R})$, and every pair $\mathtt{t}_i, \mathtt{t}_{i+1}$ is not concurrent for $0 \leq i < n$. Since $\mathtt{t}_i$ and $\mathtt{t}_{i+1}$ are not concurrent, they must either belong to the same concurrency-free input swarm protocol or they are separated by interfacing event types that enforce their ordering. Consequently, there exists a path $\mathsf{G}' \xrightarrow{\mathtt{t}'_0} \xrightarrow{\mathtt{l}'_1} \xrightarrow{\mathtt{t}'_1} \cdots \xrightarrow{\mathtt{l}'_{n'}} \xrightarrow{\mathtt{t}'_{n'}}$ satisfying the conditions above, where $\mathtt{t}'_1, \ldots, \mathtt{t}'_{n'-1}$ are interfacing event types and $\mathtt{t}'_i$, $\mathtt{t}'_{i+1}$ occur within the same input protocol for $0 \leq i < n'$. The *interfacing* step propagates subscriptions along this path, that is, if $\mathtt{t}'_{i+1} \in \sigma(\mathtt{R})$, then also $\mathtt{t}'_i \in \sigma(\mathtt{R})$. Starting from $\mathtt{t}'_{n'} \in \sigma(\mathtt{R})$, repeated application of the *interfacing* step backwards along the path ensures that $\mathtt{t}'_1 \in \sigma(\mathtt{R})$. Since $\mathtt{t}'_1$ is in the same input protocol G'_i as $\mathtt{t}$, we obtain $\mathtt{R} \in \mathit{subscribers}(\mathsf{G}'_i, \sigma)$. *(This argument is formally stated in [25, §A, Lemma 39].)*

▶ **Example 15** (Application of Algorithm 1 to Warehouse and Factory)**.** We illustrate the application of Algorithm 1 to the protocols Warehouse and Factory (depicted in Figures 1 and 4, respectively) and two empty subscriptions $\sigma_1 = \sigma_2 = \emptyset$. It is straightforward to check that these two protocols are composable, since they are sequential, confusion-free and they interface on `T`. Then, the algorithm initialises $\sigma = \sigma_1 \cup \sigma_2 = \emptyset$, $\texttt{ifr} = \{\texttt{T}\}$, and $\texttt{conc} = \{\{\texttt{closingTime}, \texttt{car}\}, \{\texttt{pos}, \texttt{car}\}\}$. Note that `conc` over-approximates the actual pairs of concurrent event types. Indeed, only `closingTime` and `car` are concurrent in the composition Warehouse || Factory (see Fig. 6). Now the algorithm starts the iteration with the step *causal consistency*. Starting with role `T` and protocol Factory, we first add the event types `partReq` and `partOK` to $\sigma(\texttt{T})$ because Factory has transitions where `T` emits such event types. In Warehouse, the same role `T` emits `partOK` after `FL` emits `pos`; consequently `pos` is added to $\sigma(\texttt{T})$. Subscriptions for the remaining roles, shown below, are obtained analogously.

$$\sigma = \left\{ \begin{array}{ll} \texttt{T} \mapsto \{\texttt{partReq}, \texttt{partOK}, \texttt{pos}\}, & \texttt{FL} \mapsto \{\texttt{partReq}, \texttt{pos}\}, \\ \texttt{D} \mapsto \{\texttt{partOK}, \texttt{closingTime}\}, & \texttt{A} \mapsto \{\texttt{partOK}, \texttt{car}\} \end{array} \right\}$$

Then, the *branching* step is executed. Note that `partReq` and `closingTime` are branching in Warehouse. Therefore, (*i*) `partReq` is added to the subscriptions of the roles already subscribed to event types of Warehouse, i.e., to `D` and `A`; (*ii*) analogously, `closingTime` is added to `T`, `FL`, and `A`. Step *joining* is not actually applied in this example because its conditions are not satisfied. Next, the *interfacing* step adds `partOK` to all roles that have subscribed to some event that follows `partOK` in some of the protocols; hence, `partOK` is added to `FL` because it already subscribed to `partReq`.

The fixed point of the iteration is:

$$\sigma = \left\{ \begin{array}{ll} \texttt{T}, \ \texttt{FL} \mapsto \{\texttt{partReq}, \texttt{partOK}, \texttt{closingTime}, \texttt{pos}\} & \\ \texttt{D} \mapsto \{\texttt{partReq}, \texttt{partOK}, \texttt{closingTime}\}, & \texttt{A} \mapsto \{\texttt{partReq}, \texttt{partOK}, \texttt{closingTime}, \texttt{car}\} \end{array} \right\}$$

In Warehouse, the event type `partReq` is looping in σ. However, it was already added to the relevant roles due to the branching rule. Thus, the *looping* phase is not applied. ⌟

Theorem 16 below states that Algorithm 1 computes a subscription σ ensuring well-formedness of the composed protocol. *(Proof in [25, §A].)*

▶ **Theorem 16** (Correctness of Algorithm 1)**.** *If σ is a subscription computed from composable protocols $\mathsf{G}_1, \ldots, \mathsf{G}_n$ and $\sigma_1, \ldots, \sigma_n$ using Algorithm 1, then $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ is σ-well-formed.*

4 Realising Swarm Protocols as Swarms of Machines

We now address the problem of correctly *realising* a swarm protocol G (Def. 28), i.e., constructing a swarm of machines that interact as specified by G. We begin by fixing the language for writing machines (§ 4.1). Following a top-down approach, we define a *projection* operation (§ 4.2) to derive correct machines for the different roles of a protocol. Then, we refine the swarm model introduced in [40] by adding a *branch-tracking* mechanism to the semantics of machines (§ 4.3) in a swarm (§ 4.4). This revised semantics is essential for (1) ensuring that the machines projected from a swarm protocol G are *eventually faithful* to G (§ 4.5), and (2) enabling the composition of swarms (presented in § 5).

4.1 Machines

A *machine* models a swarm agent that processes a local log of events to make decisions on the emission of new events. We define a machine as a grammar term (Def. 17) with a corresponding finite-state automaton (Def. 18).

▶ **Definition 17** (Machine). *A* machine *is a* regular term *from the coinductive grammar:*

$$\mathsf{M} \ ::\overset{co}{=} \ \kappa \cdot \&_{i \in I} \mathtt{t}_i ? \, \mathsf{M}_i \qquad \textit{where } \kappa \textit{ is a finite set of event types dubbed the } \text{emitter set}$$

with $\mathtt{t}_i \neq \mathtt{t}_j$ *for all* $i \neq j \in I$. *We write* $\mathbf{0}$ *when* $\kappa = I = \emptyset$.

▶ **Definition 18** (Machine states and transitions). *A machine* M *from Def. 17 can be described as the deterministic finite-state automaton where:*

- *Each distinct subterm of* M *(finitely many since* M *is regular) corresponds to a* ***state****.*
- *The* ***initial state*** *is* M *itself.*
- *There is an* ***event acceptance transition*** $\mathsf{M}_1 \xrightarrow{\mathtt{t}?} \mathsf{M}_2$ *whenever machine* M_1 *has a branch* $\mathtt{t}?$ *leading to* M_2
- *and an* ***event emission transition*** $\mathsf{M}' \xrightarrow{\mathtt{t}!} \mathsf{M}'$ *for any* $\mathtt{t}$ *in the emitter set of* M'.

▶ Remark 19. The grammar-based presentation of machines (Def. 17) and the automaton presentation (Def. 18) are equivalent: each distinct subterm corresponds to a state, branches correspond to acceptance transitions, and emitter sets yield emission self-loops.

▶ **Example 20.** By Def. 18, the automaton depicted in Fig. 2 corresponds to the machine:[8]

$$\mathsf{M}_{\mathtt{D}} \ = \ \{\mathtt{closingTime}\} \cdot \Big((\mathtt{partReq} ? \, \mathtt{partOK} ? \, \mathsf{M}) \ \& \ (\mathtt{closingTime} ? \, \mathbf{0}) \Big)$$

Note that a closing-time event can be emitted only from $\mathsf{M}_{\mathtt{D}}$ (the initial state in the automaton of Fig. 2) since `closingTime` belongs only to the emitter set of $\mathsf{M}_{\mathtt{D}}$. Also note that in Fig. 2, emitted events appear as *self-loops*: this is because (by Def. 18 above) a machine remains in its current state while emitting an event; a state transition may occur only when a machine accepts an event from its log. ⌟

4.2 Projecting a Swarm Protocol onto its Roles

By Def. 21 below, the *projection* of a swarm protocol G onto a role R yields a machine that captures the behaviour of R in G. Specifically, the projected machine retains the event types that are either emitted or subscribed to by R – and it disregards all other event types.

▶ **Definition 21** (Projection). *We coinductively define the* projection *of a* σ*-well-formed swarm protocol* G *onto a role* R *as:*

$$\mathsf{G}\!\downarrow^{\sigma}_{\mathtt{R}} \ \overset{co}{=} \ \kappa \cdot \&\Big\{ \mathtt{t}?(\mathsf{G}'\!\downarrow^{\sigma}_{\mathtt{R}}) \ \Big| \ \exists \mathtt{l} : \mathsf{G} \xrightarrow{\mathtt{l}} \xrightarrow{\mathtt{t}} \mathsf{G}' \textit{ and } \mathtt{l} \cap \sigma(\mathtt{R}) = \emptyset \textit{ and } \mathtt{t} \in \sigma(\mathtt{R}) \Big\}$$

where $\kappa = \Big\{ \mathtt{t} \ \Big| \ \exists \mathtt{l} : \mathsf{G} \xrightarrow{\mathtt{l}} \xrightarrow{\mathtt{R}\langle \mathtt{t} \rangle} \textit{ and } \mathtt{l} \cap \sigma(\mathtt{R}) = \emptyset \Big\}$.

Observe that, by Def. 21, the initial state of the projected machine $\mathsf{G}\!\downarrow^{\sigma}_{\mathtt{R}}$ has:

- A branch $\mathtt{t}?$ for every event type $\mathtt{t}$ that R subscribes to (by σ) and can be reached from G after a (possibly empty) sequence of events of type $\mathtt{l}$ *not* subscribed to by R.
- An emitter set κ consisting of all the event types that R can emit in G after a (possibly empty) sequence of events of type $\mathtt{l}$ *not* subscribed to by R.

Hence, the projection $\mathsf{G}\!\downarrow^{\sigma}_{\mathtt{R}}$ may start by either emitting an event of a type in κ or proceed to $\mathsf{G}'\!\downarrow^{\sigma}_{\mathtt{R}}$ upon accepting an event of type $\mathtt{t}$. Note that an event type $\mathtt{t}$ that is *not* subscribed to by R does *not* appear in $\mathsf{G}\!\downarrow^{\sigma}_{\mathtt{R}}$, i.e., the projected machine ignores all events of such a type $\mathtt{t}$.

[8] For readability, in our examples we often use brackets and infix notation for the branching operator &.

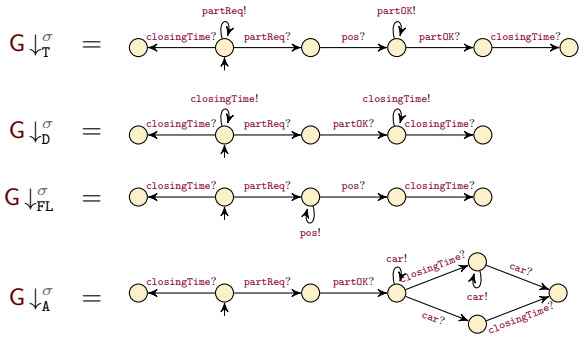

Figure 10 Projection of G = Warehouse || Factory (from Fig. 6) on roles T, D, FL, and A, using the subscription σ defined in Example 22.

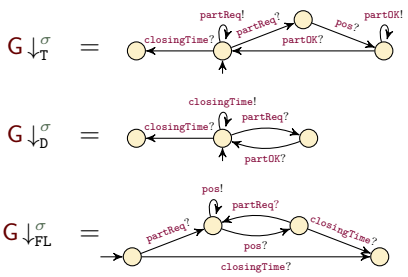

Figure 11 Projection of G = Warehouse (from Fig. 1) on roles T, D, and FL, using the subscription σ defined in Example 29.

▶ **Example 22** (Projection). Consider the swarm protocol G = Warehouse || Factory shown in Fig. 6 and the following subscription (where E stands for the set of all event types in G):

$$\sigma \;=\; \{\mathtt{T} \mapsto E \setminus \{\mathtt{car}\},\;\; \mathtt{D} \mapsto E \setminus \{\mathtt{pos}, \mathtt{car}\},\;\; \mathtt{FL} \mapsto E \setminus \{\mathtt{partOK}, \mathtt{car}\},\;\; \mathtt{A} \mapsto E \setminus \{\mathtt{pos}\}\}$$

The projections of G are shown in Fig. 10. ⌟

Notice that in Def. 21, when constructing the projection $\mathsf{G}\!\downarrow_{\mathtt{R}}^{\sigma}$ there might be multiple $\mathtt{l}$ and G_i such that $\mathsf{G} \xrightarrow{\mathtt{l}}\!\xrightarrow{\mathtt{t}_i} \mathsf{G}_i$, meaning the projected machine can accept or emit $\mathtt{t}_i$ and then reach one of potentially many different options for $\mathsf{G}_i\!\downarrow_{\mathtt{R}}^{\sigma}$. Still, Lemma 23 ensures that if G is well-formed, then $\mathsf{G}\!\downarrow_{\mathtt{R}}^{\sigma}$ relates G to only one unique machine.

▶ **Lemma 23.** *If* G *is a* σ*-well-formed protocol, then* $\mathsf{G}\!\downarrow_{\mathtt{R}}^{\sigma}$ *is uniquely defined for all* R.

Proof. *(Outline; full proof in [25, §B].)* The σ-well-formedness property ensures that the projection satisfies $\mathsf{G}'\!\downarrow_{\mathtt{R}}^{\sigma} = \mathsf{G}''\!\downarrow_{\mathtt{R}}^{\sigma}$ for all sub-terms G' and G'' reachable from G by event types not in $\sigma(\mathtt{R})$ – so it does not matter which one is chosen in the construction of $\mathsf{G}\!\downarrow_{\mathtt{R}}^{\sigma}$. This is because the subscription σ includes all event types that can influence the future behaviour of the projection – so in Def. 21, the event types selected in $\mathtt{l}$ (which are not in σ) do not influence the shape of $\mathsf{G}_i\!\downarrow_{\mathtt{R}}^{\sigma}$. ◀

4.3 Branch-Tracking Machine Semantics

In this section we define how machines emit events and process their local log of events. We fix a set of *events*, ranged over by e; each event has a unique *event type* $\mathtt{t}$. An *event log* l is a sequence $e_1 \cdot e_2 \cdot \ldots$ of pairwise distinct events. As anticipated in § 2.2, we incorporate a new *branch-tracking mechanism* to the machine semantics originally proposed in [40] to establish an explicit causal link between events. Such a link was not present nor needed in [40] – but it is necessary for swarms to work correctly with our compositional well-formedness (Def. 12), which removes restrictions assumed by the machine semantics in [40].

The behaviour of a machine M is determined by a *log-processing function* $\hat{\delta}$ (formalised in Def. 24 below). Let $\mathcal{T}$ be the set of all event types. In defining $\hat{\delta}$, we assume that machines are equipped with an additional element, which does not appear in [40]: A set of *updating event types* $\mathsf{M}.\mathrm{UP} \subseteq \mathcal{T}$.[9] If an event e has a type $\mathtt{t} \in \mathsf{M}.\mathrm{UP}$, we call e an *updating event*.

[9] We will see in Def. 28 that this set is populated with branching, joining, and looping event types from a swarm protocol.

We also add to each event e a field $e.\mathsf{last_{up}}$, which intuitively refers to the last updating event that caused e. A machine M processes its log starting from the oldest event. When processing an event e of type $\mathtt{t}$, M performs a transition if:

1. There is an acceptance transition $\mathsf{M} \xrightarrow{\mathtt{t}?} \mathsf{M}'$; i.e., M accepts events of type $\mathtt{t}$; and
2. $e.\mathsf{last_{up}}$ coincides with the last updating event observed by M; i.e., $e.\mathsf{last_{up}}$ is the last updating event that causally preceded e.

If the two conditions above hold, then M accepts e and moves to the next state M', i.e., the continuation of the event acceptance transition, from which the remaning events in the log are processed. Otherwise, M skips e and processes the next event in the log.

More precisely, in order to remember previously-seen updating events and check the pointer $e.\mathsf{last_{up}}$, each machine keeps a partial mapping $\mathsf{last_{up}} : \mathcal{T} \rightharpoonup \mathcal{E}$ that associates each event type $\mathtt{t}$ with the *last updating event* for the type $\mathtt{t}$. The mapping $\mathsf{last_{up}}$ is built by the machine while processing a log: initially, $\mathsf{last_{up}}$ is empty (i.e., $\emptyset$) and it is updated every time a machine processes an updating event. The updating of $\mathsf{last_{up}}$ is performed as follows. Let $\mathsf{branch}(\mathsf{M}, \mathtt{t})$ be a partial mapping that, for each machine M and event type $\mathtt{t}$, gives the set event types that continue along the same branch as $\mathtt{t}$ until the next updating event; specifically, if $\mathsf{M} \xrightarrow{\mathtt{t}?} \xrightarrow{\mathtt{t}_1?} \dots \xrightarrow{\mathtt{t}_n?}$ with $\mathtt{t}_1, \dots, \mathtt{t}_{n-1} \notin \mathsf{M}.\mathrm{UP}$ and $\mathtt{t}_1, \dots, \mathtt{t}_n$ not concurrent with $\mathtt{t}$, then $\mathtt{t}_1, \dots, \mathtt{t}_n \in \mathsf{branch}(\mathsf{M}, \mathtt{t})$. The processing of an event e of type $\mathtt{t}$ by a machine M that has an event acceptance transition $\mathsf{M} \xrightarrow{\mathtt{t}?} \mathsf{M}'$ is described by the following relation:

$$
\begin{array}{lcll}
(\mathsf{M}, \mathsf{last_{up}}) & \xrightarrow{e} & (\mathsf{M}', \mathsf{last_{up}}) & \text{if } e.\mathsf{last_{up}} = \mathsf{last_{up}}(\mathtt{t}) \text{ and } \mathtt{t} \notin \mathsf{M}.\mathrm{UP} \\
(\mathsf{M}, \mathsf{last_{up}}) & \xrightarrow{e} & (\mathsf{M}', \mathsf{last_{up}}[\mathsf{branch}(\mathsf{M}, \mathtt{t}) \mapsto e]) & \text{if } e.\mathsf{last_{up}} = \mathsf{last_{up}}(\mathtt{t}) \text{ and } \mathtt{t} \in \mathsf{M}.\mathrm{UP}.
\end{array}
$$

where $\mathsf{last_{up}}[T \mapsto e]$ denotes the update of $\mathsf{last_{up}}$, namely the mapping that maps each event type in the set T to event e, and behaves as $\mathsf{last_{up}}$ for other event types.

The mechanism through which a machine M processes a complete event log l, or equivalently determines its current state based on l, is specified by Def. 24 below.

▶ **Definition 24** (Log-processing function). *The* log-processing function $\hat{\delta}$ *is defined as* $\hat{\delta}(\mathsf{M}, l) = \delta_{BT}(\mathsf{M}, l, \emptyset)$ *where:*

$$
\begin{aligned}
\delta_{BT}(\mathsf{M}, \epsilon, \mathsf{last_{up}}) &= (\mathsf{M}, \mathsf{last_{up}}) \\
\delta_{BT}(\mathsf{M}, e \cdot l, \mathsf{last_{up}}) &= \begin{cases} \delta_{BT}(\mathsf{M}', l, \mathsf{last_{up}}') & \textit{if } (\mathsf{M}, \mathsf{last_{up}}) \xrightarrow{e} (\mathsf{M}', \mathsf{last_{up}}') \\ \delta_{BT}(\mathsf{M}, l, \mathsf{last_{up}}) & \textit{otherwise.} \end{cases}
\end{aligned}
$$

Note that in Def. 24, $\hat{\delta}$ is defined in terms of the auxiliary function δ_{BT}, which is in charge of constructing $\mathsf{last_{up}}$ while traversing the log. Observe that δ_{BT}, and consequently $\hat{\delta}$, returns a pair $(\mathsf{M}', \mathsf{last_{up}})$ where M' is the reached machine and $\mathsf{last_{up}}$ is the mapping built after processing the log. Based on the current state computed by the log-processing function $\hat{\delta}$, machines generate new events according to Def. 25 below.

▶ **Definition 25** (Operational semantics of machines). *A machine* M *with a local log* l *emits events according to the following rule:*

$$
\frac{\hat{\delta}(\mathsf{M}, l) = (\kappa \cdot \&_{i \in I} \mathtt{t}_i?\, \mathsf{M}_i,\ \mathsf{last_{up}}) \qquad \mathtt{t} \in \kappa \qquad e \textit{ has type } \mathtt{t} \qquad e.\mathsf{last_{up}} = \mathsf{last_{up}}(\mathtt{t})}{(\mathsf{M}, l) \xRightarrow{\mathtt{t}!} (\mathsf{M}, l \cdot e)}\ [\textsc{emit}]
$$

The transition $(\mathsf{M}, l) \xRightarrow{\mathtt{t}!} (\mathsf{M}, l \cdot e)$ in Def. 25 reads as: *"a machine* M *with a log* l *emits an event* e *of type* $\mathtt{t}$*."* Observe that the new event e is added to the end of the local log, and the transition is possible only if, after processing l, the machine reaches a state $\kappa \cdot \&_{i \in I} \mathtt{t}_i?\, \mathsf{M}_i$ where the emission of events of type $\mathtt{t}$ is enabled (i.e., $\mathtt{t} \in \kappa$). Moreover, e must be of type $\mathtt{t}$ and must reference the last updating event for $\mathtt{t}$.

4.4 Swarms

A *swarm* (of size n) is a pair (S, l) where:

- S maps indices $i \in \{1, \ldots, n\}$ to machines and their local log, i.e. $\mathsf{S}(i) = (\mathsf{M}_i, l_i)$;
- l is a *global log*, i.e., a sequence consisting of all events generated by the machines in S that preserves their order of generation. More formally:
  - Every log (global or local) $l' = e_1 \cdots e_n$ totally orders its events using a relation $<_{l'}$ defined as $e_i <_{l'} e_j$ iff $i < j$;
  - The global log l contains all and only the events of the local logs l_i (for all $i \in \{1, \ldots, n\}$) and preserves their respective ordering: $\bigcup_{i \in \{1,\ldots,n\}} l_i = l$ and $<_{l_i} \subseteq <_l$.

For brevity, we may write S instead of (S, l) (and call S a swarm) if the global log l is empty.

▶ Remark 26. The global log l of a swarm (S, l) does not exist in actual swarm implementations: it is a formalisation device which models the global ordering and non-deterministic propagation of events, as defined by swarm semantics formalised in Def. 27 below.

▶ **Definition 27** (Swarm semantics). *The behaviour of a swarm is defined by the smallest relation closed under the following rules:*

$$\frac{\mathsf{S}(i) = (\mathsf{M}_i, l_i) \quad (\mathsf{M}_i, l_i) \overset{\mathtt{t}!}{\Longrightarrow} (\mathsf{M}_i, l_i \cdot e) \quad <_{l_i} \subseteq <_l}{(\mathsf{S}, l \cdot l') \overset{\mathtt{t}!}{\longrightarrow} (\mathsf{S}[i \mapsto (\mathsf{M}_i, l_i \cdot e)], l \cdot e \cdot l')}\,[\textsc{Local}] \qquad \frac{\mathsf{S}(i) = (\mathsf{M}_i, l_i) \quad <_{l_i} \subset <_{l'} \subseteq <_l}{(\mathsf{S}, l) \xrightarrow{\tau} (\mathsf{S}[i \mapsto (\mathsf{M}_i, l')], l)}\,[\textsc{Prop}]$$

In Def. 27, rule Local formalises the emission of an event enabled at the machine at $\mathsf{S}(i)$ of the swarm (S, l). The emitted event e is appended to the local log of $\mathsf{S}(i)$ and non-deterministically merged in the global log respecting the order of events previously generated by $\mathsf{S}(i)$. Rule Prop models the asynchronous event log propagation among machines: for some machine M_i with a local log l_i (that is strictly included in the global log l), the rule non-deterministically selects a log $l' \subseteq l$ that contains l_i and transfers the events in l' to l_i while preserving their order. Note that during an execution, machines can emit events independently, and their local log may only partially reflect the "global state" in the global log (due to the non-deterministic and asynchronous propagation of events).

Def. 28 below formalises the *realisation* of a swarm protocol G, i.e., a swarm whose machines interact according to the roles G.

▶ **Definition 28** (Realisation of a swarm protocol). *A swarm (S, ϵ) of size n* realises *a protocol G with respect to a subscription σ iff for all $i \in \{1, \ldots, n\}$, (i) $\mathsf{S}(i) = (\mathsf{M}_i, \epsilon)$, and (ii) $\mathsf{M}_i = \mathsf{G}\!\downarrow^{\sigma}_{\mathtt{R}}$ for some $\mathtt{R} \in \mathsf{G}$; and (iii) M_i.UP is equal to the event types of G that are branching, joining, or looping in σ. We also say that M_i* realises *$\mathtt{R}$.*

▶ **Example 29** (Swarm protocol realisation and execution). Let E be the set of event types in Warehouse (Fig. 1) and $\sigma = \{\mathtt{T} \mapsto E, \mathtt{FL} \mapsto E \setminus \{\mathtt{partOK}\}, \mathtt{D} \mapsto E \setminus \{\mathtt{pos}\}\}$. By Def. 12, Warehouse is σ-well-formed. By Def. 28, the swarm S_W below realises Warehouse w.r.t. σ:

$$\mathsf{S}_\mathsf{W} \;=\; \{i_{\mathtt{T}_1} \mapsto (\mathsf{M}_\mathtt{T}, \epsilon), \quad i_{\mathtt{T}_2} \mapsto (\mathsf{M}_\mathtt{T}, \epsilon), \quad i_{\mathtt{FL}} \mapsto (\mathsf{M}_{\mathtt{FL}}, \epsilon), \quad i_\mathtt{D} \mapsto (\mathsf{M}_\mathtt{D}, \epsilon)\}$$

where $\mathsf{M}_\mathtt{T} = \mathsf{Warehouse}\!\downarrow^{\sigma}_{\mathtt{T}}$, $\mathsf{M}_{\mathtt{FL}} = \mathsf{Warehouse}\!\downarrow^{\sigma}_{\mathtt{FL}}$, and $\mathsf{M}_\mathtt{D} = \mathsf{Warehouse}\!\downarrow^{\sigma}_{\mathtt{D}}$ (shown in Fig. 11). Notice that the machine in $\mathsf{S}_\mathsf{W}(i_{\mathtt{T}_1})$ and $\mathsf{S}_\mathsf{W}(i_{\mathtt{T}_2})$ play the same role $\mathtt{T}$. By Def. 27, a possible execution of the swarm S_W is:

$$
\begin{array}{ll}
(\mathsf{S}_\mathsf{W}, \epsilon) \xrightarrow{\texttt{partReq!}} (\mathsf{S}_1, \mathit{partReq}_1) & \mathsf{S}_1 = \mathsf{S}_\mathsf{W}[i_{\mathsf{T}_1} \mapsto (\mathsf{M}_\mathsf{T}, \mathit{partReq}_1)] \\
\xRightarrow{\tau} (\mathsf{S}_2, \mathit{partReq}_1) & \mathsf{S}_2 = \mathsf{S}_1[i_{\mathsf{FL}_1} \mapsto (\mathsf{M}_{\mathsf{FL}}, \mathit{partReq}_1)] \\
\xRightarrow{\texttt{pos!}} (\mathsf{S}_3, \mathit{partReq}_1 \cdot \mathit{pos}_2) & \mathsf{S}_3 = \mathsf{S}_2[i_{\mathsf{FL}} \mapsto (\mathsf{M}_{\mathsf{FL}}, \mathit{partReq}_1 \cdot \mathit{pos}_2)] \\
\xRightarrow{\tau} (\mathsf{S}_4, \mathit{partReq}_1 \cdot \mathit{pos}_2) & \mathsf{S}_4 = \mathsf{S}_3[i_{\mathsf{T}_1} \mapsto (\mathsf{M}_\mathsf{T}, \mathit{partReq}_1 \cdot \mathit{pos}_2)] \\
\xRightarrow{\tau} (\mathsf{S}_5, \mathit{partReq}_1 \cdot \mathit{pos}_2) & \mathsf{S}_5 = \mathsf{S}_4[i_{\mathsf{T}_2} \mapsto (\mathsf{M}_\mathsf{T}, \mathit{partReq}_1 \cdot \mathit{pos}_2)] \\
\xRightarrow{\texttt{partOK!}} (\mathsf{S}_6, \mathit{partReq}_1 \cdot \mathit{pos}_2 \cdot \mathit{partOK}_3) & \mathsf{S}_6 = \mathsf{S}_5[i_{\mathsf{T}_1} \mapsto (\mathsf{M}_\mathsf{T}, \mathit{partReq}_1 \cdot \mathit{pos}_2 \cdot \mathit{partOK}_3)] \\
\xRightarrow{\texttt{partReq!}} (\mathsf{S}_7, \mathit{partReq}_1 \cdot \mathit{pos}_2 \cdot \mathit{partOK}_3 \cdot \mathit{partReq}_4) & \mathsf{S}_7 = \mathsf{S}_6[i_{\mathsf{T}_1} \mapsto (\mathsf{M}_\mathsf{T}, \mathit{partReq}_1 \cdot \mathit{pos}_2 \cdot \mathit{partOK}_3 \cdot \mathit{partReq}_4)] \\
\xRightarrow{\texttt{partOK!}} (\mathsf{S}_8, l) & \mathsf{S}_8 = \mathsf{S}_7[i_{\mathsf{T}_2} \mapsto (\mathsf{M}_\mathsf{T}, \mathit{partReq}_1 \cdot \mathit{pos}_2 \cdot \mathit{partOK}_5)] \\
\xRightarrow{\tau}{}^{*} (\mathsf{S}_9, l) & \mathsf{S}_9 = \mathsf{S}_8[i_{\mathsf{T}_1} \mapsto (\mathsf{M}_\mathsf{T}, l), i_{\mathsf{T}_2} \mapsto (\mathsf{M}_\mathsf{T}, l), i_{\mathsf{FL}} \mapsto (\mathsf{M}_{\mathsf{FL}}, l), i_\mathsf{D} \mapsto (\mathsf{M}_\mathsf{D}, l)]
\end{array}
$$

where, in states S_8 and S_9, the global log is $l = \mathit{partReq}_1 \cdot \mathit{pos}_2 \cdot \mathit{partOK}_3 \cdot \mathit{partReq}_4 \cdot \mathit{partOK}_5$ and its events carry the following pointers:

$$
\begin{aligned}
\mathit{partReq}_1.\mathsf{last}_\mathsf{up} &= \bot \text{ (undefined)} \\
\mathit{pos}_2.\mathsf{last}_\mathsf{up} = \mathit{partOK}_3.\mathsf{last}_\mathsf{up} &= \mathit{partReq}_4.\mathsf{last}_\mathsf{up} = \mathit{partOK}_5.\mathsf{last}_\mathsf{up} = \mathit{partReq}_1
\end{aligned}
$$

In each state of the swarm, we compute the state of a machine for its local log using $\hat{\delta}$ (Def. 24), which in turn uses the branching, joining, and looping events in l to update the mapping $\mathsf{last}_\mathsf{up}$ and only considers events carrying correct pointers. For this example, when the machines process the event $\mathit{partReq}_4$, they will update their mapping $\mathsf{last}_\mathsf{up}$ to obtain:

$$\mathsf{last}_\mathsf{up}[\texttt{partReq}, \texttt{pos}, \texttt{partOk}, \texttt{closingTime} \mapsto \mathit{partReq}_4]$$

In this updated mapping, all event types of Warehouse point to $\mathit{partReq}_4$, which means that the machines will only accept new events e such that $e.\mathsf{last}_\mathsf{up} = \mathit{partReq}_4$. Observe that, in the execution above, the event partOK_5 was emitted by the machine with ID i_{T_2} in response to the event $\mathit{partReq}_1$. Hence, $\mathit{partOK}_5.\mathsf{last}_\mathsf{up} = \mathit{partReq}_1 \neq \mathit{partReq}_4$; i.e., the event partOK_5 is causally linked to $\mathit{partReq}_1$ through branch tracking. Since the machines expect events, whose $\mathsf{last}_\mathsf{up}$ fields refer to $\mathit{partReq}_4$, the event partOK_5 is ignored by all machines after the logs propagate. Once this happens in state S_9, the machines reach the states:

$$
\begin{aligned}
\hat{\delta}(\mathsf{M}_\mathsf{T}, l) &= (\mathtt{FL}\langle\mathtt{pos}\rangle.\mathtt{T}\langle\mathtt{partOK}\rangle.\mathsf{Warehouse})\downarrow^{\sigma}_{\mathsf{T}} \\
\hat{\delta}(\mathsf{M}_\mathsf{D}, l) &= (\mathtt{T}\langle\mathtt{partOK}\rangle.\mathsf{Warehouse})\downarrow^{\sigma}_{\mathsf{D}} \\
\hat{\delta}(\mathsf{M}_{\mathsf{FL}}, l) &= (\mathtt{FL}\langle\mathtt{pos}\rangle.\mathtt{T}\langle\mathtt{partOK}\rangle.\mathsf{Warehouse})\downarrow^{\sigma}_{\mathsf{FL}}
\end{aligned}
$$

Here, the machines M_T and M_{FL} expect the forklift to emit an event of type pos, while M_D (which does not subscribe to pos in σ) is waiting for an event of type partOK that has the correct pointer and thus follows pos in the swarm protocol Warehouse. These states are consistent with each other: $(\mathtt{T}\langle\mathtt{partOK}\rangle.\mathsf{Warehouse})\downarrow^{\sigma}_{\mathsf{D}} = ((\mathtt{FL}\langle\mathtt{pos}\rangle.\mathtt{T}\langle\mathtt{partOK}\rangle.\mathsf{Warehouse})\downarrow^{\sigma}_{\mathsf{D}}$. Thus they correspond to a valid run of the Warehouse swarm protocol. ⌟

▶ Remark 30. Running the swarm of Example 29 with the semantics of [40] can lead to a state where the machines irremediably diverge due to the absence of branch tracking and the resulting lack of event causality tracking. We illustrate this situation in [25, §H].

4.5 Eventual Fidelity of Swarm Protocol Realisations

In this section, we address the problem of ensuring that a swarm behaves correctly with respect to a swarm protocol G. As our correctness criterion, we adopt the notion of *eventual fidelity* (Def. 31 below). Intuitively, a swarm is eventually faithful to a protocol G if, in every execution, once all machines have received all messages, they agree on the same *consistent view* of the execution path of G. By consistent view, we mean that machines are able to

determine which events belong to the effective execution path through G and disregard all other events that may have been emitted inconsistently. We call such a view the *effective log* (see [25, §C] for the formal definitions). Recall that the global log l of a swarm contains all events emitted by the machines. The effective log w.r.t. G, written $\mathtt{eff}(l, \mathsf{G})$, is the sublog of l that consists solely of those events of l that follow the earliest valid path through G and are causally linked by branch tracking. We also define the effective log with respect to a specific machine M, written $\mathtt{eff}(l, \mathsf{M})$, as the sublog of l containing only the events that M accepts (i.e., all the events not ignored by the log-processing function $\hat{\delta}$).

To formalise the eventual fidelity of a swarm to a protocol G, we assign to every machine in the swarm some role of G: we denote this by a mapping r from machines to the roles they play. Eventual fidelity requires that if a machine M that plays role $\mathrm{r}(\mathsf{M})$ is provided with a global log l produced by the swarm, then M accepts only those events which are in the global effective log and to which $\mathrm{r}(\mathsf{M})$ subscribes. We denote such events by $\mathtt{eff}(l, \mathsf{G}) \downarrow_{\sigma(\mathrm{r}(\mathsf{M}))}$, which is the projection of $\mathtt{eff}(l, \mathsf{G})$ onto the event types that role $\mathrm{r}(\mathsf{M})$ subscribes to in σ.

▶ **Definition 31** (Eventual fidelity). *A swarm (S, ϵ) of size n is* eventually faithful *to a protocol G with respect to a subscription σ if for all l such that $(\mathsf{S}, \epsilon) \to^* (\mathsf{S}', l)$, for all $\mathtt{R} \in \mathsf{G}$, and for all $i \in \{1...n\}$, we have $\mathtt{eff}(l, \mathsf{G}) \downarrow_{\sigma(\mathrm{r}(\mathsf{M}_i))} = \mathtt{eff}(l, \mathsf{M}_i)$ where M_i is the machine at $\mathsf{S}(i)$.*

Theorem 32 below ensures that a realisation of a swarm protocol G (i.e., a swarm consisting of machines projected from G, by Def. 28) is eventually faithful to G. *(Proof: [25, §D].)*

▶ **Theorem 32** (Eventual fidelity of swarm protocol realisations). *If a swarm S realises a σ-well-formed protocol G, then S is eventually faithful to G with respect to σ.*

▶ **Example 33** (Effective logs and eventual fidelity). Consider the swarm protocol $\mathsf{Warehouse}$ (Fig. 1), its projected machine $\mathsf{M}_\mathtt{D} = \mathsf{Warehouse}\downarrow^{\sigma}_{\mathtt{D}}$, (Fig. 11), and the log l from Example 29. By [25, Definitions 41 and 42], the effective logs for $\mathsf{Warehouse}$ and $\mathsf{M}_\mathtt{D}$ filtered from l are:

$$\begin{aligned} \mathtt{eff}(l, \mathsf{Warehouse}) &= \mathit{partReq}_1 \cdot \mathit{pos}_2 \cdot \mathit{partOK}_3 \cdot \mathit{partReq}_4 \\ \mathtt{eff}(l, \mathsf{M}_\mathtt{D}) &= \mathit{partReq}_1 \cdot \mathit{partOK}_3 \cdot \mathit{partReq}_4 \end{aligned}$$

Therefore, we have $\mathtt{eff}(l, \mathsf{Warehouse}) \downarrow_{\sigma(\mathtt{D})} = \mathtt{eff}(l, \mathsf{M}_\mathtt{D})$: hence, the effective logs for the machine and the swarm protocol align, as required by eventual fidelity (Def. 31). ⌟

5 Composing Machines and Swarms

In this section, we present an approach for realising a composition of swarm protocols $\mathsf{G}_1 \parallel \ldots \parallel \mathsf{G}_n$ by composing swarms of pre-existing machines which realise the individual protocols G_i (for $i \in \{1, \ldots, n\}$). This is crucial for enabling the reuse of pre-existing machines and their deployment in new, larger swarms (that we implement in § 6). We first formalise the composition of machines (§ 5.1); then, we introduce an automatic *adaptation* mechanism (§ 5.2) for such pre-existing machines, enabling us to compose entire swarms (Algorithm 2) while guaranteeing eventual fidelity to the composed swarm protocol (Theorem 38).

5.1 Machine Composition

Similar to swarm protocol composition (Def. 2), in Def. 34 below we define the composition of two machines M and M' as a synchronised product. Intuitively, if a transition is common to M and M', the machines synchronise and the continuation of $\mathsf{M} \parallel \mathsf{M}'$ is given by the composition of both corresponding continuations of M and M' (clause *Common Events*); otherwise, the continuation of $\mathsf{M} \parallel \mathsf{M}'$ is the composition of the continuation of either M or M' with the other machine unchanged (clauses *Events Unique*).

▶ **Definition 34** (Machine composition). *Let* $\mathsf{M} := \kappa \cdot \&_{i\in I}\mathtt{t}_i?\mathsf{M}_i$ *and* $\mathsf{M}' := \kappa' \cdot \&_{j\in J}\mathtt{t}'_j?\mathsf{M}'_j$ *be machines. Let* T *be the set of event types that occur in both* M *and* M'*. The* composition of M and M'*, written* $\mathsf{M} \mid\mid \mathsf{M}'$*, is defined as* $\mathsf{M} \mid\mid_T \mathsf{M}'$*, which in turn is coinductively defined:*

$$\mathsf{M} \mid\mid_T \mathsf{M}' \overset{co}{:=} \kappa'' \cdot \&_{k\in K}\mathtt{t}_k?\mathsf{M}_k$$

where K *is the smallest set such that for all* $i \in I, j \in J$*:*
Common Events ($\mathtt{t}_i \in T$, $\mathtt{t}_i = \mathtt{t}'_j$)**:** *There is a* $k \in K$ *with* $\mathtt{t}_k = \mathtt{t}_i$ *and* $\mathsf{M}_k = \mathsf{M}_i \mid\mid_T \mathsf{M}'_j$*.*
Events Unique to M ($\mathtt{t}_i \notin T$)**:** *There is a* $k \in K$ *with* $\mathtt{t}_k = \mathtt{t}_i$ *and* $\mathsf{M}_k = \mathsf{M}_i \mid\mid_T \mathsf{M}'$*.*
Events Unique to M′ ($\mathtt{t}'_j \notin T$)**:** *There is a* $k \in K$ *such that* $\mathtt{t}_k = \mathtt{t}'_j$ *and* $\mathsf{M}_k = \mathsf{M} \mid\mid_T \mathsf{M}'_j$*.*
Then, the emitter set κ'' *is* $\kappa'' = (\kappa \cup \kappa') \cap \{\mathtt{t}_k \mid k \in K\}$*.*

Note that by Def. 34, the acceptance transitions of $\mathsf{M} \mid\mid \mathsf{M}'$ consist of the acceptance transitions of both M and M'; common transitions are synchronised via the set T, hence they are only enabled in a state if both M and M' can take them. Further, an event type $\mathtt{t}$ belongs to the emitter set of $\mathsf{M} \mid\mid \mathsf{M}'$ if and only if (i) $\mathtt{t}$ belongs to the emitter set of either M or M', and (ii) either $\mathtt{t} \in T$ and both machines accept $\mathtt{t}$, or $\mathtt{t} \notin T$ and one of the machines accepts $\mathtt{t}$. This means a machine either implements an interfacing role and thus only emits events in T or it is non-interfacing and thus only emits events not in T. Also note that, as with swarm protocol composition (Def. 2), we do not introduce new syntax for composed machines: composed machines are treated as ordinary machines, to maintain backward compatibility with existing tooling (that we extend in § 6).

▶ **Example 35** (Relating swarm protocol composition, projection, and machine composition). Consider the swarm protocols Warehouse and Factory in Figures 1 and 4, and the subscription σ from Example 22. Also consider the projected machines $\mathsf{Warehouse}\downarrow^{\sigma}_{\mathtt{D}}$ in Fig. 11, and $\mathsf{Factory}\downarrow^{\sigma}_{\mathtt{D}} = \mathtt{partReq}?\cdot\mathtt{partOK}?\cdot\mathbf{0}$. If we compose these two machines (using Def. 34), then we derive the machine denoted as $\mathsf{G}\downarrow^{\sigma}_{\mathtt{D}}$ in Fig. 10, and therefore we have:

$$\mathsf{Factory}\downarrow^{\sigma}_{\mathtt{D}} \mid\mid \mathsf{Warehouse}\downarrow^{\sigma}_{\mathtt{D}} = (\mathsf{Factory} \mid\mid \mathsf{Warehouse})\downarrow^{\sigma}_{\mathtt{D}}$$

Note that this holds because, in σ, the roles of both projected machines subscribe to the interfacing event types that synchronise the composed swarm protocols (by Def. 2). ⌟

5.2 Machine Adaptation

In Def. 36 we introduce a function $\mathcal{A}$ that *adapts* a single machine M, projected from a swarm protocol G_k, so that it behaves correctly when used in a swarm with other adapted machines that were projected from swarm protocols $\mathsf{G}_1, \ldots, \mathsf{G}_n$. Intuitively, the adaptation function $\mathcal{A}$ restructures the transition graph of M in two steps (described in more detail below): (1) it adds automatically generated acceptance transitions to wait for extra synchronisation event types required by a composed subscription σ, and (2) it disables transitions whose continuations are incompatible with the other projections. Notably, $\mathcal{A}$ does not rewrite the logic of M: the adapted machine looks different from M (see Example 37 below) but is still just the original automaton with a thin structural wrapper that delays progress until required synchronisation events are emitted by other machines, or disables incompatible paths.

▶ **Definition 36** (Machine adaptation for swarm composition). *Let* $\mathsf{G}_1, \ldots, \mathsf{G}_n$ *be swarm protocols with subscriptions* $\sigma_1, \ldots, \sigma_n$*. Let* M *be a machine that, for some* $k \in \{1, \ldots, n\}$*, realises role* $\mathtt{R}$ *in the swarm protocol* G_k *for subscription* σ_k*. Let* σ *be a well-formed subscription for* $\mathsf{G}_1 \mid\mid \ldots \mid\mid \mathsf{G}_n$ *(obtained via Algorithm 1). Then, we define the* machine adaptation $\mathcal{A}$ *as:*

$$\mathcal{A}\big(\mathsf{M}, (\mathsf{G}_i)_{i\in\{1,\ldots,n\}}, \mathtt{R}, k, \sigma\big) = (\mathsf{G}_1\downarrow^{\sigma}_{\mathtt{R}} \mid\mid \ldots \mid\mid (\mathsf{M} \mid\mid \mathsf{G}_k\downarrow^{\sigma}_{\mathtt{R}}) \mid\mid \ldots \mid\mid \mathsf{G}_n\downarrow^{\sigma}_{\mathtt{R}})\,.$$

Algorithm 2 Swarm composition (shortened version of [25, §E, Algorithm 3]).

Input:
- $\mathsf{G}_1, \ldots, \mathsf{G}_n$ composable protocols, which are respectively realised by swarms $\mathsf{S}_1, \ldots, \mathsf{S}_n$ for the subscriptions $\sigma_1, \ldots, \sigma_n$.
- Mappings $\mathrm{r}_1, \ldots, \mathrm{r}_n$ that respectively map machines to the role they implement, i.e., $\mathrm{r}_k(\mathsf{S}_k(i))$ is the role realised by the machine at $\mathsf{S}_k(i)$.

Output: A swarm S that realises $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$.

1: Apply Algorithm 1 to obtain a well-formed subscription σ for $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$.
2: For each swarm S_k, for each machine $\mathsf{S}_k(i)$:
adapt $\mathsf{S}_k(i)$ into $\mathsf{M}' = \mathcal{A}\big(\mathsf{M}, (\mathsf{G}_i)_{i\in\{1,\ldots,n\}}, \mathrm{r}_k(\mathsf{S}_k(i)), k, \sigma\big)$ according to Def. 36.
3: Return a swarm built up from the machines adapted in the previous step.

We also set $\mathcal{A}\big(\mathsf{M}, (\mathsf{G}_i)_{i\in\{1,\ldots,n\}}, \mathtt{R}, k, \sigma\big).\mathrm{UP}$ *as the set of branching, joining, or looping event types selected by Algorithm 1: this overapproximates the set of branching/joining/looping event types of* $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$.

To better see the effect of the adaptation $\mathcal{A}$ in Def. 36, observe that by Algorithm 1 we have $\sigma_k \subseteq \sigma$, hence the subscriptions σ and σ_k may differ; consequently, $\mathsf{G}_k\!\downarrow_{\mathtt{R}}^{\sigma_k}$ and $\mathsf{G}_k\!\downarrow_{\mathtt{R}}^{\sigma}$ may not coincide. Additionally, $\mathtt{R}$ might be an interfacing role; therefore, it might appear in other protocols besides G_k. Hence, the input machine M must be adapted to play role $\mathtt{R}$ while conforming to *all* protocols where $\mathtt{R}$ appears. Consequently, the adaption of M involves the following two steps (that were outlined above):

1. M is composed with $\mathsf{G}_k\!\downarrow_{\mathtt{R}}^{\sigma}$ to adapt it to the behaviour expected by the subscription σ (which may be larger than σ_k). This adaptation may force M to await and accept additional events types, i.e., those in $\sigma(\mathtt{R})$ that are not in $\sigma_k(\mathtt{R})$.
2. M is also composed with the projections obtained from the remaining protocols of the composition. This step may prune some transitions of M that are absent in the other protocols and add necessary transitions from the other protocols to ensure correct synchronisation in the composed swarm.

We extend the adaptation mechanism in Def. 36 to whole swarms: Algorithm 2 takes a set of swarms that respectively realise protocols $\mathsf{G}_1, \ldots, \mathsf{G}_n$, and automatically generates a swarm that realises $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$. Specifically, Algorithm 2 uses Algorithm 1 to find a suitable subscription for the composition, and then it applies the adaptation function in Def. 36.[10] *(For a more detailed version of Algorithm 2, see [25, §E, Algorithm 3].)*

▶ **Example 37** (Composing swarms by adapting their machines). Recall the swarm S_W whose machines are projected from the swarm protocol Warehouse and subscription σ in Example 29. In order to compose S_W with another swarm S_F projected from Factory using Algorithm 2, we apply the adaptation in Def. 36 to all machines in both S_W and S_F, and we obtain a swarm $\mathsf{S}_{\mathsf{W}\,||\,\mathsf{F}}$ that realises the composed protocol $\mathsf{Warehouse} \,||\, \mathsf{Factory}$. We illustrate this on the forklift machine $\mathsf{M}_\mathtt{FL} = \mathsf{Warehouse}\!\downarrow_\mathtt{FL}^{\sigma}$. We first apply Algorithm 1 (as in Example 15) to obtain a subscription σ' such that $\mathsf{Warehouse} \,||\, \mathsf{Factory}$ is σ'-well-formed, and which contains the initial σ and the interfacing event types needed for synchronising the machines that implement Warehouse and Factory. By Def. 36, the machine adaptation of $\mathsf{M}_\mathtt{FL}$ is:

$$\mathcal{A}(\mathsf{M}_\mathtt{FL}, (\mathsf{Warehouse}, \mathsf{Factory}), \mathtt{FL}, 1, \sigma') \;=\; \Big(\mathsf{M}_\mathtt{FL} \;||\; \mathsf{Warehouse}\!\downarrow_\mathtt{FL}^{\sigma'}\Big) \;||\; \mathsf{Factory}\!\downarrow_\mathtt{FL}^{\sigma'}$$

[10] The complexity of Algorithm 2 is exponential in the size of the projections of the input protocols; instead, a "monolithic" adaptation (i.e., composing input machines with a projection of the whole composition $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$) would be exponential in the input protocols. If all roles subscribe to all event types, then these complexities are equal, because protocols and projections have the same sizes. However, in practice, the subscriptions (thus, the projections) are usually much smaller, hence Algorithm 2 is more efficient: e.g., in our experiments in § 6.2, the roles only need to subscribe to ∼30% of the event types.

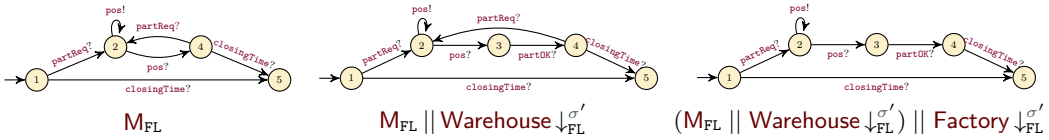

Figure 12 How Def. 36 adapts the machine $M_{FL} = Warehouse\downarrow^{\sigma}_{FL}$ to $Warehouse \,||\, Factory$.

where "1" is the index of Warehouse in the tuple (Warehouse, Factory). The stages of the adaptation are depicted in Fig. 12, where the state numbers show the relationship between the original machine and its adaptations. The original M_{FL} is adapted with the projection of its original swarm protocol (Warehouse) using σ' (computed with Algorithm 1) to obtain the machine $M_{FL} \,||\, Warehouse\downarrow^{\sigma'}_{FL}$: observe that this adapted machine has an additional state 3 which forces the machine to wait for an event of type `partOK` after `pos`, between the original states 2 and 4. When $M_{FL} \,||\, Warehouse\downarrow^{\sigma'}_{FL}$ is further composed with $Factory\downarrow^{\sigma'}_{FL}$, the adapted machine loses the possibility in state 4 of awaiting another event of type `partReq` and looping to state 2: hence, in state 4, the adapted machine can only await and process an event of type `closingTime`. Using Algorithm 2, this final machine is added to the adapted swarm $S_{W\,||\,F}$. Then, Algorithm 2 repeats this adaptation for every machine in S_W and S_F. ⌟

We can now state the correctness of our swarm composition procedure. *(Proof: [25, §F].)*

▶ **Theorem 38** (Correctness of swarm adaptation and composition). *Let* S *be a swarm obtained by applying Algorithm 2 to swarms* $S_1, \ldots, S_n$*, protocols* $G_1, \ldots, G_n$ *and subscriptions* $\sigma_1, \ldots, \sigma_n$*. Then,* S *is eventually faithful to* $G_1 \,||\, \ldots \,||\, G_n$*.*

The key insight in the proof of Theorem 38 is that the subscription σ computed by Algorithm 1 includes all the events needed for the synchronization of the composed swarm. This ensures that each adapted machine is a correct projection of the composed swarm protocol. Therefore, a machine adapted from the input swarm S_i will execute without desynchronising and diverging from other machines adapted from another input swarm S_j (for $i, j \in \{1 \ldots n\}$ such that $i \neq j$).

6 Implementation and Experiments

We now present our implementation of the theory in §§ 3–5. In § 6.1 we discuss how we extend the open source Actyx toolkit [3] to support branch-tracking machine semantics and machine adaptation (which are key elements of our compositional approach), and to statically verify our new, compositional well-formedness of swarm protocols. Then, in § 6.2 we benchmark different strategies for computing the subscriptions of composed swarms.

Our implementation is available in the companion artifact of this paper. It enables a worfklow for constructing complex swarms from simpler ones, where a developer can (1) assemble a library of swarm protocols and corresponding machine implementations, (2) compose a set of protocols to model a desired scenario, and (3) automatically adapt the machine implementations (via Algorithms 1 and 2) so they follow the composed protocol.

6.1 Branch Tracking, Adaptations, and New Verification Facilities

Branch Tracking and Adaptation. The `machine-runner` library [2] of Actyx features a TypeScript API to program *machine implementations*. E.g., Fig. 3 shows the implementation of a machine M (as formalised in Def. 17) for the role `D` of our Warehouse use case.

The `machine-runner` library also helps instantiating and running machine implementations in a swarm using the Actyx middleware [1] which implements the log propagation mechanisms described in § 4.4. Let us discuss our extension of `machine-runner`.

To support our new branch-tracking semantics (Def. 24), *(1)* we added a pointer $e.\mathsf{last}_{\mathsf{up}}$ ("previous updating event") to each event e, and *(2)* we modified `machine-runner` to produce and use $e.\mathsf{last}_{\mathsf{up}}$. We also added new methods to automatically adapt machine implementations by Def. 36. This allows developers to re-use and deploy existing machine implementations in composed swarms. For example, to adapt the machine implementation `door` of role `D` in Fig. 3 to the composed swarm Warehouse || Factory, a programmer can simply invoke:

```
adaptedDoor = adaptMachine('D', listOfProtocols, 0, subscriptions, [door, s0])
```

The parameters to `adaptMachine` are the machine role `D`, a list of the swarm protocols in the composition, the index of Warehouse in that list, a well-formed (and automatically generated) subscription for the composition, and the original `door` machine with its initial state.

New Static Verification Facilities. The `machine-check` library of Actyx can statically verify (i) the well-formedness (as defined in [40]) of a protocol G w.r.t. subscription σ, and (ii) whether the machine implementation of a role `R` of G emits and accepts events as specified by the projection of $\mathsf{G}\!\downarrow_{\mathtt{R}}^{\sigma}$. We extended `machine-check` to verify our new well-formedness of protocol compositions (Definitions 2 and 12) and generate well-formed subscriptions (Algorithm 1). We also updated the `machine-check` projection code (which was based on [40]) to conform to our revised projection in Def. 21. This enables (1) the static verification of machine implementations in composed swarm protocols, and (2) the automatic adaptation of previously-implemented machines to work correctly in a composed swarm.

6.2 Experiments on Compositional Subscription Computation

A key element for the correct composition of protocols and swarms is the availability of a subscription σ suitable for composition (required by Def. 31 and Theorem 32). We analyse the performance and trade-offs of Algorithm 1, which computes such subscriptions (see § 5).

Algorithm 1 may compute a subscription larger than necessary, causing non-optimal performance in the deployed swarm. Therefore, we compare Algorithm 1 with a naive algorithm (available in [25, §G]) that computes the "exact" subscription by directly applying Def. 12. More precisely, this algorithm takes a set $\{\mathsf{G}_1, \ldots, \mathsf{G}_n\}$ of composable protocols and subscriptions $\{\sigma_1, \ldots, \sigma_n\}$, expands the composed protocol using Def. 2, and applies Def. 12 to produce the smallest $\sigma \supseteq \sigma_1, \ldots, \sigma_n$ for which $\mathsf{G} = \mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ is σ-well-formed. As mentioned in § 3.3, the expansion of the n composed protocol makes the time complexity of this "exact" algorithm exponential in n – whereas Algorithm 1 is polynomial in n.

Benchmark Selection and Experimental Setup. We created a benchmark consisting of 454 sets of randomly-generated swarm protocols. Each set consists of n swarm protocols $\mathsf{G}_1, \ldots, \mathsf{G}_n$ (with $n \in \{1, \ldots, 10\}$) with up to 9 roles each; each role emits up to 9 event types. To cover a broad range of possible practical cases, the random generation produces variety of branching and looping patterns; moreover, protocols G_i and G_{i+1} share an interfacing role, for all $i \in \{1, \ldots, n-1\}$. To avoid behaviour restrictions (Example 4), all protocols follow a pattern: two interfacing event types $\mathtt{t}_1$ and $\mathtt{t}_2$ may be separated by a random number of choices with non-interfacing event types, but $\mathtt{t}_1$ and $\mathtt{t}_2$ are always enabled in the same order.

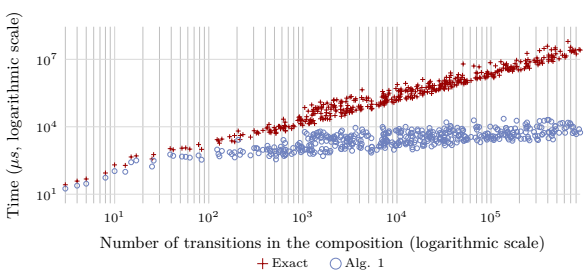

Figure 13 Average execution times for generating well-formed subscriptions, w.r.t. the number of transitions in the composition. (Intel Xeon Gold 6226R, 32GB RAM, AlmaLinux 9.5; averages produced by Criterion 0.7.0 [23] with 50+ repetitions, including outliers; negligible deviations).

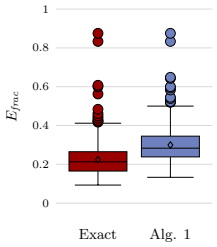

Figure 14 Boxplot showing the average fraction of events in the computed subscriptions. (Lower is better.)

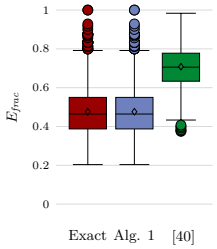

Figure 15 Comparison of subscription sizes obtained using our definition of well-formedness and that of [40].

Our experiments invoke both Algorithm 1 and our "exact" algorithm on empty input subscriptions, to obtain the smallest possible output subscription for each protocol composition $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$. We measured the execution times using Criterion [23]; for each randomly-generated set of protocols, Criterion invoked Algorithm 1 and the "exact" algorithm at least 50 times after 3 seconds of warm-up, reporting averages, medians, and standard deviations.

Results. Fig. 13 plots the execution time measured in our experiments against the number of transitions in the composition $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$; note that the number of transitions grows exponentially in n. As expected, Algorithm 1 is more scalable than the "exact" algorithm, which suffers from the exponential blow-up of the composition: e.g., generating a subscription for a composition of protocols with $\sim 10^5$ transitions takes $\sim$0.01 seconds for Algorithm 1 and $\sim$10 seconds for the "exact" algorithm. Notably, the trends in Fig. 13 persist if we discount the time needed to expand the composition of protocols before invoking the "exact" algorithm – which spends much more time applying Def. 12 on the expanded composition.

Fig. 14 compares the accuracy of the two algorithms, measured as the fraction of event types involved in the generated subscription σ. Specifically, E_{frac} is the fraction of all event types subscribed to by a role in σ on average across all roles. For instance, $E_{frac} = 0.2$ means that, on average, a role subscribes to one out of five event types, while $E_{frac} = 1.0$ means that all roles subscribe to every event type. In general, a lower E_{frac} means less subscriptions and a higher degree of concurrency, leading to a swarm with smaller and more efficient machines. The plot shows that the "exact" algorithm requires roles to subscribe to $\sim$22.4% of all event types in the input protocols. Algorithm 1 requires subscribing to $\sim$29.9% of the event types, which is quite close to the "exact" algorithm. The difference occurs because Algorithm 1 may not see that parts of the input protocols become unreachable in the composition, and thus, may produce redundant subscriptions to some updating event types; also, Algorithm 1 adds interfacing event types to subscriptions (even when not required by well-formedness). This is necessary to ensure correct machine adaptation without expanding the composed protocol.

Comparing Subscription Sizes with Those of [40]. In § 2.2 we mentioned that the well-formedness definition of [40] may lead to larger subscriptions than necessary, compared to our Def. 12 and Algorithm 1. We now quantify this claim experimentally. Fig. 15 compares the sizes of minimal well-formed subscriptions according to our Def. 12 and [40] for 2093 randomly generated swarm protocols. Since [40] does not support concurrent events, these 2093 protocols are sequential (Def. 14), hence the minimal well-formed subscriptions for [40] and Def. 12 are larger than those in Fig. 14. Fig. 15 shows that our Def. 12 and Algorithm 1

require roles to subscribe to ∼47.6% of all event types in the input protocols on average, whereas [40] requires ∼70.8%.[11] Fig. 15 also shows that, for sequential protocols, Algorithm 1 yields the same output subscriptions as the "exact" algorithm: this is because the protocols contain no interfacing event types, nor behaviour restrictions like those in Example 4.

7 Related Work

This work embraces the research path taken in [40], advocating *local-first cooperation* [38, 37] in the design, analysis, and development of distributed applications modelled as swarms of machines, using swarm protocols as a specification language. We tackle the problem of compositional design and verification of swarms – which was left open in [40] and, to the best of our knowledge, is not addressed elsewhere. The importance of modularisation for managing software complexity has been well established since early work [44]: separate parts of a system should be composed via well-defined *interfaces* that hide implementation details between components. We accomplish this using *interfacing roles* (Def. 1) whose emitted events act as the glue that binds different swarm protocols and swarms into larger ones.

Technical Differences w.r.t. [40]. Besides introducing our new compositionality results, in this work we simplify the theory of [40] aligning it to the common usages of the open source Actyx toolkit [3], without reducing the expressiveness of the model:

- The grammar in Def. 17 is as in [40], except that we do not explicitly model the "commands" a machine performs when emitting events: we assume a 1:1 correspondence between events and commands, leaving the latter implicit. Moreover, our transitions range over single event types instead of finite non-empty sequences of event types.
- In Def. 27, our rule Local simplifies the corresponding rule in [40]: since we have a 1:1 correspondence between events and commands, the shuffling operator of [40] is unnecessary. Also, our rule Global simplifies the corresponding rule in [40].[12]

A key challenge of our compositional approach, compared to the "monolithic" approach of [40], is that in the monolithic setting, the eventual fidelity of a machine M is only ensured if M knows and accepts the *guard event* for every command that precedes M's event emissions – and this, in our one-event-per-command setting, would mean that M must know and accept all events that precede M's emissions. However, in our composed swarms, a machine M projected from one swarm protocol may encounter events produced by other machines and protocols that M did not originally anticipate. To address this, our revised machine semantics (§ 4.3) tracks event causality more precisely, requiring a machine M that plays role R to accept only those events that can affect R's behaviour. This is achieved via *updating event types* and our improved swarm semantics (Def. 27): each event carries minimal causal information, i.e., a pointer to the last relevant updating event observed by the emitting machine. This

[11] In Fig. 15, the boxplots of the "exact" algorithm and Algorithm 1 contain two outliers with $E_{frac} = 1$, greater than all values in the boxplot for [40]. These outliers are very small protocols with a choice where a role R only appears in one of the branches. In this case, our Def. 12 requires R to subscribe to event types in all branches of the choice, whereas [40] only requires R to subscribe to the branch where R participates. Hence, our Def. 12 requires additional subscriptions for such cases – but it also provides a stronger guarantee w.r.t. [40]: a machine playing role R can determine if the swarm is following a branch of the protocol where R is not involved.

[12] More specifically, the closure property imposed on the sub-log l' in [40] is not necessary in this work. This is because we do not need to ensure that for each event e in the global log l emitted by a machine of a swarm S, say S(j), l includes each event e' that precedes e in the local log of S(j).

allows a machine to decide whether to accept or ignore an event. The main result of [40] is that any swarm S that realises a well-formed G is eventually faithful to G; we generalise this result to composed swarms and protocols in §§ 4.5 and 5.2.

Global Types and Behavioural Types. Swarm protocols are inspired by *global types*, a.k.a. *choreographies* [32, 33]; they describe multiparty interaction pattern from a global point of view, and they can be projected onto roles to obtain local specifications, against which one can check the correctness of implementations. The literature on global types / choreographies includes compositional approaches: e.g., partial choreographies that can be composed ensuring deadlock-freedom [42], and refinements to replace part of a choreography with a more complex one [20]. However, the highly dynamic and non-deterministic semantics of swarms is significantly different from the semantics in global types literature. When compared to the literature on behavioural types [26, 34, 5], our approach and [40] significantly depart from the usual assumptions. Standard behavioural typing approaches ensure all interacting components are aware of the current state of the distributed computation. In contrast, we aim at *eventual* consistency [14]: i.e., machines can make transient discordant choices that are reconciled as the logs propagate in the swarm. This is akin to data replication methods, where there is a trade-off between availability, consistency, and partition tolerance [28]. Several approaches have emerged to balance this trade-off, including conflict-free replicated data types [47], cloud types [15], consistency contracts [49], invariants [30, 36, 7], linearizability [51], and operational models for applications such as GSP [16, 29]; a key difference in our approach lies in how we achieve eventual consistency, and how we introduce compositional mechanisms. Moreover, in standard behavioural type sytems, the variability in the number of participants has been addressed via *parametricity*, e.g. in multiparty session types [52, 21, 22, 17, 35]: this requires explicit handling of the parameters of the protocol. Instead, swarm protocols do not restrict the number of instances playing a given role.

In behavioural types literature, most compositional approaches require component protocols to be aware that they are part of a composition: e.g. hybrid types [27] distinguish inter-component vs. intra-component interactions; other approaches allow composition via open endpoints [42, 50]. An approach closer to ours is *Participants-as-Interfaces* (PaI) [9, 11, 12, 10, 13]. Like us, PaI allows viewing a closed system as an open one – but PaI's interfaces use complementary roles as *forwarders* that connect protocols, whereas our interfacing roles simply play their roles and do not act as forwarders.

Choreographic Programming Languages [41]. differ from swarm protocols in both goals and approach. Choreographic programs provide a global *implementation* of communicating roles via point-to-point messages, with notions of compositionality inspired by programming languages theory (e.g., higher-order choreographic programs [48, 19]). Swarm protocols and projected machines, by contrast, serve as specifications only: machines are implemented and checked separately, e.g., using the Actyx toolkit. Moreover, machines operate under asynchronous, subscription-based event propagation, with possible event reordering and machine replication. These goals and semantics are significantly different from choreographic programming languages, and thus, they lead to a different notion of composition.

8 Conclusion and Future Work

We have presented novel results on the compositional verification and implementation of swarms and swarm protocols. Our results enable a compositional swarm design process, as well as the reuse of existing swarm systems and their extension with new functionality.

We believe our novel compositionality results can be applied to other local-first distributed systems. We have implemented our results in a custom version of the open source Actyx toolkit for swarm application development [3], thus enabling the compositional design and implementation of swarms in practice. We have also streamlined the swarm theory presented in [40], while aligning it with the observed use of Actyx tools in the real world.

Future Work. Swarms and machines have some similarity with *actor systems* [31, 4] (asynchronous communication, replication) but differ significantly in their semantics (out-of-order event propagation). We are not aware of specific work on the composition of actor systems – although the communicating finite-state machines used in [11, 12, 13] have analogies with actors (message buffers can be seen as a form of mailbox), hence their approach to compositionality may be applicable to actors. This is a valuable area of study, and we will investigate whether our compositionality results can be applied to actor systems.

Currently, our method of composing protocols and swarms (Algorithms 1 and 2) and our correctness result (Theorem 38) are limited to compositions of *sequential* protocols without internal concurrency: i.e., we support compositions of the form $\mathsf{G}_1 \,||\, \ldots \,||\, \mathsf{G}_n$ where each G_i is a sequential protocol (by Def. 14) – and this allows us to reuse and compose all swarm protocols that are well-formed according to [40], and all machines projected from such protocols. However, we do not currently support arbitrary compositions of concurrent protocols, like e.g. $(\mathsf{G}_1 \,||\, \mathsf{G}_2) \,||\, (\mathsf{G}_3 \,||\, \mathsf{G}_4)$. To address this limitation, we plan to build upon our results and generalise them to design a full-fledged *module system for swarm development*. We will also study better compositional methods for the computation of subscriptions and the minimisation of swarm machines, in order to improve the precision of our Algorithm 1 (measured in § 6.2) without losing scalability for larger swarms.

References

1 Actyx AG. Actyx decentralized event database, streaming and processing engine, 2024. URL: `https://github.com/Actyx/Actyx`.

2 Actyx AG. @actyx/machine-runner library, 2024. accessed 13-12-2024. URL: `https://www.npmjs.com/package/@actyx/machine-runner`.

3 Actyx AG. Actyx developer website. `https://developer.actyx.com`, 2024. [Online; accessed on 29 October 2024].

4 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, December 1986. `doi:10.7551/mitpress/1086.001.0001`.

5 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. `doi:10.1561/2500000031`.

6 Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. *Distributed Systems*, 53, 1993.

7 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno M. Preguiça. IPA: invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, 2018. `doi:10.14778/3297753.3297760`.

8 Stephanie Balzer, Marco Carbone, Roland Kuhn, and Peter Thiemann. Next generation protocols for heterogeneous systems (dagstuhl seminar 24051). *Dagstuhl Reports*, 14(1):108–129, 2024. `doi:10.4230/DAGREP.14.1.108`.

9 Franco Barbanera, Ugo de'Liguoro, and Rolf Hennicker. Connecting open systems of communicating finite state machines. *J. Log. Algebraic Methods Program.*, 109, 2019. `doi:10.1016/j.jlamp.2019.07.004`.

10 Franco Barbanera, Mariangiola Dezani-Ciancaglini, Lorenzo Gheri, and Nobuko Yoshida. Multicompatibility for multiparty-session composition. In *Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming*, pages 1–15, 2023. doi:10.1145/3610612.3610614.

11 Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese, and Emilio Tuosto. Composition and decomposition of multiparty sessions. *J. Log. Algebraic Methods Program.*, 119:100620, 2021. doi:10.1016/j.jlamp.2020.100620.

12 Franco Barbanera, Ivan Lanese, and Emilio Tuosto. On composing communicating systems. In Clément Aubert, Cinzia Di Giusto, Larisa Safina, and Alceste Scalas, editors, *Proceedings 15th Interaction and Concurrency Experience, ICE 2022, Lucca, Italy, 17th June 2022*, volume 365 of *EPTCS*, pages 53–68, 2022. doi:10.4204/EPTCS.365.4.

13 Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Composition of synchronous communicating systems. *J. Log. Algebraic Methods Program.*, 135:100890, 2023. doi:10.1016/j.jlamp.2023.100890.

14 Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1–2):1–150, October 2014. doi:10.1561/2500000011.

15 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 283–307, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-31057-7_14.

16 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 568–590, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2015.568.

17 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019. doi:10.1145/3290342.

18 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

19 Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. Modular Compilation for Higher-Order Functional Choreographies. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:37, 2023. doi:10.4230/LIPIcs.ECOOP.2023.7.

20 Ugo de'Liguoro, Hernán C. Melgratti, and Emilio Tuosto. Towards refinable choreographies. *J. Log. Algebraic Methods Program.*, 127:100776, 2022. doi:10.1016/j.jlamp.2022.100776.

21 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 435–446. ACM, 2011. doi:10.1145/1926385.1926435.

22 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

23 Criterion Developers. Criterion documentation, 2025. Accessed 18-02-2026. URL: https://docs.rs/criterion/0.7.0/criterion.

24 Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. doi:10.1145/857076.857078.

25 Florian Furbach, Lucas Clorius, Roland Kuhn, Hernán Melgratti, Alceste Scalas, and Emilio Tuosto. Compositional design, implementation, and verification of swarms (technical report), 2026. doi:10.48550/arXiv.2604.16097.

26 Simon Gay and Antonio Ravara, editors. *Behavioural Types: from Theory to Tools*. Automation, Control and Robotics. River, 2009.

27 Lorenzo Gheri and Nobuko Yoshida. Hybrid multiparty session types: Compositionality for protocol specification through endpoint projection. *Proc. ACM Program. Lang.*, 7(OOPSLA), April 2023. doi:10.1145/3586031.

28 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

29 Alexey Gotsman and Sebastian Burckhardt. Consistency Models with Global Operation Sequencing and their Composition. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2017.23.

30 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause i'm strong enough: reasoning about consistency choices in distributed systems. In *POPL 2016*, pages 371–384, 2016. doi:10.1145/2837614.2837625.

31 Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. William Kaufmann. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.

32 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, pages 273–284, 2008. doi:10.1145/1328438.1328472.

33 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM (JACM)*, 63(1):1–67, 2016. doi:10.1145/2827695.

34 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

35 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In Peter Müller, editor, *Programming Languages and Systems*, pages 251–279, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-44914-8_10.

36 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018. doi:10.1145/3276534.

37 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In Hidehiko Masuhara and Tomas Petricek, editors, *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, Onward! 2019, pages 154–178, New York, NY, USA, 2019. ACM. doi:10.1145/3359591.3359737.

38 Roland Kuhn. Local-first cooperation: Autonomy at the edge, secured by crypto, 100% available. https://www.infoq.com/articles/local-first-cooperation/, 2021. [Online; accessed on 29 October 2024].

39 Roland Kuhn and Alan Darmasaputra. Behaviorally typed state machines in typescript for heterogeneous swarms. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, ISSTA 2023, pages 1475–1478, New York, NY, USA, 2023. ACM. doi:10.1145/3597926.3604917.

40 Roland Kuhn, Hernán C. Melgratti, and Emilio Tuosto. Behavioural types for local-first software. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ECOOP.2023.15.

41 Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. doi:10.1017/9781108981491.

42 Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, Lecture Notes in Computer Science, pages 425–439. Springer, 2013. doi:10.1007/978-3-642-40184-8_30.

43 D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, SE-9(3):240–247, 1983. doi:10.1109/TSE.1983.236733.

44 David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. doi:10.1145/361598.361623.

45 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

46 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. doi:10.1145/98163.98167.

47 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS 2011*, pages 386–400, 2011. doi:10.1007/978-3-642-24550-3_29.

48 Gan Shen, Shun Kashiwa, and Lindsey Kuper. HasChor: Functional Choreographic Programming for All. *Proceedings of the ACM on Programming Languages*, 7(ICFP):541–565, 2023. doi:10.1145/3607849.

49 K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 413–424. ACM, ACM, 2015. doi:10.1145/2737924.2737981.

50 Claude Stolze, Marino Miculan, and Pietro Di Gianantonio. Composable partial multiparty session types for open systems. *Software and Systems Modeling*, 22(2):473–494, 2023. doi:10.1007/s10270-022-01040-x.

51 Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 980–993, 2019. doi:10.1145/3314221.3314617.

52 Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010. doi:10.1007/978-3-642-12032-9_10.