







Formal Foundations for Reowolf: Multi-Party Sessions via Synchronous Protocol Programming[★]

Christopher A. Esterhuyse ¹, Benjamin Lion² ,
Hans-Dieter A. Hiep^{3,4} , and Farhad Arbab⁵ 

¹ University of Amsterdam, Amsterdam, The Netherlands ✉ c.a.esterhuyse@uva.nl

² INRIA, Rennes, France benjamin.lion@inria.fr

³ NLNet Foundation, Amsterdam, The Netherlands hdh@nlnet.nl

⁴ Leiden Institute of Advanced Computer Science (LIACS), Leiden, The Netherlands

⁵ CWI, Amsterdam, The Netherlands f.arbab@cwi.nl

Abstract. The Reowolf project developed *connectors* as a replacement of two-party network sockets for multi-party communication in next-generation internet applications. Users control connectors via protocols in the bespoke *protocol description language* (PDL), which is based on synchronous languages such as Reo and Esterel. The novelty lies in the emphasis on dynamism: users refine protocols throughout their execution. We formalise the semantics of PDL, distinguishing dual notions of protocol behaviour: *accepted* behaviour is highly (de)compositional and specifies *what* communication is allowed, while *constructed* behaviour arises from protocol execution and accounts for *how* execution steps interdepend and interleave via messages sent and received. Toward machine-checking the correctness of the connector runtime reference implementation, we specify the API and correctness criteria of PDL runtime systems.

Keywords: synchronous languages · formal specification · protocol · communication · composition · message passing · distributed computing

1 Introduction

Thanks to decades of fruitful research, synchronous coordination languages like Reo [1,3] and Esterel [7,21] enable the specification of complex, multi-party communication behaviour using synchronous protocols. These protocols specify essential ordering and data-dependencies of messages and computations, but stop short of fixing the low-level implementation details. Consequently, protocols are abstract and compositional, affording powerful systematic analysis and verification against high-level properties such as fairness and deadlock freedom. Such protocols are often applied in the coordination of communications between software components. Extensive literature explores this usage, *e.g.*, compiling protocols expressed in Reo [14,32,33,34] and Esterel [16,17,18,45] into low-level ‘glue code’ which mediates communications between software components. The resulting applications exhibit complex coordinated behaviour, but they can be systematically reasoned about via their protocols, *e.g.*, to verify properties. Also, this approach decouples components, making them easier to maintain and reuse.

[★] This is a preprint version of the article. The final publication is available at Springer via https://doi.org/10.1007/978-3-031-95589-1_1.

The *Reowolf project* investigated the application of Reo-like synchronous protocols to the coordination of communications on the Internet [19], *e.g.*, to express delicate multi-party transactions as composable synchronous interactions. In this new context, where distributed peers come and go, it is impractical to collect and compile all peers’ requirements as a single *session protocol* in overview before the session begins. Instead, the idea is that communication behaviour unfolds, one synchronous round at a time, in between changes to the protocol. Peers can come and go, and adjust the session protocol to reflect their changing requirements. This idea is implemented in the *connector runtime* middleware system, whose API is the *connector*. Like BSD-style network sockets, each connector maintains the user-facing abstraction of *session*, by automating the underlying control communications and resource management. Unlike sockets, connectors are extensively programmable. Users communicate entirely via Reowolf’s *Protocol Description Language* (PDL), which is based on Reo and other high-level synchronous languages, but was co-designed with connectors. The Reowolf project contributed the design, Rust implementation, and benchmarking of the connector runtime, as detailed in extensive technical documentation [20]. To frame our present contributions, we overview the usage of connectors in Section 2: we lay out the connector API and user requirements as Invariants RI_1 to RI_4 . These characterise the users’ control over communications; users can rely on the realisation of their piece-meal specifications of stateful, synchronous data flows.

In this article, we formalise the main contributions of the Reowolf project, and clarify their fundamental properties. Firstly, we give a **definition of PDL** in Section 3. Fundamentally, we distinguish dual notions of (communication) behaviour: *accepted* behaviours specify the set of all possible behaviours, while *constructed* behaviours restrict the accepted behaviours, by considering the execution of the session at large, forcing a causal ordering of messages, and interleaving the sending of messages with local computations. We prove key properties of PDL as Theorems 1 to 4. For example, these formalise an essential idea: behaviours constructed from any composite protocol are accepted by each part. Secondly, we give a **specification of PDL Runtimes**, in Section 4, as runtime systems which let users interleave protocol execution and refinement. We define their correctness as preserving Properties 1 to 4, which connect Invariants RI_1 to RI_4 to the PDL semantics. We implement an illustrative PDL runtime, show that it satisfies all properties but *completeness* (Property 4), and discuss approaches to implementing PDL runtimes that preserve these properties. Section 5 enumerates lines of future work: defining and evaluating various PDL protocols and runtimes. This includes the next steps toward verifying the correctness of the connector runtime implementation. Finally, we compare Reowolf’s PDL and connectors to related works (Section 6), before we conclude with a summary (Section 7).

All definitions and proofs in this article are formalised and machine-checked with the Coq proof assistant. The resulting artefact is available at <https://zenodo.org/records/14936561>. Please see Appendix A for a breakdown of the artefact. This includes an explanation of its parameters (*i.e.*, assumptions) and a table detailing the correspondences between terms in the article vs. the artefact.

2 User Communication Sessions via Connectors and PDL

Users of Reowolf connectors, physically distributed over the Internet, have a consistent experience of a shared communication session. To each user (and their application), the session is represented by a local, socket-like *connector* object. As with sockets, users interact only indirectly, via direct actions on their connector. The API provides users with three fundamental operations on their connector:

1. A user can **join a (possibly new) session** by participating in a set of rendezvous at chosen IP addresses. For example, Amy and Bob rendezvous at IPv4 address 65.49.82.45 to create and join a two-party session. Each rendezvous creates a new, persistent *port*, a logical place of message exchange.
2. A user can **reflect on the current behaviour** of the session, which fixes the messages exchanged so far. Thus, users can read received messages.
3. A user can **refine the session protocol** p by replacing it with its composition with user-provided protocol p' . We say the user *refines* p or *injects* p' into the session. Section 3 elaborates on PDL, but in short, protocols specify behaviour: protocols are stateful, message-passing programs, whose nondeterministic execution constructs behaviour. Consequently, injecting protocols is comparable to spawning new worker actors, processes, or threads.

Users are ultimately concerned with protocols and sessions via their *behaviour*. We adopt the notion and nomenclature of (finite) behaviours from Reo, *e.g.*, in [2]: each behaviour specifies the message, if any, observed per logical place (port) per discrete chronological instant (round), for $n : \mathbb{N} \triangleq \{0, 1, 2, 3, \dots\}$ rounds completed so far, *i.e.*, rounds are what we call the *indices* of behaviours. While the connector runtime defines message data as byte sequences for parity with UDP datagrams and IP packets, instead, we model message data \mathcal{D} with the natural numbers (\mathbb{N}). Ports $p, p', p_1, \dots : \mathcal{P}$ are an arbitrary data type that users understand as shared variables. We fix the following notations; here and henceforth, we use \uplus to denote the disjoint set union and we use \star to mark the absence of messages at ports.

$$\begin{aligned}
 d, d', d_1, \dots : \mathcal{D} &\triangleq \mathbb{N} \triangleq \{0, 1, 2, 3, \dots\} && (\underline{\text{data}} \text{ and } \underline{\text{natural numbers}}) \\
 m, m', m_1, \dots : \mathcal{M} &\triangleq \mathcal{P} \rightarrow \mathcal{D} \uplus \{\star\} && (\underline{\text{message map}} \text{ or } \underline{\text{messages}}) \\
 M, M', M_1, \dots : \mathcal{M}^* &\triangleq \text{list}(\mathcal{M}) && (\underline{\text{message map lists}} \text{ or } \underline{\text{behaviours}})
 \end{aligned}$$

The connector API also lets users follow the conventions of the socket API: messages can be sent and received on the fly. Precisely, these calls are transparently translated into *oneshot* protocols, which are injected as usual; **oneshot** is a protocol combinator, defined precisely in Figure 1, which lets users immediately exchange messages. For example, Amy sends data 4 to Bob via port p ; transparently, the connector injects **oneshot**(**send** $\{4\} \hookrightarrow p$) into the session.

Users expect their runtime to preserve Invariants RI_1 to RI_4 . Section 3 defines their underlying terms (*e.g.*, ‘*accepts*’) and Section 4 addresses their preservation.

RI₁ (consistency): Users’ observations of the messages remain consistent with some behaviour $[m_1, m_2, \dots, m_n]$, which is only extended over time, *i.e.*,

messages cannot change retroactively. To support PDL runtimes that are physically distributed (discussed in Section 5.1), we consider consistency to be preserved even if (a) some messages are hidden from some users, and (b) some users have not yet observed the latest round's messages m_n .

- RI₂ (acceptance):** Each protocol presently injected by users *accepts* the future session behaviours w.r.t. the PDL semantics. This safety property is the basis of the users' power to meaningfully participate in communications.
- RI₃ (provenance):** Each sent message has a sensible *provenance*: it can be traced back to some (user that injected some) protocol that sent it at some instant. Users may be oblivious to the identities or locations of these senders (or their peers in general). Nevertheless, provenance is desirable, because
 - it enforces the role of the runtime to facilitate real user communications, for example, rather than fabricating arbitrary messages, and
 - it ensures that users can be ultimately held accountable for the messages they send, *e.g.*, to enforce data protection regulations.
- RI₄ (productivity):** While it satisfies the other invariants, some next communication round is always eventually completed. Intuitively, this ensures that the PDL runtime does not get stuck or overlook specified behaviour.

3 Protocol Description Language (PDL)

This section defines the PDL, affording the expression of composable protocols. Ultimately, the meaning of protocols is in their two key notions of behaviour: the behaviours each protocol *accepts* and the subset of those it *constructs*.

In practice, users rely on some syntactic protocol composition operator which is symmetric, associative, and commutative w.r.t. the PDL semantics. For simplicity, instead, we represent protocols as decomposed into sets of their constituent *primitive* protocols. Thus, protocol composition is set union (\cup) and *empty protocol* $\{\}$ is the identity element w.r.t. composition. Fortunately, our formalism gives this protocol sensible semantics; later we show that it always has the same behaviour as the primitive protocol **loop sync**. We fix the following notations; here and henceforth, we use 2^ϕ to denote the powerset of any set ϕ .

$$\begin{aligned}
 s, s', s_1, \dots & : \mathcal{S} && \text{(primitive protocol statements)} \\
 S, S', S_1, \dots & : 2^\mathcal{S} && \text{(composite protocols, i.e., statement sets)} \\
 \text{constructs} \subseteq \text{accepts} & \subseteq 2^\mathcal{S} \times \mathcal{M}^* && \text{(semantic protocol-behaviour relations)}
 \end{aligned}$$

3.1 PDL Syntax and Small-Step Semantics

Each primitive PDL protocol is executed as a *worker*: a pair of a local *memory* in Σ (persistent, local stores mapping *variables* to data) and a PDL *statement* in \mathcal{S} . Like ports, variables $v, v', v_1, \dots : \mathcal{V}$ are arbitrary. Workers are comparable to the usual notions of agent, thread, or process. We fix the following notations:

$$\begin{aligned}
 \sigma, \sigma', \sigma_1, \sigma_2, \dots & : \Sigma \triangleq \mathcal{V} \rightarrow \mathcal{D} && \text{(variable stores)} \\
 w, w', w_1, w_2, \dots & : \mathcal{W} \triangleq \Sigma \times \mathcal{S} && \text{(\underline{w}orkers)}
 \end{aligned}$$

Figures 1 and 2 give the precise syntax and semantics of PDL primitives, respectively, which realise a granular notion of execution as a mostly-conventional while-language. Precisely, *update* gives a small-step operational semantics for PDL: *update* transitions between individual workers as a function of the given messages. Moreover, each *update* is labelled by a *synchronicity* $\delta : \Delta \triangleq \{\mathbf{X}, \checkmark\}$.

The *asynchronous* (\mathbf{X}) steps are largely conventional for while-languages. Example 1 demonstrates a typical imperative program: asynchronous steps implement Euclid’s algorithm for the greatest common divisor of integers in v and v' , using v'' to swap v and v' , and terminating as **done** with the result in v' .

Example 1 (asynchronous local computation: greatest common divisor of v and v').
while $0 < (v \% v')$ **do** (**write** $(v \% v') \hookrightarrow v''$; **write** $v' \hookrightarrow v$; **write** $v'' \hookrightarrow v'$).

Statements **send** and **recv** are also asynchronous, letting workers remember and react to the messages in their environment. Their roles in *update* are also similar: they specialise the worker’s reaction to the value at the given port. Later, Section 3.4 distinguishes their roles as one might expect: **send** is the *only* way to put new messages at ports. In this role, **send** E expresses a *nondeterministic choice* whose outcomes are identified by the elements of E . Example 2 demonstrates message-passing that is simultaneously conditional, synchronous, and non-deterministic; it synchronously forwards any number from port p , unless it is 21, to port p' , but incremented by 1 or 2, chosen non-deterministically.

Example 2 (message passing and non-determinism, within a synchronous round).
recv $p \hookrightarrow v$; **if** $(v \neq 21)$ **send** $\{v + 1, v + 2\} \hookrightarrow p'$.

The **sync** statement is uniquely *synchronous* (\checkmark). Intuitively, it delimits successive synchronous rounds. This is insignificant from the perspective of (updating) a single worker, but in Section 3.2, we formalise the intention: **sync** also acts as a synchronous barrier: **all** workers synchronise together, ‘committing’ the round’s behaviour, and advancing to the next. Example 3 demonstrates; it accepts (and constructs only) two rounds of behaviour: $\{p \hookrightarrow 1\}, \{p \hookrightarrow 2\}$. Section 5.2 discusses a generalisation, where a *subset* of workers synchronise.

Example 3 (completing two rounds). **send** $\{1\} \hookrightarrow p$; **sync**; **send** $\{2\} \hookrightarrow p$; **sync**.

$o : \mathcal{O} := + \mid - \mid * \mid \div \mid \% \mid \wedge \mid \vee \mid < \mid = \mid \neq$	(binary operators)
$e, e_1, e_2 : \mathcal{E} := d : \mathcal{D} \mid v : \mathcal{V} \mid e_1 \ o \ e_2 \mid \neg e$	(\mathcal{D} -type expressions)
$s, s_1, s_2 : \mathcal{S} := \mathbf{done} \mid \mathbf{sync} \mid \mathbf{write} \ e \ \hookrightarrow \ v \mid s_1 ; s_2$	(computation statements)
$\mid \mathbf{if} \ e \ s_1 \ \mathbf{else} \ s_2 \mid \mathbf{while} \ e \ \mathbf{do} \ s$	(control flow statements)
$\mid \mathbf{send} \ (E : 2^{\mathcal{E}}) \hookrightarrow p \mid \mathbf{recv} \ p \hookrightarrow v$	(communication statements)
<hr/>	
$\mathbf{if}' \ e \ s \triangleq \mathbf{if} \ e \ s \ \mathbf{else} \ \mathbf{done}$	$\mathbf{loop} \ s \triangleq \mathbf{while} \ 1 \ \mathbf{do} \ s$
$\mathbf{assert} \ e \triangleq \mathbf{if}' \ \neg e \ \mathbf{loop} \ \mathbf{done}$	$\mathbf{oneshot} \ s \triangleq s ; \mathbf{loop} \ \mathbf{sync}$

Fig. 1. The syntax of primitive PDL protocol-expressions \mathcal{E} and -statements \mathcal{S} .

$$\begin{aligned}
\text{update}(m, \langle \sigma, \mathbf{done} \rangle) &\triangleq \langle \mathbf{X}, \langle \sigma, \mathbf{done} \rangle \rangle \\
\text{update}(m, \langle \sigma, \mathbf{sync} \rangle) &\triangleq \langle \checkmark, \langle \sigma, \mathbf{done} \rangle \rangle \\
\text{update}(m, \langle \sigma, \mathbf{write } e \mapsto v \rangle) &\triangleq \langle \mathbf{X}, \langle \sigma[v := \text{eval}(\sigma, e)], \mathbf{done} \rangle \rangle \\
\text{update}(m, \langle \sigma, \mathbf{if } e \text{ s else } s' \rangle) &\triangleq \begin{cases} \langle \mathbf{X}, \langle \sigma, s \rangle \rangle & \text{if } \text{eval}(\sigma, e) \neq 0 \\ \langle \mathbf{X}, \langle \sigma, s' \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, \mathbf{while } e \text{ do } s \rangle) &\triangleq \begin{cases} \langle \mathbf{X}, \langle \sigma, \mathbf{done} \rangle \rangle & \text{if } \text{eval}(\sigma, e) = 0 \\ \langle \mathbf{X}, \langle \sigma, s ; \mathbf{while } e \text{ do } s \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, \mathbf{recv } p \hookrightarrow v \rangle) &\triangleq \begin{cases} \langle \mathbf{X}, \langle \sigma[v := m(p)], \mathbf{done} \rangle \rangle & \text{if } m(p) \neq \star \\ \langle \mathbf{X}, \langle \sigma, \mathbf{recv } p \hookrightarrow v \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, \mathbf{send } E \hookrightarrow p \rangle) &\triangleq \begin{cases} \langle \mathbf{X}, \langle \sigma, \mathbf{done} \rangle \rangle & \text{if } \exists e \in E, m(p) = \text{eval}(\sigma, e) \\ \langle \mathbf{X}, \langle \sigma, \mathbf{send } E \hookrightarrow p \rangle \rangle & \text{otherwise} \end{cases} \\
\text{update}(m, \langle \sigma, s_1 ; s_2 \rangle) &\triangleq \begin{cases} \langle \delta, \langle \sigma', s_2 \rangle \rangle & \text{if } s_1 = \mathbf{done} \\ \langle \delta, \langle \sigma', s_1' ; s_2 \rangle \rangle & \text{otherwise} \end{cases} \\
&\quad \text{where } \langle \delta, \langle \sigma', s_1' \rangle \rangle \triangleq \text{update}(m, \langle \sigma, s_1 \rangle)
\end{aligned}$$

Fig. 2. Primitive protocol update semantics, defined via $\text{update} : \mathcal{M} \times \mathcal{W} \rightarrow \Delta \times \mathcal{W}$ in terms of the (omitted) expression evaluation function $\text{eval} : \Sigma \times \mathcal{E} \rightarrow \mathcal{D}$.

3.2 Composite Protocols and (A)synchrony

Definition 1 generalises worker-update steps to sets of workers $W, W', \dots : 2^{\mathcal{W}}$. $W \xrightarrow{m} W'$ asynchronously updates **one** worker in W , and $W \xRightarrow{m} W'$ synchronously updates **each** worker in W . We omit W, W' , or m from terms of this form only when they are arbitrary or clear in context. For example, $(W \rightarrow)$ denotes the assertion that some worker in W can perform some asynchronous update.

Definition 1 (synchronous (\Rightarrow) and asynchronous (\rightarrow) worker-set steps).

$$\frac{\text{update}(m, w) = \langle \mathbf{X}, w' \rangle}{W \uplus \{w\} \xrightarrow{m} \{w'\} \cup W} \quad \frac{}{\emptyset \xRightarrow{m} \emptyset} \quad \frac{W \xRightarrow{m} W', \text{update}(m, w) = \langle \checkmark, w' \rangle}{W \uplus \{w\} \xRightarrow{m} \{w'\} \cup W'}$$

Lemma 1 identifies a property of the worker-set update semantics. Because it is one of the few properties underlying our main theorems, it is one of the few characteristics of the primitive PDL semantics that we consider to be essential. Here, it emerges from our definition of update in Figure 2. Section 5.2 discusses variants of the primitive PDL semantics which preserve this property.

Intuitively, Lemma 1 asserts that asynchronous⁶ updates do not let workers observe or remember the *absence* of messages at ports. The condition $W \neq W'$ excepts the case where adding messages *unblocks* the workers: removing a *blockage*: an asynchronous loop $W \rightarrow W$. Example 4 demonstrates: adding message $p \mapsto 1$ unblocks the worker. It still takes an asynchronous step, but the worker is changed: the new worker remembers the observed message, but not the prior blockage.

⁶ In fact, our definition of update affords generalising Lemma 1 to all unblocked updates. This is included in our artefact. But Lemma 1 suffices for our main results.

Lemma 1 (asynchronous message-determinism). *Any asynchronous step to distinct workers is preserved by adding messages (at empty ports).*

For each $W \neq W'$ and $m \leq m'$, $W \xrightarrow{m} W'$ implies $W \xrightarrow{m'} W'$.

Proof. By Definition 1, m asynchronously updates some $w \in W$ to some $w' \neq w$, and it suffices to show property P : that m' does likewise. We distinguish the cases of the first statement s of w . If $s = \mathbf{send} \ v \hookrightarrow p$ or $s = \mathbf{recv} \ p \hookrightarrow v$, then because $w \neq w'$, necessarily $m(p) \neq \star$. Because $m' \geq m$, $m'(p) = m(p)$, so P holds. For any other s , the update of w is independent of the messages, so P holds. \square

Example 4 (unblocking a worker blocked at $\mathbf{recv} \ p$ by adding $1 : \mathcal{D}$ at port p).

$$\begin{array}{l} \frac{\text{update}(m[p := \star], \langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle) = \langle \mathbf{X}, \langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle \rangle}{\{\langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle\} \xrightarrow{m[p := \star]} \{\langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle\}} \quad (\text{blocked}) \\ \frac{\text{update}(m[p := 1], \langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle) = \langle \mathbf{X}, \langle \sigma[v := 1], \mathbf{done} \rangle \rangle}{\{\langle \sigma, \mathbf{recv} \ p \hookrightarrow v \rangle\} \xrightarrow{m[p := 1]} \{\langle \sigma[v := 1], \mathbf{done} \rangle\}} \quad (\text{unblocked}) \end{array}$$

3.3 Linear Execution Traces and Accepted Behaviour

Where Section 3.2 generalised the execution of one worker to several, we finally generalise one execution step to linear *traces* of contiguous execution steps. We denote traces by concatenating \rightarrow and \Rightarrow steps. Finally, let $\xrightarrow{*}$ denote the transitive reflexive closure of \rightarrow . E.g., $W \xrightarrow{m*} W'$ abbreviates $W \xrightarrow{m} \xrightarrow{m} \dots \xrightarrow{m} W'$, i.e., a trace of asynchronous steps labelled with the same message map m .

Definitions 2 to 6 define key concepts: *traces* of contiguous steps are delimited into *rounds*, such that the n^{th} synchronous step ends the n^{th} round. We generally restrict our attention to *growing* rounds (Definition 6), where each message persists to the end of the round. Intuitively, each growing round incrementally collects messages in m until \xRightarrow{m} makes m observable as *behaviour* (Definition 4).

Definition 2 (trace). *A trace is a sequence of contiguous \rightarrow or \Rightarrow steps.*

Definition 3 (round). *A round is a trace with only one synchronous step, at the end. For example, round $W \xrightarrow{m_1} \xrightarrow{m_2} \xrightarrow{m_3} \xRightarrow{m_4} W'$ consists of four steps.*

Definition 4 (behaviour). *The behaviour of a trace is the concatenation of its synchronized messages. E.g., the behaviour of $\xrightarrow{m_1*} \xRightarrow{m_1} \xrightarrow{m_2*} \xRightarrow{m_2}$ is $[m'_1, m'_2]$.*

Definition 5 (message \leq). $m \leq m' \triangleq \forall p : \mathcal{P}, m(p) \in \{m'(p), \star\}$.

Definition 6 (growing round). *A round is growing iff $m \leq m'$ for each m and m' labelling consecutive steps in the round.*

Definition 7 defines *acceptance*, the first of two notions of protocol behaviour. Acceptance is *imperative* in the sense that behaviour arises from traces of contiguous ('stateful') execution steps. However, acceptance is also *declarative* in the sense that messages are growing but otherwise unspecified. Acceptance treats protocols as behaviour-recognisers or -acceptors, i.e., we check for acceptance given protocol-behaviour pairs. Acceptance is suitable for reasoning about execution in an environment open to messages sent externally, e.g., to reason about *what* executions of a known protocol are possible in an unknown system.

Definition 7 (accepted behaviour). *Protocol S accepts the behaviour of each trace consisting of only growing rounds, starting from $\{\langle \emptyset, s \rangle \mid s \in S\}$.*

Example 5 (examples of protocols accepting behaviours).

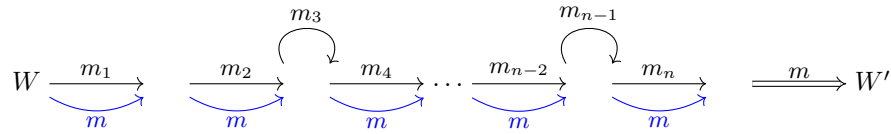
- $\{\mathbf{loop\ sync}\}$ synchronises each round immediately, regardless of the messages, like the empty protocol $\{\}$, i.e., leaving behaviour unconstrained.
- $\{\mathbf{loop\ done}\}$ and $\{\mathbf{done}\}$ perform an endlessly unproductive sequence of asynchronous updates, so they accept only the zero-round behaviour $[]$.
- $\{\mathbf{recv\ } p \hookrightarrow v ; \mathbf{sync}\}$ and $\{\mathbf{recv\ } p \hookrightarrow v ; \mathbf{send\ } \{v\} \hookrightarrow p ; \mathbf{sync}\}$ accept $[]$ and any $[m]$ as long as $m(p) \neq \star$, i.e., there must be *some* message at p .
- $\{\mathbf{send\ } \{0, 1\} \hookrightarrow p ; \mathbf{recv\ } p \hookrightarrow v ; \mathbf{assert\ } v ; \mathbf{sync}\}$ sends either 0 or 1 at port p , but then only synchronises if 1 was chosen. This demonstrates a useful pattern: workers can process and suppress behaviour before it is observable.

Recall Lemma 1: asynchronous steps are preserved by adding to the messages. Lemma 2 follows: for each growing round, there is a constant round with the same behaviour. Thus, all behaviours are sufficiently characterised by focusing only on *constant* rounds. These have useful properties, namely Lemma 3: workers in constant rounds cannot interact, because the messages are fixed, so each workers' updates proceed concurrently. Finally, Theorem 1 results: the change to each worker each round is entirely *determined* only by its behaviour. This has practical applications. For example, one can reliably ‘replay’ the execution of any set of workers, arbitrarily many rounds into the future, given only the behaviour they produced. Section 5.1 discusses how this lets (distributed) PDL runtimes explore traces concurrently, and use behaviours to identify (desirable) constant rounds.

Definition 8 (constant round). *A round is constant in messages m or m -constant iff each of its steps is labelled m . Note that constant rounds are growing.*

Lemma 2 (growing to constant). *For each growing round synchronising m , a round exists with the same start and end workers, but which is constant in m .*

Proof. Each given step $W_k \xrightarrow{m_k} W_{k+1}$ (in black) is ignored if $W_k = W_{k+1}$, i.e., it is a loop. Otherwise, as $m_k \leq m$, Lemma 1 lifts it to $W_k \xrightarrow{m} W_{k+1}$ (in blue).

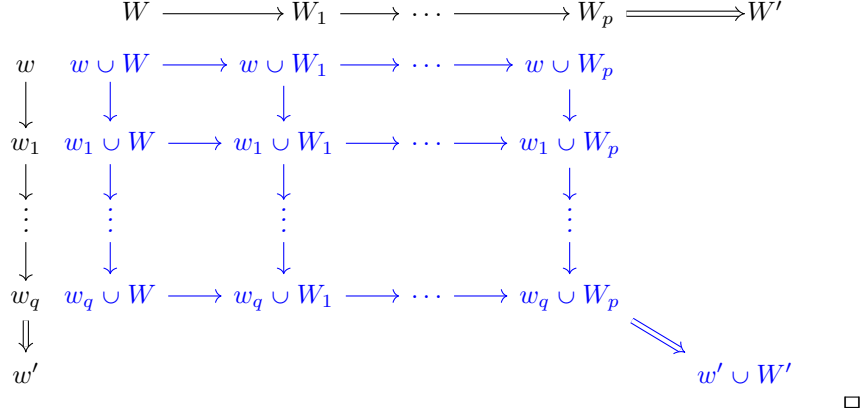


□

Lemma 3 (constant confluence). *For any workers W and messages m , all m -constant rounds starting from W necessarily end in the same workers.*

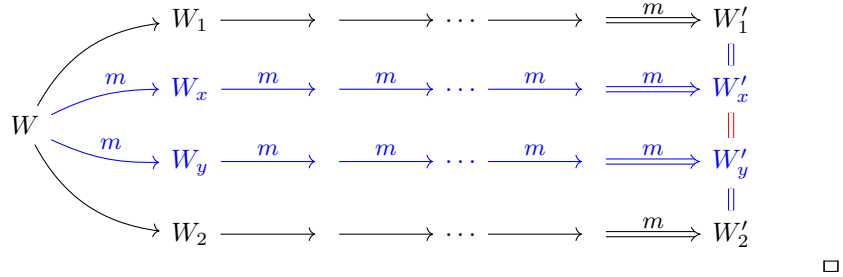
Proof. We prove the confluence of rounds from W by induction on the list of workers in W . The base case of $W = \emptyset$ is trivial: the only step is $\xrightarrow{m} \emptyset$. In the inductive case, rounds $W \xrightarrow{m^*} \xrightarrow{m}$ are confluent. We omit the proof that rounds $\{w\} \xrightarrow{m^*} \xrightarrow{m}$ are confluent, where there is only one worker to update each step.

Take arbitrary rounds from W to W' and from $\{w\}$ to $\{w'\}$ respectively (in black). Their steps can be arbitrarily interleaved (in blue), but must end in $W' \cup \{w'\}$. For legibility, the implicit m is omitted from each step in the figure below.



Theorem 1 (round determinism). Growing rounds with the same start (W) and behaviour (m) necessarily have the same end ($W'_1 = W'_2$).

Proof. By Lemma 2, for each given round (in black), some constant round has the same behaviour (in blue), whose ends are the same (in red) by Lemma 3.



Finally, PDL satisfies strong (de)compositionality properties: by Theorems 2 and 3, the behaviours accepted by any protocol is the set-intersection of those of its constituent protocols. First of all, we expect PDL programmers to pervasively rely on Theorem 2 to control the behaviour of their session, because they can rely on the runtime system to avoid executions whose behaviour is not accepted by their injected protocols. The specification of accepted behaviours is similar to *constraint programming*: injecting protocols constrains the future behaviour. Users can also exploit this (de)compositionality during reasoning and verification, for example, to predict possible messages before the session begins.

Theorem 2 (compositionality). Given growing rounds $W_\alpha \xrightarrow{*m} W_\omega$ and $W_a \xrightarrow{*m} W_z$, there exists a growing round $(W_\alpha \cup W_a) \xrightarrow{*m} (W_\omega \cup W_z)$.

Proof. Given rounds r_1, r_2 (in black), construct a new one (in blue) from the union of all initial workers. Repeat asynchronous steps of r_1 (with the given messages), and then r_2 (but replace messages with m using Lemma 1). Synchronise m .

$$\begin{array}{c}
W_\alpha \xrightarrow{m_1} W_\beta \xrightarrow{m_2} \dots \xrightarrow{m_3} W_\psi \xRightarrow{\quad\quad\quad m \quad\quad\quad} W_\omega \\
\begin{array}{c} W_\alpha, m_1 \\ W_a \end{array} \xrightarrow{\quad} \begin{array}{c} W_\beta, m_2 \\ W_a \end{array} \xrightarrow{\quad} \dots \xrightarrow{\quad} \begin{array}{c} W_\psi, m \\ W_a \end{array} \xrightarrow{\quad} \begin{array}{c} W_\psi, m \\ W_b \end{array} \xrightarrow{\quad} \dots \xrightarrow{\quad} \begin{array}{c} W_\psi, m \\ W_y \end{array} \xrightarrow{\quad} \begin{array}{c} W_\omega, \\ W_z \end{array} \\
W_a \longrightarrow W_b \longrightarrow \dots \longrightarrow W_y \xRightarrow{\quad m \quad} W_z \quad \square
\end{array}$$

Theorem 3 (decompositionality). Given growing round $W_\alpha \cup W_a \xrightarrow{*} \xRightarrow{m} W$, there exist worker sets W_ω and W_z and growing rounds $(W_\alpha \xrightarrow{*} \xRightarrow{m} W_\omega)$ and $(W_a \xrightarrow{*} \xRightarrow{m} W_z)$, such that $W = (W_\omega \cup W_z)$.

Proof. The proof is inductive, building two traces using the given round's steps, from the first to the last. We consider an arbitrary step from $W_\psi \cup W_y$ with some m' to some W , where, by the inductive hypothesis, $W_\alpha \xrightarrow{*} W_\psi$ and $W_a \xrightarrow{*} W_y$ are constructed so far. If the step is asynchronous, exactly one worker w is updated to some w' , so $W = ((W_\psi \cup W_y) \setminus \{w\}) \cup \{w'\}$. W_ψ or W_y must contain w . Let it be W_ψ ; we omit the symmetric case. Construct $W_\alpha \xrightarrow{*} W_\psi \xrightarrow{m'} (W_\psi \setminus \{w\}) \cup \{w'\}$ (in black) and leave the other round intact (in blue). Indeed $(W_\psi \setminus \{w\}) \cup \{w'\} \cup W_y = W$. Only the final step is synchronous; by definition of \Rightarrow , there exist W_ω and W_z updated by $W_\psi \xRightarrow{m'} W_\omega$ and $W_y \xRightarrow{m'} W_z$, where $W = W_\omega \cup W_z$ (in red).

$$\begin{array}{c}
W_\alpha, \xrightarrow{m_1} W_\beta, \xRightarrow{\quad} W_\beta, \xRightarrow{\quad} W_\beta, \dots W_\psi, \xRightarrow{\quad m \quad} W_\omega, \\
W_a \xRightarrow{\quad} W_a \xrightarrow{m_2} W_b \xrightarrow{m_2} W_c \dots W_y \xRightarrow{\quad m \quad} W_z \quad \square
\end{array}$$

3.4 Behaviour Constructed from PDL Protocols

Recall that acceptance uses the *update* function to compute workers from the prior workers, given any growing messages. Construction (Definition 10) also introduces *offered* (Definition 9), which computes messages from the prior workers.

Construction is suitable as a model for executing the system in its entirety, notably, by PDL runtimes, in order to compute behaviour from the session protocol. Construction ensures the desired characteristics. Each observed message has a sensible provenance (Invariant RI_3). Moreover, each step is computable from the prior step, despite the domains of port, protocol, and data being arbitrarily large (thus, infeasible to enumerate). Figure 3 visualises the incremental construction of rounds, unfolding the (finite) options of each next messages and workers, by applying functions *offered* and *update*, in alternating fashion: the n^{th} messages are selected from the n^{th} workers' offers, the $(n+1)^{\text{st}}$ workers are selected from the n^{th} workers updated with the n^{th} messages, and so on.

Definition 9 (offered messages). w offers to send d at p iff $\langle p, d \rangle \in \text{offered}(w)$.
 $\text{offered}(\langle \sigma, s \rangle : \mathcal{W}) : \text{list}(\mathcal{P} \times \mathcal{D}) \triangleq [\langle p, \text{eval}(\sigma, e) \rangle \mid (\text{send } E \hookrightarrow p) = s, \forall e \in E]$

Definition 10 (constructive round). A growing round is *constructive* iff, for each step from some W labelled m' , where m labels the prior step if it exists and $m = \emptyset$ otherwise, either $m = m'$ or there exist $w \in W$ and $\langle p, d \rangle \in \text{offered}(w)$, such that $m[p := d] = m'$, in which case we say w sends d at p .

Definition 11 (constructive behaviour). Protocol S constructs the behaviour of each trace from $\{\langle \emptyset, s \rangle \mid s \in S\}$ consisting of only constructive rounds.

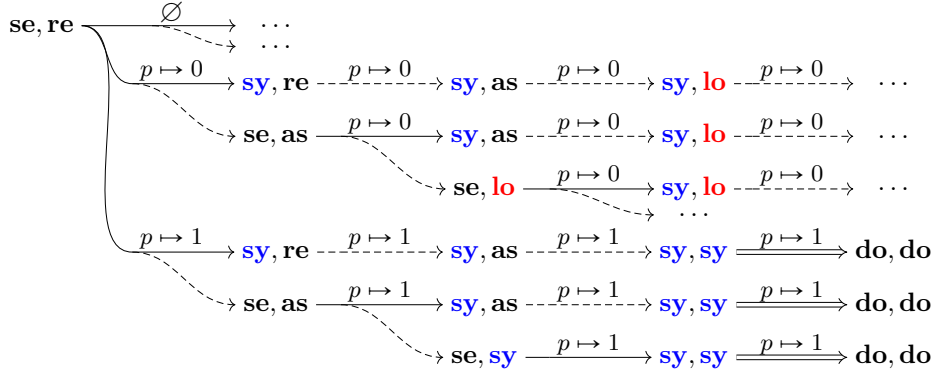


Fig. 3. This shows the (search) tree of constructive rounds starting from the workers initialised from the protocol $\{(\text{send } \{0, 1\} \hookrightarrow p; \text{sync}), (\text{recv } p \hookrightarrow v; \text{assert } v; \text{sync})\}$, from Example 5. Each worker is identified by the first two letters of its statement, *e.g.*, worker $\langle \{v \mapsto 0\}, (\text{assert } v; \text{sync}) \rangle$ is abbreviated as ‘as’. We colour **lo** (for **loop done**) and **sy** to reveal patterns. Arrows depict (a)synchronous steps as usual, but we *dash* them iff only their *second* worker is updated. Looping traces are truncated to (\dots) .

Example 6 (Examples of Protocols Constructing Behaviours).

- $\{\text{loop sync}\}$ constructs arbitrarily many rounds without any messages. Again, this primitive protocol behaves like the trivial, empty protocol $\{\}$.
- $\{\text{recv } p \hookrightarrow v; \text{sync}\}$ is stuck awaiting a message that is never sent.
- $\{\text{recv } p \hookrightarrow v; \text{send } \{v\} \hookrightarrow p; \text{sync}\}$ has a cyclic dependency on the message at port p . The **recv** blocks forever for a message that is never sent.
- $\{(\text{send } \{0, 1\} \hookrightarrow p; \text{sync}), (\text{recv } p \hookrightarrow v; \text{send } \{6 + v\} \hookrightarrow p'; \text{sync})\}$ demonstrates the propagation of a nondeterministic choice between workers, through a message at port p . The recipient is oblivious to the nondeterminism, expressing the functional relationship $m(p) + 6 = m(p')$.

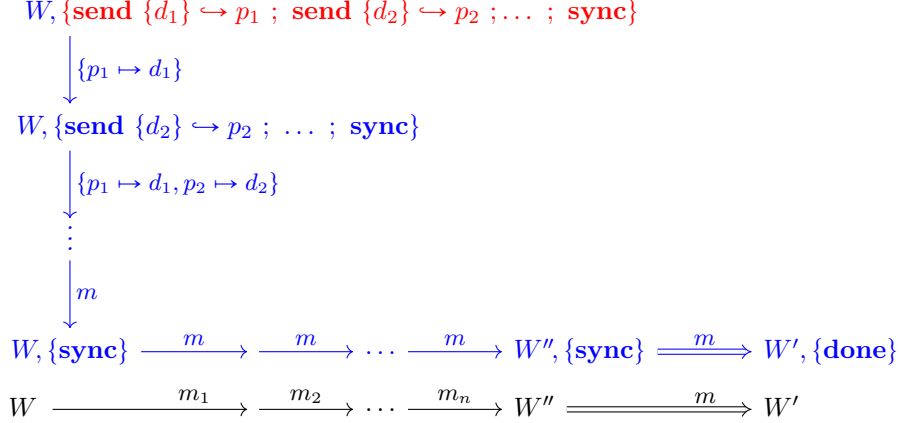
Because constructed rounds are growing, many prior results also apply to construction. Rounds are still determined by their behaviour (Theorem 1), and composing protocols preserves their constructed behaviours (Theorem 2, because the new round is constructive if both given rounds are constructive). However, unlike acceptance, construction is not *decompositional* (Theorem 3): composite protocols construct behaviours *not* constructed by the constituents independently. Example 7 demonstrates behaviour emerging from protocol interactions.

Example 7 (behaviour constructed from protocol interactions). Consider primitive protocol $\{(\mathbf{send} \{1\} \hookrightarrow p_1 ; \mathbf{recv} p_2 \hookrightarrow v ; \mathbf{sync})\}$ where $p_1 \neq p_2$, and the same but with p_1 and p_2 swapped. Independently, each protocol forever awaits a message that is never offered. But their composition constructs $[\{p_1 \mapsto 1, p_2 \mapsto 1\}]$, where each worker first sends and then receives the message sent by its peer.

Intuitively, construction resembles (nondeterministic) *actor programming*: computation and communication unfold inter-dependently. Theorem 4 precisely relates the behaviours accepted and constructed by each protocol S : behaviours accepted by S are those constructed by S in composition with some existentially quantified ‘oracle’ protocol, which stands in for the environment of S , offering whatever messages S needs to end the round. Like accepted behaviours, a protocol’s oracles are generally not enumerable unless ports, messages, or behaviours are finitely enumerable. This gives PDL programmers another view on acceptance: the behaviours their protocol can construct as a part of any (composite) protocol.

Lemma 4 (round oracle). *Given growing round $W \xrightarrow{*} \xRightarrow{m} W'$, there exists oracle $s_o : \mathcal{S}$, worker w'_o , and constructive round $W \cup \{\langle \emptyset, s_o \rangle\} \xrightarrow{*} \xRightarrow{m} W' \cup \{w'_o\}$.*

Proof. Build an oracle (in red) that sends each message in m in some order before a **sync** statement. Build a round (in blue) in two stages. The first (vertical) stage builds m , one message at a time, while W takes no steps. The second (horizontal) stage reproduces the given round (in black), but made m -constant with Lemma 2.



□

Theorem 4 (acceptance vs. construction). *Protocol S accepts behaviour M iff (\Leftrightarrow) there exists an ‘oracle’ protocol S_o where $S \cup S_o$ constructs M .*

Proof. (\Rightarrow) By Lemma 2, a trace of constant rounds r exists whose behaviour $M \triangleq [m_1, m_2, \dots, m_n]$ is accepted by S . Apply Lemma 4, but generalized to an n -round oracle $(o_1 ; o_2 ; \dots ; o_n)$ constructing M . (\Leftarrow) By definition, a trace of constructive rounds r with behaviour M from $S \cup S_o$ exists. By Definition 10, $S \cup S_o$ accepts M . Finally, by Theorem 3, S accepts M , i.e., removing S_o preserves acceptance. □

Acceptance and construction provide useful, dual views on behaviour. Their differences reflect their different assumptions about the execution environment. Acceptance reflects ignorance of the environment. This view is useful to users, who reason about the behaviour of known protocols as parts of unknown composite (session) protocols. Construction is holistic, in the sense that it closes the system, only considering messages as they are sent, interleaved with worker updates. This view is appropriate for reasoning about systems at large, which users will do some of the time, but which PDL runtimes do all of the time.

4 PDL Runtimes

Section 3 defined the PDL syntax and semantics. In these terms, we specify and demonstrate the user interface and correctness properties of PDL runtimes.

4.1 A Specification of PDL Runtimes

A PDL runtime is an interactive system that realises a communication session. Each runtime configuration $c, c_1, c', \dots : \mathcal{C}$ has a *current* (session) protocol. Users communicate indirectly, by directly and arbitrarily interleaving user actions:

1. Some user *injects* a chosen input protocol into the current protocol.
2. Users observe the behaviour that results from *running* the system.

Precisely, a PDL runtime provides four user-facing operators:

$$start : \mathcal{C} \quad run : \mathcal{C} \rightarrow \mathcal{M}^* \times \mathcal{C} \quad proto : \mathcal{C} \rightarrow 2^{\mathcal{S}} \quad inject : \mathcal{C} \rightarrow 2^{\mathcal{S}} \rightarrow \mathcal{C}$$

where, as introduced earlier, \mathcal{S} is the domain of PDL programs, \mathcal{M} is the domain of message maps, and \mathcal{M}^* is the domain of message map lists, *i.e.*, behaviours.

The PDL runtime preserves the users' basic expectations of the current protocol by preserving Properties 1 and 2.

Property 1 (initially trivial protocol) $proto(start) = \{\}$.

Property 2 (injection composes) $proto(inject(c, S : 2^{\mathcal{S}})) = proto(c) \cup S$.

Additionally, PDL runtimes must ultimately preserve each of the *runtime invariants* defined in Section 2. Each behaviour must be observed consistently between users (Invariant RI_1), accepted by each component of the current protocol (Invariant RI_2), the result of a round if it exists (Invariant RI_4), and such that each message must have been sent by a user-injected protocol (Invariant RI_3).

Because constructed behaviours are accepted by each injected part of the session protocol, it suffices for the PDL runtime to preserve Properties 3 and 4.

Property 3 (soundness) *Each $run(c) = \langle M, c' \rangle$ implies $proto(c)$ constructs M .*

Property 4 (completeness) *For all $c : \mathcal{C}$, if any $M \neq \square$ exists that is constructed by $proto(c)$, after some finite run-steps from c , any $M' \neq \square$ is observed.*

4.2 Example: The Silent PDL Runtime

To demonstrate our PDL runtime specification, we consider the *silent PDL runtime*. Its implementation is comprised of the following function definitions, fixing its configurations as the composite PDL protocols $\mathcal{C} \triangleq 2^S$:

$$start \triangleq \{\}. \quad run(S) \triangleq \langle [], S \rangle. \quad proto(S) \triangleq S. \quad inject(S, S') \triangleq S \cup S'.$$

It is easy to see why this implementation satisfies Properties 1 to 3: it simply collects injected protocols, and always produces zero-round behaviour $[]$, which every protocol constructs, by definition. Our artefact includes a simple proof of each of these properties. But clearly this implementation is not *complete* (Property 4). We use protocol $\{\}$ constructing $\{\} \xRightarrow{\emptyset} \{\}$ as witnesses for our proof of *incompleteness*: while $\{\}$ constructs $[\emptyset]$, run steps only ever observe $[]$.

Hopefully, this demonstrates that the preservation of completeness is the first major challenge in implementing correct PDL runtimes. The difficulty arises from the fact that rounds may complete after arbitrarily many (finite) steps. Indeed, it is generally undecidable whether given workers can synchronise; it is analogous to the halting problem. Fortunately, because finite protocols offer finitely many messages, the tree of traces rooted at any finite workers W is finite *per depth*. Figure 3 visualises how offers and updates guide these searches. Of course, typical search optimisations can improve efficiency drastically. For example, by caching visited workers, many *transpositions* (the same workers reached via distinct traces) are avoided. We expect the properties of PDL to afford even more optimisations. For example, because of *determinism* (Theorem 1), we expect completeness to be preserved if $W \xrightarrow{m[p:=\star]} W'$ stays unexplored once a constructive $W \xrightarrow{m[p:=d]} W'$ is discovered. But presently, rigorous proofs of these claims are still future work.

5 Future Work

5.1 Formalising the Rest of the Connector Runtime

We strive to recreate the existing Rust implementation of the connector runtime as a PDL runtime, atop our present specification, such that we can check its correctness. Here, we overview its facets which remain to be formalised.

Round Search We conjecture that it is decidable whether given workers synchronise after a bounded number of asynchronous (\rightarrow) steps. Such a proof lays the groundwork for terminating search algorithms that are correct up to a *fuel* : \mathbb{N} bound. For example, we imagine modelling the *timeout* mechanism of the connector runtime as run fuel. Many round-search algorithms are conceivable. Which of them are correct and efficient? We also hope to model the algorithm underlying the connector runtime and (ideally) verify its correctness.

During the formalisation of the semantics of PDL, we also discovered an elegant relation between the description of a protocol as a proposition and the runtime implementation of a protocol as a proof. In proof terms, a protocol states the existence of a round (or an n -round trace), and the runtime works to find a

proof of that proposition. We aim to explore this relation further in the future, to discriminate runtime algorithms from a proof theoretic perspective.

Distributed Workers Distributed PDL runtimes partition the workers in each configuration \mathcal{C} over a network of *processes* $\pi, \pi', \pi_1, \dots : \Pi$. Processes only interact by passing asynchronous *control messages*. The processes cooperate to simulate a search (see *Round Search*, above). The execution of all workers sharing (data) messages is simulated: each process π isolates its subset of workers in a local message environment m , and then π explicitly informs its peers of newly-sent message $\langle p, d \rangle$ via a control message $\langle \text{sent}, n, m, p, d \rangle$, where $n : \mathbb{N}$ identifies the current round, to avoid confusion if control messages arrive late.

The processes realise a synchronised round $W \xrightarrow{*} \xRightarrow{m} W'$ through explicit coordination. Each round, an elected process selects and announces an arbitrarily chosen m after learning that, for each process π (including itself), there exists some $m_\pi \subseteq m$, where, locally, $W_\pi \xrightarrow{*} \xRightarrow{m} W'_\pi$. This information is communicated to the leader from π via the control message $\langle \text{ready}, n, m_\pi \rangle$. To minimise the burden on the leader, the connector runtime lets processes aggregate this information toward the leader. A sink tree is overlaid atop the network, with the leader at the root. With $\langle \text{ready}, n, m \rangle$, a child process π informs its parent that each process in the subtree rooted at π is ready to synchronise with some $m_\pi \subseteq m$.

Restricted Ports Users of the connector runtime define *access* as a process-port relation, such that each process π can send and receive messages at port p only if $\text{access}(\pi, p)$. Port messages are only ever observed by accessors. For example, $\langle \text{sent}, n, m, p, d \rangle$ is sent only to p -accessors, and m omits messages of ports not accessed by the recipient. Users of the connector runtime also define which process *determines*: $\mathcal{P}_d \rightarrow \Pi$ each *deterministic* port $\mathcal{P}_d \subseteq \mathcal{P}$. Whenever $\pi = \text{determines}(p)$, only workers at π may execute **send** $E \hookrightarrow p$ where $|E| \leq 1$, *i.e.*, the nondeterministic choice is trivial. The benefit of deterministic ports is that their values can be omitted from control communications while preserving determinism: all growing rounds that start from the same W with the same messages at ports $\mathcal{P} \setminus \mathcal{P}_d$ necessarily have the same messages at port \mathcal{P}_d .

Access and determinism give users greater control over how the PDL runtime communicates port data via control messages. For example, users can restrict the observation of messages at particular ports to (the users at) particular processes.

Channel API The connector runtime lets users express *access* and *determines* (see *Restricted Ports*, above) implicitly via its API, which is a restriction of that described in Section 2: sets of users rendezvous at sets of IP addresses; each rendezvous includes only a *sender* and *receiver* process, and creates a new *channel* comprised of ports p and p_d . The latter is accessed only by this sender and receiver and is determined by the sender. Users understand p_d as carrying user data across the channel from the sender to the receiver only, while p is used in session-wide control communications, *e.g.*, to discriminate values sent at p_d .

As users never read values at p , these act purely as discriminators and can be chosen arbitrarily by the connector runtime. In fact, it suffices if the values at p

are restricted to $\{0, 1\}$, such that control messages can be densely packed into bit vectors. Each p is split into multiple ports if more bits are necessary.

5.2 Exploring Variations of the PDL

We intentionally minimise the coupling between the syntax and semantics of PDL primitives (Section 3.1) and the rest of the language (Sections 3.3 and 3.4) to ease future experimentation with variants of PDL.

Ports as Data If PDL is changed such that $\mathcal{P} \subseteq \mathcal{D}$ (ports are data), protocols become significantly more expressive and flexible. For example, `send` $\{e_d\} \hookrightarrow e_p$ sends dynamic data (e_d) at a dynamic port (e_p). However, this would make protocols more difficult to reason about; *e.g.*, generally, we could not statically check which workers or processes *access* given ports (see *Restricted Ports*, above).

N-Round Lookahead We consider definitions of *productivity* (Invariant RI_4) which change the role of synchronisation. For example, we consider a generalisation where PDL runtimes must always *maximise* the number of completed rounds up to a *lookahead* bound which is currently implicitly 1. Consider Example 8, which synchronises N rounds before blocking forever, where N is the data at port p_N in the first round. With 1-lookahead, the PDL runtime is correct to complete the first round with behaviour $\{p_N \mapsto 1\}$. But with 3-lookahead, this would be incorrect, because it would not produce the maximal number of rounds up to the end of round 3, *e.g.*, where $\{p_N \mapsto 3\}$ and $\{p_N \mapsto 182\}$ are maximal instead.

Example 8 (N-Sync). `recv` $p_N \hookrightarrow v$; **while** v **do** (**write** $v - 1 \hookrightarrow v$; **sync**).

Local Synchronisation We consider decoupling different workers' synchronisations, *e.g.*, as in the Reo semantics. Precisely, an unspecified subset of workers *stutter* each round: maintaining their states and not interacting. Intuitively, this change empowers the PDL runtime to make more decisions at the cost of weakening the meaning of **sync**. In practice, this change prevents slow workers from impeding productivity, but it introduces threats of unfairness and starvation, unless they are prevented, *e.g.*, by users specifying how workers are prioritised. This change affords the concurrent synchronisation of local *synchronous regions* of the system, as is done with Reo in [2], and with Dreams in [46].

5.3 Definition, Analysis, and Optimisation of PDL Protocols

PDL is designed to afford (de)compositional reasoning about protocols and their properties. We see benefit in future work that develops a corpus of PDL protocols that solve useful problems, or have desirable properties.

We are particularly interested in letting PDL runtimes transparently optimise execution by recognising protocols and exploiting their known properties. Most interestingly, these may include opportunities that are unknowable to the users, because users lack the overview of the session protocol, *e.g.*, in ad-hoc sessions

between strangers over the Internet. For example, if Amy in Helsinki routes outgoing messages to Bob in Tokyo, and Bob filters incoming messages before forwarding them to Dan in Helsinki, the runtime can transparently re-configure the session to filter Amy’s messages before routing them to Dan, *i.e.*, avoiding the inter-continental round-trip. These kinds of session optimisations are already supported by (and benchmarked with) the connector runtime [20], but currently, they must be recognised and triggered manually. The Reowolf project documentation [20] highlights the potential of systematic protocol analysis and transformation via existing tools. For example, can we encode PDL protocols as annotated port-graphs and protocol transformations as graph-rewriting rules, *e.g.*, using the PBPO⁺ graph-rewriting formalism and tools [44]?

6 Related Work

6.1 Related to Behavioural Specification with the Reowolf PDL

Here, we compare PDL to well-researched synchronous languages Reo, Esterel, and Lustre. We leave out of scope the comparisons to the many other (synchronous) languages such as Signal [22,6], ARx [47], and HipHop.js [8].

Reo PDL is directly inspired by Reo, so the languages have many similarities. Reo is also used to specify and coordinate synchronous and multi-party communications [1,2] in interactive systems, regulating the interactions between software components as constraints on their messages. Notably, the compiler [15] of textual Reo [14] with constraint automata semantics [4] generates executable *coordinators* that interface with users. Reo is more declarative, easing static protocol verification and transformation, and top-down coordinator re-configuration [36,38,39,37].

However, there is less emphasis on the dynamic composition of Reo protocols, or their application to distributed systems. For example, Reo coordinators are centralised, and cannot interface with other coordinators. In contrast, Reowolf is designed around its dynamic and distributed applications: the protocol unfolds dynamically as users inject new protocol components on the fly.

Esterel PDL and Esterel have similar syntax, and the same control flow: sequential, but punctuated by synchronous message passing [7,17,21].

Esterel reflects its design for a different use case: *real-time reactive systems*, which emphasise timely reactions to external events. Hence, notably, Esterel and PDL have very different relationships with nondeterminism: both embrace it for program composition, but only in PDL are nondeterministic programs executable. Esterel programmers are responsible for ensuring determinism, whereas PDL programmers rely on the behaviour being determined at runtime. In fact, nondeterminism is desirable in PDL protocols, because it affords flexibility to later protocol composition and refinement. Effectively, PDL programmers enjoy ignorance of their environment, at the cost of complexity in the runtime system.

Lustre Lustre [26,25] semantically separates synchronous computations (within a logical instant) and sequences of computation (over consecutive instants).

Lustre has operators calculating (arithmetic) functions and operators ‘scheduling’ functions over time. Composition of synchronous programs is done per instant, on a shared memory machine. Lustre provides some meaningful primitives to specify these functions constructively, which requires them to be deterministic.

In comparison, PDL has nondeterministic and synchronous primitives. Agents interpret PDL programs and interact with other agents. Agents explicitly delineate instants with the **sync** statement, whose semantics guarantees the consistent values at variables shared between agents (*i.e.*, ports).

6.2 Related to Network Programming with Reowolf Connectors

Dreams Like the connector runtime, the engine of the Dreams framework [46] translates Reo-like synchronous protocols into networks of automated, message-passing agents, coordinated via distributed consensus algorithms. The works focus on different features. Dreams statically partitions the system into *synchronous regions*, enabling (only) region-local synchronisation and reconfiguration. In contrast, Reowolf has global synchronisation (*i.e.*, the trivial case of Dreams), but lets users change the protocol on the fly. We see the best features of Dreams and Reowolf as complementary, and we see promise in combining their strengths, *e.g.*, by relaxing the meaning of our **sync** statement, as discussed in Section 5.2. Can future sessions have synchronous regions (as in Dreams), while allowing users to change protocols in synchronous regions on the fly (as in Reowolf)?

Bulk Synchronous Processing (BSP) BSP is an abstract model of parallel computation [50]. The main idea is to break time into sequential *supersteps*, within which, processors compute in parallel, and between which, processors pass messages and synchronisation barriers. Literature has developed software and hardware for BSP, typically for the purpose of high performance computing, maximising program portability [10,23] and performance [48,51,52]. Many BSP ideas are also evident in Reo(wolf), *e.g.*, in emphasising the separation of computation from communication. Reowolf takes it further: like BSP supersteps, rounds of PDL communication behaviour are separated by synchronisation barriers. Which BSP hardware / runtime environments are suitable PDL runtimes?

Message Passing Interface (MPI) The MPI standard [12] eases and decouples the tasks of 1. library developers of general-purpose languages, and 2. their users implementing data-parallelism in general, and large simulations in particular [24].

Reowolf and MPI have in common that they augment host applications with high-level abstractions for coordinated, multiparty message passing. MPI-2 in particular overlaps with Reowolf in expressing task-parallelism via workers.

While Reowolf and MPI differ considerably in the details of their programming abstraction, more fundamentally, they have different consequences on runtime behaviour. Reowolf focuses on internet programming, so PDL affords a distributed and continuous refinement of the session protocol. In contrast, MPI affords the abstract specification of task- and data-parallelism in programs at compile-time, such that programmers can reason about run-time performance.

Software Defined Networks (SDN) Like Reowolf, SDNs [30] provide an abstraction over distributed systems. For example, the OpenFlow [40] control protocol is used for remote administration of network switch packet-forwarding tables. Rules can be made on a controller, and dynamically pushed to the network switches, *e.g.*, to change the routing algorithm [42]. Tools such as NorthStar [35] and Khathará [9] deploy and manage containerised applications atop SDNs.

SDNs and Reowolf provide abstractions over different layers in the OSI stack. SDNs are suited to network administrators, while PDL isolates coordination logic typical to applications. As such, we see potential for their combination. Can connectors efficiently route messages (via transparent usage of SDN technologies) within the bounds of what users accept (as users specify via PDL protocols)?

Multi Party Session Types (MPST) Dyadic session types specified the communication behaviour of message channels [27]. MPSTs generalised these to multiparty sessions; each *global* MPST (specifying the session at large) is projected onto each peer, yielding *local* MPSTs (specifying roles in the session) [28]. MPST variants are embedded in host languages such as Rust [11], Scala3 [13], Haskell [41], and OCaml [31], reducing the verification of (dynamic) session properties, such as deadlock freedom, to (static) type-checking of host programs.

Both MPST and Reowolf’s PDL specify communications via nondeterministic, message-passing programs. Both paradigms also let (control) messages keep peers’ nondeterministic choices aligned: senders choose and recipients follow. But where MPSTs usually specify asynchronous messaging, and are statically decomposed from a global specification, PDL protocols specify synchronous messaging, and are dynamically composed throughout the session. Reowolf has more similarities to variants of MPSTs which explore these features. For example, [43] dynamically coordinates (untyped) Python programs, and [5] adopts synchronous coordination. We expect MPST to inspire applications of PDL in the future, particularly for common uses of MPST: static program analysis, generation, and transformation.

Publish / Subscribe Protocols The publish/subscribe paradigm characterises a family of decentralised network protocols (such as XMPP [29] or MQTT [49]) intended for lightweight communications between IoT devices. For instance, in MQTT, users can (un)subscribe to topics, or publish data to a topic, which persistent, distributed, automated *broker* agents disseminate to subscribers. Likewise, PDL runtimes coordinate user communications. However, because PDL specifies the synchronous inter-dependencies of messages, PDL affords users specifying the ordering and data inter-dependencies between multiple parties’ messages; *e.g.*, PDL affords the definition of distributed transactions.

7 Conclusion

In this article, we formalised the essential contributions of the Reowolf project, and demonstrated the promise of Reowolf connectors as a network API for multiparty internet applications. Like the classic BSD-style sockets, connectors pass messages between users over the network. Unlike sockets, users control session

behaviour by continuously refining the session protocol, which is expressed in Reowolf’s Protocol Description Language (PDL). Consequently, two concepts coincide in PDL protocols: (1) user specifications of the session behaviour, and (2) executable programs delegated by users to the runtime system.

Our article contributes formal definitions of (key properties of) PDL, specifies (key requirements on) the implementation of the connector runtime, and explains their connection. We show how PDL’s dual semantic notions of protocol behaviour support the complex requirements on connectors. PDL is declarative, as it lets users (de)compositionally express and reason about the behaviour *accepted* by (their own) parts of the session protocol. But PDL is also imperative, as PDL-runtime systems can *construct* behaviour one synchronous round at a time, by executing the session protocol, throughout its refinement by the users.

These contributions lay the groundwork for future work to (re)define particular PDL protocols and runtimes, and to rigorously formalise and prove their properties. We identified particularly promising future directions. For example, to verify the correctness of the connector runtime, we must first verify the correctness of its underlying round-search algorithm. These efforts contribute to the greater vision of extending the rigour and programmability of formal protocol languages to the decentralised and dynamic world of internet programming.

Acknowledgements. Benjamin Lion was part of the Cert-T project, funded by the European MSCA-PF grant agreement 101153247. Christopher Esterhuyse was funded by the projects AMdEX-fieldlab (Kansen Voor West EFRO grant KVV00309) and AMdEX-DMI (Dutch Metropolitan Innovations ecosystem for smart and sustainable cities, made possible by the Nationaal Groeifonds).

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.* **14**(3), 329–366 (2004). <https://doi.org/10.1017/S0960129504004153>, <https://doi.org/10.1017/S0960129504004153>
2. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*. *Lecture Notes in Computer Science*, vol. 7000, pp. 169–206. Springer (2011). https://doi.org/10.1007/978-3-642-24933-4_9, https://doi.org/10.1007/978-3-642-24933-4_9
3. Arbab, F.: Proper protocol. In: Ábrahám, E., Bonsangue, M.M., Johnsen, E.B. (eds.) *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. *Lecture Notes in Computer Science*, vol. 9660, pp. 65–87. Springer (2016). https://doi.org/10.1007/978-3-319-30734-3_7, https://doi.org/10.1007/978-3-319-30734-3_7
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.* **61**(2), 75–113 (2006). <https://doi.org/10.1016/j.scico.2005.10.008>, <https://doi.org/10.1016/j.scico.2005.10.008>

5. Bejleri, A., Yoshida, N.: Synchronous multiparty session types. In: Vasconcelos, V.T., Yoshida, N. (eds.) *Proceedings of the First Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@DisCoTec 2008*, Oslo, Norway, June 7, 2008. *Electronic Notes in Theoretical Computer Science*, vol. 241, pp. 3–33. Elsevier (2008). <https://doi.org/10.1016/J.ENTCS.2009.06.002>, <https://doi.org/10.1016/j.entcs.2009.06.002>
6. Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous programming with events and relations: the signal language and its semantics. *Science of computer programming* **16**(2), 103–149 (1991)
7. Berry, G., Gonthier, G.: The esternel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.* **19**(2), 87–152 (1992). [https://doi.org/10.1016/0167-6423\(92\)90005-V](https://doi.org/10.1016/0167-6423(92)90005-V), [https://doi.org/10.1016/0167-6423\(92\)90005-V](https://doi.org/10.1016/0167-6423(92)90005-V)
8. Berry, G., Serrano, M.: Hiphop. js:(a) synchronous reactive web programming. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 533–545 (2020)
9. Bonofiglio, G., Iovinella, V., Lospoto, G., Battista, G.D.: Kathará: A container-based framework for implementing network function virtualization and software defined networks. In: *2018 IEEE/IFIP Network Operations and Management Symposium, NOMS 2018*, Taipei, Taiwan, April 23–27, 2018. pp. 1–9. IEEE (2018). <https://doi.org/10.1109/NOMS.2018.8406267>, <https://doi.org/10.1109/NOMS.2018.8406267>
10. Cheatham, T.E., Fahmy, A.F., Stefanescu, D.C., Valiant, L.G.: Bulk synchronous parallel computing-a paradigm for transportable software. In: *28th Annual Hawaii International Conference on System Sciences (HICSS-28)*, January 3–6, 1995, Kihei, Maui, Hawaii, USA. pp. 268–275. IEEE Computer Society (1995). <https://doi.org/10.1109/HICSS.1995.375451>, <https://doi.org/10.1109/HICSS.1995.375451>
11. Chen, R., Balzer, S., Toninho, B.: Ferrite: A judgmental embedding of session types in rust. In: Ali, K., Vitek, J. (eds.) *36th European Conference on Object-Oriented Programming, ECOOP 2022*, June 6–10, 2022, Berlin, Germany. *LIPICs*, vol. 222, pp. 22:1–22:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPICs.ECOOP.2022.22>, <https://doi.org/10.4230/LIPICs.ECOOP.2022.22>
12. Clarke, L., Glendinning, I., Hempel, R.: The mpi message passing interface standard. In: *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3*, April 25–29, 1994. pp. 213–218. Springer (1994)
13. Cledou, G., Edixhoven, L., Jongmans, S., Proença, J.: API generation for multiparty session types, revisited and revised using scala 3. In: Ali, K., Vitek, J. (eds.) *36th European Conference on Object-Oriented Programming, ECOOP 2022*, June 6–10, 2022, Berlin, Germany. *LIPICs*, vol. 222, pp. 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPICs.ECOOP.2022.27>, <https://doi.org/10.4230/LIPICs.ECOOP.2022.27>
14. Dokter, K., Arbab, F.: Treo: Textual syntax for reo connectors. In: Bliudze, S., Bensalem, S. (eds.) *Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS 2018*, Thessaloniki, Greece, 15th April 2018. *EPTCS*, vol. 272, pp. 121–135 (2018). <https://doi.org/10.4204/EPTCS.272.10>, <https://doi.org/10.4204/EPTCS.272.10>
15. Dokter, K., Lion, B., Arbab, F., Smeyers, M., Mirlou, A., Esterhuyse, C.A.: Reo Language Compiler (2018), <https://github.com/ReoLanguage/Reo>

16. Edwards, S.: ESUIF: an open estereel compiler. In: Maraninchi, F., Girault, A., Rutten, É. (eds.) *Synchronous Languages, Applications, and Programming*, SLAP 2002, Satellite Event of ETAPS 2002, Grenoble, France, April 13, 2002. *Electronic Notes in Theoretical Computer Science*, vol. 65, p. 79. Elsevier (2002). [https://doi.org/10.1016/S1571-0661\(05\)80442-6](https://doi.org/10.1016/S1571-0661(05)80442-6), [https://doi.org/10.1016/S1571-0661\(05\)80442-6](https://doi.org/10.1016/S1571-0661(05)80442-6)
17. Edwards, S.A.: An estereel compiler for large control-dominated systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **21**(2), 169–183 (2002). <https://doi.org/10.1109/43.980257>, <https://doi.org/10.1109/43.980257>
18. Edwards, S.A., Zeng, J.: Code generation in the columbia estereel compiler. *EURASIP J. Embed. Syst.* **2007** (2007). <https://doi.org/10.1155/2007/52651>, <https://doi.org/10.1155/2007/52651>
19. Esterhuyse, C.A., Hiep, H.A.: Reowolf: Synchronous multi-party communication over the internet. In: Arbab, F., Jongmans, S. (eds.) *Formal Aspects of Component Software - 16th International Conference, FACS 2019, Amsterdam, The Netherlands, October 23-25, 2019, Proceedings. Lecture Notes in Computer Science*, vol. 12018, pp. 235–242. Springer (2019). https://doi.org/10.1007/978-3-030-40914-2_12, https://doi.org/10.1007/978-3-030-40914-2_12
20. Esterhuyse, C.A., Hiep, H.D.A.: Reowolf 1.0 project deliverables. Record on Zenodo (2024), <https://zenodo.org/records/10838450>
21. Florence, S.P., You, S., Tov, J.A., Findler, R.B.: A calculus for estereel: if can, can. if no can, no can. *Proc. ACM Program. Lang.* **3**(POPL), 61:1–61:29 (2019). <https://doi.org/10.1145/3290374>, <https://doi.org/10.1145/3290374>
22. Gautier, T., Le Guernic, P., Besnard, L.: *Signal: A declarative language for synchronous programming of real-time systems*. Springer (1987)
23. Gerbessiotis, A.V., Valiant, L.G.: Direct bulk-synchronous parallel algorithms. *J. Parallel Distributed Comput.* **22**(2), 251–267 (1994). <https://doi.org/10.1006/JPDC.1994.1085>, <https://doi.org/10.1006/jpdc.1994.1085>
24. Gropp, W.: MPI (message passing interface). In: Padua, D.A. (ed.) *Encyclopedia of Parallel Computing*, pp. 1184–1190. Springer (2011). https://doi.org/10.1007/978-0-387-09766-4_222, https://doi.org/10.1007/978-0-387-09766-4_222
25. Halbwachs, N.: A synchronous language at work: the story of lustre. In: 3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings. pp. 3–11. IEEE Computer Society (2005). <https://doi.org/10.1109/MEMCOD.2005.1487884>, <https://doi.org/10.1109/MEMCOD.2005.1487884>
26. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. *Proc. IEEE* **79**(9), 1305–1320 (1991). <https://doi.org/10.1109/5.97300>, <https://doi.org/10.1109/5.97300>
27. Honda, K.: Types for dyadic interaction. In: *International Conference on Concurrency Theory*. pp. 509–523. Springer (1993)
28. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. pp. 273–284. ACM (2008). <https://doi.org/10.1145/1328438.1328472>, <https://doi.org/10.1145/1328438.1328472>
29. Hornsby, A., Walsh, R.: From instant messaging to cloud computing, an xmpp review. In: *IEEE international symposium on consumer electronics (ISCE 2010)*. pp. 1–6. IEEE (2010)

30. Hu, F., Hao, Q., Bao, K.: A survey on software-defined network and openflow: From concept to implementation. *IEEE Commun. Surv. Tutorials* **16**(4), 2181–2206 (2014). <https://doi.org/10.1109/COMST.2014.2326417>, <https://doi.org/10.1109/COMST.2014.2326417>
31. Imai, K., Lange, J., Neykova, R.: KmcLib: Automated inference and verification of session types from ocaml programs. In: Fisman, D., Rosu, G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 13243, pp. 379–386. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_20, https://doi.org/10.1007/978-3-030-99524-9_20
32. Jongmans, S.T.Q., Arbab, F.: Can high throughput atone for high latency in compiler-generated protocol code? In: Dastani, M., Sirjani, M. (eds.) *Fundamentals of Software Engineering - 6th International Conference, FSEN 2015 Tehran, Iran, April 22-24, 2015, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 9392, pp. 238–258. Springer (2015). https://doi.org/10.1007/978-3-319-24644-4_17, https://doi.org/10.1007/978-3-319-24644-4_17
33. Jongmans, S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Automatic code generation for the orchestration of web services with reo. In: Paoli, F.D., Pimentel, E., Zavattaro, G. (eds.) *Service-Oriented and Cloud Computing - First European Conference, ESOC 2012, Bertinoro, Italy, September 19-21, 2012. Proceedings. Lecture Notes in Computer Science*, vol. 7592, pp. 1–16. Springer (2012). https://doi.org/10.1007/978-3-642-33427-6_1, https://doi.org/10.1007/978-3-642-33427-6_1
34. Jongmans, S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Orchestrating web services using reo: from circuits and behaviors to automatically generated code. *Serv. Oriented Comput. Appl.* **8**(4), 277–297 (2014). <https://doi.org/10.1007/S11761-013-0147-1>, <https://doi.org/10.1007/s11761-013-0147-1>
35. Kraska, T.: Northstar: An interactive data science system. *Proc. VLDB Endow.* **11**(12), 2150–2164 (2018). <https://doi.org/10.14778/3229863.3240493>, <http://www.vldb.org/pvldb/vol11/p2150-kraska.pdf>
36. Krause, C., Giese, H., Vink, de, E.: Compositional and behavior-preserving reconfiguration of component connectors in reo. *Journal of Visual Languages and Computing* **24**(3), 153–168 (2013). <https://doi.org/10.1016/j.jvlc.2012.09.002>
37. Krause, C.: *Reconfigurable Component Connectors*. Ph.D. thesis, Leiden University (2011)
38. Krause, C., Costa, D., Proença, J., Arbab, F.: Reconfiguration of reo connectors triggered by dataflow. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* **10** (2008)
39. Krause, C., Maraïkar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in reo using high-level replacement systems. *Sci. Comput. Program.* **76**(1), 23–36 (2011). <https://doi.org/10.1016/j.scico.2009.10.006>, <https://doi.org/10.1016/j.scico.2009.10.006>
40. Lara, A., Kolasani, A., Ramamurthy, B.: Network innovation using openflow: A survey. *IEEE communications surveys & tutorials* **16**(1), 493–512 (2013)
41. Lindley, S., Morris, J.G.: Embedding session types in haskell. In: Mainland, G. (ed.) *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. pp. 133–145. ACM (2016). <https://doi.org/10.1145/2976002.2976018>, <https://doi.org/10.1145/2976002.2976018>

42. McKeown, N., Anderson, T.E., Balakrishnan, H., Parulkar, G.M., Peterson, L.L., Rexford, J., Shenker, S., Turner, J.S.: Openflow: enabling innovation in campus networks. *Comput. Commun. Rev.* **38**(2), 69–74 (2008). <https://doi.org/10.1145/1355734.1355746>, <https://doi.org/10.1145/1355734.1355746>
43. Neykova, R.: Session types go dynamic or how to verify your python conversations. In: Yoshida, N., Vanderbauwhede, W. (eds.) *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013. EPTCS*, vol. 137, pp. 95–102 (2013). <https://doi.org/10.4204/EPTCS.137.8>, <https://doi.org/10.4204/EPTCS.137.8>
44. Overbeek, R., Endrullis, J., Rosset, A.: Graph rewriting and relabeling with pbpo⁺: A unifying theory for quasitoposes. *J. Log. Algebraic Methods Program.* **133**, 100873 (2023). <https://doi.org/10.1016/J.JLAMP.2023.100873>, <https://doi.org/10.1016/j.jlamp.2023.100873>
45. Potop-Butucaru, D., Edwards, S.A., Berry, G.: *Compiling esterel*, vol. 86. Springer Science & Business Media (2007)
46. Proença, J., Clarke, D., de Vink, E.P., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: Ossowski, S., Lecca, P. (eds.) *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*. pp. 1510–1515. ACM (2012). <https://doi.org/10.1145/2245276.2232017>, <https://doi.org/10.1145/2245276.2232017>
47. Proença, J., Cledou, G.: Arx: reactive programming for synchronous connectors. In: *Coordination Models and Languages: 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 22*. pp. 39–56. Springer (2020)
48. da Rosa Righi, R., de Quadros Gomes, R., Rodrigues, V.F., da Costa, C.A., Alberti, A.M.: Migbsp++: Improving process rescheduling on bulk-synchronous parallel applications. In: *12th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2015, Marrakech, Morocco, November 17-20, 2015*. pp. 1–8. IEEE Computer Society (2015). <https://doi.org/10.1109/AICCSA.2015.7507256>, <https://doi.org/10.1109/AICCSA.2015.7507256>
49. Soni, D., Makwana, A.: A survey on mqtt: a protocol of internet of things (iot). In: *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*. vol. 20, pp. 173–177 (2017)
50. Valiant, L.G.: A bridging model for parallel computation. *Commun. ACM* **33**(8), 103–111 (1990). <https://doi.org/10.1145/79173.79181>, <https://doi.org/10.1145/79173.79181>
51. Wang, E., Barve, Y., Gokhale, A., Sun, H.: Dynamic resource management for cloud-native bulk synchronous parallel applications. In: *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. pp. 152–157 (2023). <https://doi.org/10.1109/ISORC58943.2023.00028>
52. Zhao, X., Papagelis, M., An, A., Chen, B.X., Liu, J., Hu, Y.: Zipline: an optimized algorithm for the elastic bulk synchronous parallel model. *Mach. Learn.* **110**(10), 2867–2903 (2021). <https://doi.org/10.1007/S10994-021-06064-W>, <https://doi.org/10.1007/s10994-021-06064-w>

A PDL (Runtime) Formalisation in Coq vs. this Article

The definitions in this article correspond to those of our artefact, which is machine-checked with the Coq proof assistant. The repository at <https://zenodo.org/records/14936561> includes the Coq file itself `reowolf_formalism.v`, as well as instructions for installing Coq locally or via a Docker image.

Figure 4 shows the *parameters* of our formalisation. As our definitions are otherwise constructive, these separate our specification from an (executable) implementation. The parametrisation of \mathcal{P} and \mathcal{V} is desirable, affording the choice of any concrete port- and variable-types. The remaining three parameters can be understood as *assumptions*, scoping our formalism. Equivalently, these properties are assumed but have no proofs yet. The first two assumptions constrain the choice of \mathcal{P} and \mathcal{V} ; they must form *setoids*, because our proofs rely on being able to decide (in)equality of port pairs and variable pairs. The final assumption requires the refinement of the memory-storage to some finite map structure, for example, implemented via Coq’s inbuilt association lists or AVL trees.

Figure 5 shows the correspondences between the definitions in this article to those in the artefact, *e.g.*, to help readers inspect the relevant Coq definitions.

in natural language	in Section 3 notation	definition in the artefact
• the port type	\mathcal{P}	<code>port</code> in line 77
• the variable type	\mathcal{V}	<code>var</code> in line 77
• decidable port equality	$\forall p\ p' : \mathcal{P}, p = p' \vee p \neq p'$	<code>eq_dec_port</code> in line 78
• decidable variable equality	$\forall v\ v' : \mathcal{V}, v = v' \vee v \neq v'$	<code>eq_dec_var</code> in line 79
• decidable memory equality	$\forall \sigma\ \sigma' : \Sigma, \sigma = \sigma' \vee \sigma \neq \sigma'$	<code>eq_dec_memory</code> in line 146

Fig. 4. Parameters or assumptions of the formalism in our artefact at <https://zenodo.org/records/14936561>. Each occurs in the artefact with the `Parameter` keyword.

Term in article	Definition in Artefact
• \mathcal{D} in Section 2	<code>data</code> in line 71
• \mathcal{P} in Section 2	<code>port</code> in line 77
• \mathcal{V} in Section 2	<code>var</code> in line 77
• \mathcal{M} in Section 2	<code>msg</code> in line 82
• \leq in Definition 5	<code>leq_msg</code> in line 93
• \mathcal{E} in Figure 1	<code>expression</code> in line 114
• \mathcal{S} in Figure 1	<code>statement</code> in line 121
• Σ in Section 3.1	<code>memory</code> in line 142
• \mathcal{W} in Section 3.1	<code>worker</code> in line 152
• <i>eval</i> declared in the Figure 2 caption	<code>eval</code> in line 177
• Δ in Section 3.1	<code>synchronicity</code> in line 189
• <i>update</i> in Figure 2	<code>update</code> in line 223
• Definition 6	<code>growing_msglist</code> in line 572
• Definition 8	<code>constant_msglist</code> in line 585
• $W \xRightarrow{m} W'$ in Definition 1	<code>sync</code> in line 708
• $W \xrightarrow{m} W'$ in Definition 1	<code>async</code> in line 717
• Lemma 1	<code>asynchronous_step_determinism</code> in line 950
• $W \xrightarrow{*} W'$ in Section 3.3	<code>apath</code> in line 975
• Definition 3	<code>round</code> in line 988
• Definition 4	<code>round_final_m</code> in line 993
• Lemma 2	<code>per_growing_round_a_cst</code> in line 1677
• Lemma 3	<code>cst_confluence</code> in line 1468
• Theorem 1	<code>round_determinism</code> in line 1712
• Theorem 2	<code>compositionality</code> in line 1950
• Theorem 3	<code>decompositionality</code> in line 2090
• Definition 9	<code>offered</code> in line 2134
• Definition 10	<code>constructive_round</code> in line 2170
• Lemma 4	<code>const_constructability</code> in line 2454
• 2^S in Figure 1	<code>protocol</code> in line 2512
• Definition 7	<code>accepts</code> in line 2537
• Definition 11	<code>constructs</code> in line 2541
• Theorem 4	<code>acceptance_vs_construction</code> in line 2735
• PDL runtime $\{\mathcal{C}, start, proto, inject,$ and <i>run</i> $\}$ signature in Section 4.1	<code>PdlRuntime</code> record with fields <code>{config, start,</code> <code>proto, inject, run}</code> in line 2752
• Property 1	<code>initially_trivial_protocol</code> in line 2763
• Property 2	<code>injection_composes</code> in line 2767
• Property 3	<code>soundness</code> in line 2774
• Property 4	<code>completeness</code> in line 2788
• <i>Silent PDL runtime</i> in Section 4.2	<code>silent_pdl_runtime</code> in line 2763
• Proofs of Properties 1 to 3 and in- verse of Property 4 for the silent PDL runtime in Section 4.2 (only their existence is claimed)	<code>silent_initially_trivial_protocol</code> at line 2817, <code>silent_injection_composes</code> at line 2824, <code>silent_soundness</code> at line 2840, and <code>silent_incompleteness</code> at line 2850 (resp.).

Fig. 5. Correspondences between the terms in this article and the definitions and theorems in the artefact at <https://zenodo.org/records/14936561>.