Reowolf: Executable, Compositional, Synchronous Protocol Specifications

Christopher A. Esterhuyse, Benjamin Lion, Hans-Dieter A. Hiep, Farhad Arbab

Abstract

Low-level communication primitives such as BSD sockets are not adequate for next-generation Internet applications. Instead, we propose programmable connectors that declare high-level, application-specific communication intent using a compositional, formal protocol description language suitable for verification. This paper contributes the Protocol Description Language (PDL), that has a formal compositional semantics and is executable as witnessed by a distributed, dynamically (re)configureable run-time interpreter.

Introduction

Currently, networks of computing systems operate by *de facto* conventions. Applications make use of informally specified protocol stacks that are implemented within operating systems to enable peer-to-peer inter-process communication. This includes applications deployed on a single machine, in local area networks, and on the global Internet. Realistic protocol stacks are large and complex, e.g. various application layer protocols (e.g. BGP, DNS, HTTP, FTP) are on top of transport layer protocols (e.g. SCTP, TCP, TLS, UDP) on top of Internet protocols (e.g. IPv4, IPv6).

Virtually all Internet applications use a decades-old BSD socket application programming interface (API). However, application protocols implemented on top of sockets are often not transparent and lack rigorous standards. Network middleware must resort to guess the high-level application intent hidden by sockets [21]. Furthermore, high-level security properties (e.g. kill-switch absence) are obfuscated by the tight coupling between an application's state and its socket communications. As a result, applications are difficult and costly to analyze, impeding the availability of protocol implementations with desirable qualities. To avoid this friction, application developers favor centralized application architectures over decentralized architectures.

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Presently, we introduce *Reowolf connectors* as an alternative to BSD sockets for realizing multi-party, synchronous communication sessions between networked applications. This model lets application programmers express intended behavior at a higher level of abstraction, thereby abstracting from complex low-level implementation details such as those of distributed consensus algorithms. Reowolf connectors offer an API that lets application programmers declaratively specify their communication intent using the Protocol Description Language (PDL), delegating the implementation of protocols to the operating system and networking environment. One of our design goals for PDL is to make the formal verification of high-level security properties tractable. Thus, PDL needs a compositional, formal meaning that can be analyzed with mathematical rigor.

There is a working prototype of Reowolf connectors implemented in Rust at the user-mode level, which includes a run-time interpreter of PDL, available in a persistent Zenodo repository [1]. The goal of this article is to present interesting theoretical aspects of our work and to give a formal basis for the aforementioned implementation. In particular, the theoretical contributions of this article include:

1. We define the Protocol Description Language (PDL) intended for formally and unambiguously specifying the behavior of network protocols. The design of PDL is heavily based on the Reo coordination language [2, 3, 17, 19, 28] but differs at crucial points. (Section 1)
2. We give a formal but idealistic semantics, by assuming the availability of *oracles*. We give semantics in two ways: an operational semantics and a denotational semantics. The operational semantics is the most natural semantics of the language, and the denotational semantics witnesses that our semantics is compositional. We show the equivalence of these two. (Section 2)
3. Towards an implementation of a run-time interpreter of PDL, we eliminate the need for oracles from the semantics and instead give a realistic semantics for a fragment of PDL that is suitable for incrementally unfolding a protocol's behavior. This semantics shows that, for certain protocols, we can effectively generate oracles on-the-fly. We show how this third semantics relates to the first two. (Section 3)

We then introduce connectors as a replacement for sockets for multi-party network programming and describe *in abstracto* our prototype implementation in Section 4. The prototype generates communication behavior from protocols, specified just-in-time, by distributed applications connected by the Internet. The rest of the paper reflects on our contributions: Sections 5 and 6 evaluate the properties of connectors and PDL by their own merits, and in comparison to related work, respectively. Section 7 concludes with a summary.

1 Protocol Description Language

In this section, we give an account of a *formal* protocol and introduce the syntax and semantics for the Protocol Description Language (PDL). The core idea is that formal protocols can be defined in terms of *components*. In general, we distinguish two types of components: *protocol components* that are specified in PDL, and *native components* which are given a fixed interpretation. An example of a native component is an IP component that offers connectivity over the Internet, or a clock component that independently tracks time.

Components exchange data with each other via shared *ports*. The ports through which a component can exchange data define the *interface* of that component. For example, the native IP component has an interface consisting of ports through which IP packets are exchanged, and a clock component has a port through which the current clock value is exchanged. For the remainder, by *protocol* we mean a set of (interacting) components, and we say that these components are *composed* together to form the protocol.

We discuss a few core design principles of PDL. First, a component is intentionally not aware of the other components with which it composes into a protocol. That is, the behavior of an individual component cannot depend on particular intentional properties of the other components with which it is composed. When two components are composed, only the behavior that both individual components share is permissible: but neither component can inspect the other component by means other than data exchanged through their interface. Second, the coordination of data exchange is explicit and exogenous to components. This leads us, for each component, to be able to recognize a trace of observable behavior, that is the data exchanged at ports over time. A component can be analyzed and its properties verified on its own, independently of its context.

The most natural way to give meaning to a composition of components is to intersect their individual behavior: two components form a new component which restricts the behavior of its underlying components to the largest common subset, i.e., the intersection. Additionally, we express *hiding* of a port on a component as a unary operator that removes that port from the interface. The resulting component accepts as behavior anything that the original component accepts, but ignoring the data exchanged on the hidden port. The conformance of an application to a protocol is equivalent to asking if the intersection of the protocol component with the native component that represents the behavior of the application is non-empty. Of course, other semantic operations may be of interest. In the following, we syntactically describe protocol components expressed in PDL with intersection as composition operation.

Syntax. Let V be a set of variables with typical element x. Let P be a set of port variables with typical element p. We assume V and P are disjoint. The abstract syntax for our protocol description language with two syntactical categories for components (C) and statements (S) is defined:

$$
\begin{aligned}
C &::= S \mid C \cap C \mid \exists p.C \\
S &::= \textbf{skip} \mid x := e \mid x \leftarrow p \mid x \rightarrow p \mid \textbf{assert } b \mid \textbf{sync} \\
&\quad \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \textbf{ od} \mid S \,;\, S
\end{aligned}
$$

where e stands for an expression; b represents the usual Boolean expressions over variables in V extended with the novel *firing* operator $¿p$ for a port p. The inverted question mark symbol ¿ is used as a prefix unary operator, because if $¿p$ is true it anticipates that the port fires. Intuitively, the operations $x \leftarrow p$ and $x \rightarrow p$ model a *get* and *put* operation on port p, respectively. While the state variables x refer to standard memory locations in a component state, a port variable p refers to a shared store between components. The firing operator $¿p$ acts as a condition on the current value at port b. The *sync* operation enforces all ports to have the same value, which, if successful, acts as a reset operation on the port's value. The interface $\mathcal{I}(c)$ of a component c is the collection of all free port variables occurring in its program. We refer to $\mathcal{I}_p(c)$ and $\mathcal{I}_g(c)$ for the set of ports on which component c puts (output ports) and gets (input ports), respectively. An existential free component can be written as the intersection of statements only, i.e., $C = S_1 \cap \ldots \cap S_n$ for $n \in \mathbb{N}$. In that case, we call such component C a *composite* and all components S_i for $1 \leq i \leq n$ its *primitives.*

Example. Consider the *voting* component $A(p_A, q_A, n, R)$:

```
while true do $n \rightarrow p_A$ ;
   if $¿q_A$ then $x \leftarrow q_A$ else skip fi ;
   if $j = R$ then $j := 0$ ; $n := 1 - n$ else $j := j + 1$ fi ;
sync od
```

with $n \in \{1, 0\}$ and x initialized to 0. Component A selects a vote n, and keeps on voting the same value N times, and then flips its vote. Individually, component A exhibits streams of bits at port p_A, that consist of a sequence of R repetitions of n, followed by R repetitions of $1 - n$, etc.

Consider the following comparison component $U(i_1, i_2, o)$:

```
while true do
  if $¿i_1 \land ¿i_2$ then $x_1 \leftarrow i_1$ ; $x_2 \leftarrow i_2$ ;
       if $x_1 = x_2$ then $x_1 \rightarrow o$ else skip fi
  else if $¿i_1 \land \neg¿i_2$ then $x_1 \leftarrow i_1$ ; $x_1 \rightarrow o$ else skip fi
       if $¿i_2 \land \neg¿i_1$ then $x_2 \leftarrow i_2$ ; $x_2 \rightarrow o$ else skip fi
  fi ; sync od
```

```
$Rep(p,q,r)$ = while true do
                   if $¿p$ then $x \leftarrow p$; $x \rightarrow q$; $x \rightarrow r$
                   else assert $\neg ¿q \wedge \neg ¿r$ fi; sync
               od
$Same(p,q,r)$ = while true do
                   if $¿p$ then
                      $x \leftarrow p$; $y \leftarrow q$; assert $p = q$; $x \rightarrow r$
                   else assert $\neg ¿q \wedge \neg ¿r$ fi; sync
               od
```

Figure 1. Definition of protocols *Rep* and *Same*, parametric over ports p, q, and r.

Component U compares the values at its ports i_1 and i_2, and outputs on o the value of both ports if they fired with the same value, or if only one port fires, of that firing port. We consider the expression $M(p_{A_1}, o_1, ..., p_{A_k}, o_k, o)$, for $k > 1$, defined as:

$$U(p_{A_k}, o_k, o) \cap U(p_{A_1}, p_{A_2}, o_1) \cap \bigcap_{1 \leq i \leq k-1} U(p_{A_i}, o_i, o_{i+1})$$

M is the composite of a series of U components, each casting the result of its comparison to a next comparison unit. As a result, if it fires, port o contains the outcome of the majority of the votes among voters $A_1, ..., A_k$. Finally, we write composition of voters with the voting protocol as:

$$M(p_{A_1}, o_1, ..., p_{A_k}, o_k, o) \cap \bigcap_{1 \leq i \leq k} A_i(p_{A_i}, o, b_i, R_i)$$

where $b_i \in \{0, 1\}$ is the vote of component A_i, and R_i is its repetition.

We make several observations. The protocol is not centralized, and is defined in terms of several comparison units. The voters do not have access to votes of other voters, but can speculate on the result of the vote before voting. Each unit may just compare the votes, without necessarily accessing the value of the vote. One may therefore employ some encryption and decryption protocols to make the voting protocol completely private.

2 Idealistic Semantics

Consider the voting protocol detailed in Section 1 instantiated for three voters: Alice, Bob, and Dan. Three rounds of votes of each voters are recorded in Table 1. Note that, individually, each pair of ports reflects the behavior of a voter, i.e., its vote and the result. A cell in Table 1 consists of the value of a port in a round, e.g., port p_A at round 1 has value 0, and port p_C at round 2 has value 1. The property of synchrony imposed by the majority protocol induces a relation among the cells in Table 1. In every round, the output of the vote is the value given by the majority vote, i.e., the port q_A, q_B, and q_C always output the same value, which is the majority of the votes at p_A, p_B, and p_C. We offer a semantics for which each component denotes a set of such tables, and where a composite component restricts which of individual tables are allowed.

	p_A	q_A	p_B	q_B	p_C	q_C
1	0	0	1	0	0	0
2	1	1	1	1	1	1
3	1	1	0	1	1	1
...	...		...		...	

Table 1. Three round of votes and results for Alice, Bob, and Dan, respectively with interface (p_A, q_A), (p_B, q_B), and (p_C, q_C).

$$(\sigma, s, \textbf{skip}) \longrightarrow (\sigma, s, \checkmark)$$
$$(\sigma, s, x := e) \longrightarrow (\sigma, s[x := [\![e]\!](s)], \checkmark)$$
If $\sigma(0)(p) \neq \star$:
$$(\sigma, s, x \leftarrow p) \longrightarrow (\sigma, s[x := \sigma(0)(p)], \checkmark)$$
If $\sigma(0)(p) = s(x)$:
$$(\sigma, s, p \leftarrow x) \longrightarrow (\sigma, s, \checkmark)$$
If $[\![b]\!](\sigma(0), s) = \textbf{true}$:
$$(\sigma, s, \textbf{assert } b) \longrightarrow (\sigma, s, \checkmark)$$
$$(\sigma, s, \textbf{sync}) \xrightarrow{\checkmark} (\sigma', s, \checkmark)$$
If $[\![b]\!](\sigma(0), s) = \textbf{true}$:
$$(\sigma, s, \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) \longrightarrow (\sigma, s, S_1)$$
If $[\![b]\!](\sigma(0), s) = \textbf{false}$:
$$(\sigma, s, \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) \longrightarrow (\sigma, s, S_2)$$
If $[\![b]\!](\sigma(0), s) = \textbf{true}$:
$$(\sigma, s, \textbf{while } b \textbf{ do } S \textbf{ od}) \longrightarrow (\sigma, s, S \ ; \textbf{while } b \textbf{ do } S \textbf{ od})$$
If $[\![b]\!](\sigma(0), s) = \textbf{false}$:
$$(\sigma, s, \textbf{while } b \textbf{ do } S \textbf{ od}) \longrightarrow (\sigma, s, \checkmark)$$

Figure 2. Operational semantics for PDL.

Operational semantics. Consider a value domain $\mathcal{D}$. By $O(\mathcal{D})$ we denote the value domain extended by a special element $\star$ that represents the absence of a value. Let Σ denote $V \rightarrow \mathcal{D}$, and Δ denote $\mathbb{N} \rightarrow P \rightarrow O(\mathcal{D})$. Here, Σ is the set of all (internal) states with typical element s, and Δ the set of all (observable) streams with typical element σ. We define $s[x := v]$ as the usual state update for state s, variable x and value $v \in \mathcal{D}$. We define σ' as the tail of σ, i.e., $\sigma'(x) = \sigma(x+1)$ for all $x \in \mathbb{N}$. Further, we assume that $[\![e]\!] : \Sigma \rightarrow \mathcal{D}$ is defined for every expression e, and that $[\![b]\!] : (P \rightarrow O(\mathcal{D})) \times \Sigma \rightarrow \{\textbf{true}, \textbf{false}\}$ is defined compositionally for Boolean expressions b, where we have $[\![¿p]\!](a,s) = \textbf{true}$ if $a(p) \neq \star$, and $[\![¿p]\!](a,s) = \textbf{false}$ if $a(p) = \star$.

We show in Figure 2 a small-step operational semantics for statements. We consider an inductively defined labeled relation between the triples of the form (σ, s, S) where, in the place of S we may have a check mark ✓ to indicate termination, and the label on the relation is either a check mark ✓ or empty.

Further, the small-step relation is closed under the following rule:

$$\frac{(\sigma_1, s_1, S_1) \xrightarrow{X} (\sigma_2, s_2, S_1')}{(\sigma_1, s_1, S_1 \ ; S_2) \xrightarrow{X} (\sigma_2, s_2, S_1' \ ; S_2)}$$

where we identify ✓ ; S_2 and S_2.

We now consider (finite or infinite) chains of triples where each pair of the successive elements is related by the above relation. We write $(\sigma, s, S) \downarrow$ if there exists a chain with (σ, s, S) as its first triple and either (1) the chain is finite and its last triple has a check mark in the third place, or (2) it has infinitely many triples with a check mark in the third place. Intuitively, such a finite chain represents a terminating run, and such an infinite chain represents a run that performs **sync** infinitely often.

A component denotes a set of streams, i.e. $[\![C]\!] \subseteq \Delta$. Let $\sigma \downarrow q$ be the oracle such that for all $i \in \mathbb{N}$, $(\sigma \downarrow q)(i)(p) = \star$ if $p = q$ and equals $\sigma(i)(p)$ otherwise. Let $\iota \in \Sigma$ denote some fixed initial but unspecified state. We define the semantics as follows:

$$\begin{aligned}
[\![C_1 \cap C_2]\!] &= [\![C_1]\!] \cap [\![C_2]\!] \\
[\![\exists p.C]\!] &= \{\tau \in \Delta \mid \exists \sigma \in [\![C]\!].(\tau \downarrow q) = (\sigma \downarrow q)\} \\
[\![S]\!] &= \{\sigma \mid (\sigma, \iota, S) \downarrow\}
\end{aligned}$$

Consider the following two statements:

$$\begin{aligned}
\Omega &= \textbf{while true do skip od} \\
\omega &= \textbf{while true do sync od}
\end{aligned}$$

Although both Ω and ω represent infinitely running programs, their denotations are different. Namely, $[\![\Omega]\!] = \emptyset$, since there does not exist a terminating run (**true** is never false in any state, thus the loop never exits) nor does it perform **sync** infinitely often. However, $[\![\omega]\!] = \Delta$, i.e., any stream is acceptable: we always have a chain wherein **sync** occurs infinitely often.

Further, consider the replicator protocol depicted in Figure 1. Its denotational semantics is as follows:

$$[\![Rep(p,q,r)]\!] = \{\sigma \mid \forall k.\ \sigma(k)(p) = \sigma(k)(q) = \sigma(k)(r)\}$$

that is, at any time k either the value observed at p is the same as those at q and r, or there is no value observed at any of the ports p, q, and r.

$$\begin{aligned}
[\textbf{skip}] &= \{(\lambda, \sigma, \lambda) \mid \sigma \in \Delta\} \\
[x := e] &= \{((s,i), \sigma, (s[x := [\![e]\!](s)], i)) \mid \sigma \in \Delta\} \\
[x \leftarrow p] &= \{((s,i), \sigma, (s[x := \sigma(i)(p)], i)) \mid \sigma(i)(p) \neq \star\} \\
[x \rightarrow p] &= \{((s,i), \sigma, (s,i)) \mid \sigma(i)(p) = s(x)\} \\
[\textbf{assert } b] &= \{((s,i), \sigma, (s,i)) \mid [\![b]\!](\sigma(i), s) = \textbf{true}\} \\
[\textbf{sync}] &= \{((s,i), \sigma, (s, i+1)) \mid \sigma \in \Delta\} \\
[S_1 \ ; S_2] &= [S_1] \circ [S_2] \cup \{(\lambda, \sigma, \star) \mid (\lambda, \sigma, \star) \in [S_1]\} \\
[\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}] &= [\textbf{assert } b \ ; S_1] \cup [\textbf{assert } \neg b \ ; S_2]
\end{aligned}$$

$$[\textbf{while } b \textbf{ do } S \textbf{ od}] = \bigcup_{k=0}^{\infty} [(\textbf{while } b \textbf{ do } S \textbf{ od})^k] \cup \bigcap_{k=0}^{\infty} \text{prog}(S,k)$$

where

$$[S_1] \circ [S_2] = \{(\lambda, \sigma, \tau) \mid (\lambda, \sigma, \kappa) \in [S_1], (\kappa, \sigma, \tau) \in [S_2]\}$$

$$\begin{aligned}
(\textbf{while } b \textbf{ do } S \textbf{ od})^0 &= \textbf{while true do skip od} \\
(\textbf{while } b \textbf{ do } S \textbf{ od})^{k+1} &= \textbf{if } b \textbf{ then } S \ ; (\textbf{while } b \textbf{ do } S \textbf{ od})^k \\
&\qquad \textbf{else skip fi}
\end{aligned}$$

given $P = \{((s,i), \sigma, (t,i)) \mid i \in \mathbb{N}\}^C$,

$$\text{prog}(S,k) = ([S^{\underline{k}}] \circ \bigcup_{m=1}^{\infty} ([S^{\underline{m}}] \cap P))\dagger$$

and $X\dagger = \{(\lambda, \sigma, \star) \mid (\lambda, \sigma, \tau) \in X\}$ and

$$\begin{aligned}
S^{\underline{0}} &= \textbf{skip} \\
S^{\underline{k+1}} &= \textbf{assert } b \ ; S \ ; S^{\underline{k}}
\end{aligned}$$

Figure 3. Denotational semantics for PDL.

Denotational semantics. We call $C \subseteq (\mathbb{N} \to O(\Sigma)) \times \Delta \times (\mathbb{N} \to O(\Sigma))$ a component, where $(\lambda, \sigma, \tau) \in C$ consists of the stream of initial states λ, the observable behavior σ, and the stream of final states τ. For simplicity, we first consider a subspace of components, namely those that have a single initial state, and an optional final state. We write (s,i) for a stream $\lambda \in \mathbb{N} \to O(\Sigma)$ when $\lambda(j) = s$ if $j = i$, and $\star$ otherwise. We reuse the symbol $\star$ to denote the stream consisting of $\star$ only.

We interpret statements compositionally by defining $[S] \subseteq (\mathbb{N} \to O(\Sigma)) \times \Delta \times (\mathbb{N} \to O(\Sigma))$. We show the equality $[\![S]\!] = \{\sigma \in \Delta \mid ((s,0), \sigma, \tau) \in [S]\}$. In the following, when unspecified, s is a state that ranges over Σ and i is an index that ranges over $\mathbb{N}$. We define $[S]$ on the structure of the statement S in Figure 3.

Intuitively, the component $\bigcup_{k=0}^{\infty} [(\textbf{while } b \textbf{ do } S \textbf{ od})^k]$ contains all streams in the denotation of the k-unfolding of the while statement for some k. Alternatively, the component $\bigcap_{k=0}^{\infty} \text{prog}(S,k)$ contains all streams in the denotation of S^k for all k. The first component represents either runs that terminate and for which there is a witness for k, or runs that enter finitely many times the loop but are non terminating in S. The second component includes runs that enter the loop infinitely many times, but always eventually synchronize.

We show, in Theorem 2.1, that the denotational semantics coincide with the operational semantics.

Theorem 2.1. *For all statements S in PDL:*

$$[\![S]\!] = \{\sigma \in \Delta \mid ((s, 0), \sigma, \tau) \in [S]\}$$

Proof. See appendix. □

Reo. The semantics of PDL is faithful to the semantics of Reo [4, 29]. A component in PDL denotes a set of sequences of port-value assignments, which is analog to a Reo connector that denotes a set of time data stream tuples over its ports, with integer time. As well as in Reo, constraints over port assignments are transitive: if A *always* fires with B and B *always* fires with C, then A *always* fires with C. The idealistic semantics of PDL introduced in this section shares the same declarative paradigm with Reo: the emphasize is on *what* behaviors are specified by the set of interacting components, and not on *how* such behavior is constructed. PDL, however, differs with Reo in that its sequential nature opens a more imperative understanding of protocols. We give, in the next section, an alternative semantics, called realistic semantics, that defines the operational generation of some oracles.

3 Realistic Semantics

Section 1 gives to PDL a semantics as components. A program written in PDL has as meaning a set of streams of port-value assignments. As shown earlier, the semantics is compositional, which is of interest for reasoning about a composite program in terms of its parts. The results of Section 1 come, however, at a price: the operational intentionality of the language is lost. For instance, in the time data streams formalism, the statements $x \leftarrow p$ and $x \rightarrow p$ are semantically equivalent, because they both represent the exchange of a value x through a port p. On the other hand, their operational intentions differ: $x \leftarrow p$ denotes production of the value of x through p, and $x \rightarrow p$ denotes consumption of a value designated as x through p. In this section, we provide a realistic semantics, that closely describes how an implementation may construct some oracle given by the idealistic semantics in Section 1. As expected, there exist oracles in the ideal semantics that cannot be constructed, and some finite runs in the realistic semantics cannot appear in any idealistic semantics. For instance, below, we give a composition that is not *causal* as an example of the former, and a run generated with a one step look-ahead as an example of the latter.

Causality. Intuitively, causality prohibits cyclic dependencies between send and receive operations at a port in a round. Consider the following two component descriptions:

$$C_1 = \textbf{while true do } x \leftarrow p; x \rightarrow q \text{ ; } \textbf{sync od}$$
$$C_2 = \textbf{while true do } y \leftarrow q; y \rightarrow p \text{ ; } \textbf{sync od}$$

Denotationally, the two components have the same set of streams, and $[\![C_1 \cap C_2]\!] = [\![C_1]\!] \cap [\![C_2]\!] = [\![C_1]\!] = [\![C_2]\!]$. Operationally, in each iteration, C_1 commits to exchange the value that it consumes from p through q, whereas in each of its iterations, C_2 commits to exchange the value that it consumes from q through p. The *cause* of the fulfillment of the commitment by C_1 in each round, thus, must be the availability of a data item on p, whereas the *cause* of the fulfillment of the commitment by C_2 in each round must be the availability of a data item on q. In spite of the fact that $[\![C_1 \cap C_2]\!] = [\![C_1]\!] = [\![C_2]\!]$, at run-time the concurrent execution of C_1 and C_2 deadlocks in a *causality loop*, because the success of the commitment by each component depends on the success of the commitment by the other. For $i \in \{1, 2\}$, the cause of the success of each exchange by C_i depends on the success of an exchange by C_{3-i}. However, neither C_i actually produces any data for any exchange to succeed. We refer to such cyclic dependencies as violation of *causality*.

Look-ahead. The ideal semantics in Section 1 defines the behavior of a component as a set of oracles. We call a (finite) *run* a (finite) sequence of port assignments, and call a *step* one element of a run. Intuitively, a one step look-ahead is an extension of the ideal semantics to accept finite runs that can be constructed up to reaching the next **sync** statement.

Consider the following two component descriptions:

$$C_1 = 0 \leftarrow p \text{ ; } \textbf{sync} \text{ ; } 1 \leftarrow p \text{ ; } \textbf{sync} \text{ ; } \omega$$
$$C_2 = 0 \rightarrow p \text{ ; } \textbf{sync} \text{ ; } 2 \rightarrow p \text{ ; } \textbf{sync} \text{ ; } \omega$$

where we use $n \leftarrow p$ as shorthand notation for $x \leftarrow p$; **assert** $x = n$ with $n \in \{0, 1\}$.

The component $[\![C_1 \cap C_2]\!]$ has empty behavior since the applications will diverge in the second round. Operationally, however, there is a run of size one in which p has value 0 in the first step. The run cannot be extended further, since p cannot be assigned any value in the next round. Therefore, the finite run $\langle\{p \mapsto 0\}\rangle$ consisting of only the single step $\{p \mapsto 0\}$ is a valid behavior only in what we refer to as a *one step look-ahead* semantics.

Observe that the ideal semantics has an infinite look-ahead, and can detect any further inconsistencies. Generally, however, no finite sequence of operations can implement infinite look-ahead. A finite k-step look-ahead semantics is a superset of the ideal semantics that also contains the k-length prefix of each of its runs. In principle, every finite k-step look-ahead semantics is implementable. Clearly, the ideal semantics rejects every k-length prefix that it does not contain. A k-step look-ahead semantics *over-approximates* the ideal semantics by admitting such *junk* runs. Observe that the smaller the k value, the more junk runs that a k-step look-ahead semantics contains. On the other hand, larger k values necessarily require more look-ahead to ascertain the validity of a run, which lead to less efficient implementations. Below, we introduce an operational semantics that avoids causality loops and abides by the one-step look-ahead constraint explained above.

Operational semantics. We consider a component C as the product of n PDL components, i.e., an expression of form:

$$C = S_1 \cap \ldots \cap S_n$$

where $c \in C$ denotes a primitive component S_i for some $1 \leq i \leq n$.

We take inspiration from [15, 17, 33, 37, 38], and more generally the literature on solving Constraint Satisfaction Problems (CSP). We distinguish the satisfaction problems of (1) finding which port fires at which round, and (2) finding which value to assign to each firing port.

We introduce some notation. A *firing map* is a partial function from ports to Boolean and we use $\rho : P \rightharpoonup \{\top, \bot\}$ as a typical element and $\emptyset$ as the firing map with empty domain. We use $\mathrm{dom}(\rho)$ to denote the set of ports $p \in P$ on which ρ is defined. We say that p fires in ρ if $\rho(p) = \top$, and does not fire otherwise. An *assignment* is a partial function from ports to values and we use $\mu : P \rightharpoonup V$ as a typical assignment. Observe that a step is a total assignment. Similarly to Section 1, we use s to range over component memory store, and S to range over the syntactic category of PDL statements.

A component state is a quadruple (ρ, μ, s, S) of a firing map ρ, an assignment μ, a map from state variables to values s, and a program statement S. We use $p!d$ and $p?d$ to denote the act of putting and the getting of value $d \in \mathcal{D}$ at port p, respectively. The small step operational semantics consists of the closure of the rules defined in Figure 4, under the following rule:

$$\frac{(\rho, \mu, s, S_1) \xrightarrow{X} (\rho', \mu, s', S_1')}{(\rho, s, S_1 \ ; \ S_2) \xrightarrow{X} (\rho', \mu, s', S_1' \ ; \ S_2)}$$

where we identify $\checkmark \ ; S_2$ with S_2.

Observe that the small step operational semantics is non-deterministic: in a state (ρ, μ, s, S), (1) multiple extensions of ρ' may exist, and/or (2) multiple values in the value domain of a port may satisfy the port-value constraints that determine a state.

Below, we introduce a deterministic relation $\twoheadrightarrow$ on the set of states of a component that models the concurrent execution of the component. In our context, the relation $\twoheadrightarrow$ is deterministic because it is functional on a set of states and a label. For a primitive component $c \in C$, a set of states Σ of c, and a state $(\rho, \mu, s, S) \in \Sigma$, we define the relation:

$$\frac{(\rho, \mu, s, S) \xrightarrow{X} (\rho', \mu', s', S'),\ X \neq \checkmark}{\Sigma \overset{X}{\twoheadrightarrow} \Sigma \cup \{(\rho', \mu', s', S')\}}$$

$$(\rho, \mu, s, \mathbf{skip}) \longrightarrow (\rho, \mu, s, \checkmark)$$
$$(\rho, \mu, s, x := e) \longrightarrow (\rho, \mu, s[x := [\![e]\!](s)], \checkmark)$$
$$(\rho, \mu, s, \mathbf{sync}) \xrightarrow{\checkmark} (\emptyset, \emptyset, s, \checkmark)$$
If $\rho(p)$ is true, $d \in \mathcal{D}$ and $p \notin \mathrm{dom}(\mu)$ or $\mu(p) = d$:
$$(\rho, \mu, s, x \leftarrow p) \xrightarrow{(\rho, p?d)} (\rho, \mu[p := d], s[x := d], \checkmark)$$
If $\rho(p)$ is true, $p \notin \mathrm{dom}(\mu)$ or $\mu(p) = s(x)$:
$$(\rho, \mu, s, x \rightarrow p) \xrightarrow{(\rho, p!s(x))} (\rho, \mu[p := s(x)], s, \checkmark)$$
If $[\![b]\!](\rho, s) = \mathbf{true}$:
$$(\rho, \mu, s, \mathbf{assert}\ b) \longrightarrow (\rho, \mu, s, \checkmark)$$
$$(\rho, \mu, s, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}) \longrightarrow (\rho, \mu, s, S_1)$$
$$(\rho, \mu, s, \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}) \longrightarrow (\rho, \mu, s, S \ ; \ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od})$$
If $[\![b]\!](\rho, s) = \mathbf{false}$:
$$(\rho, \mu, s, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}) \longrightarrow (\rho, \mu, s, S_2)$$
$$(\rho, \mu, s, \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}) \longrightarrow (\rho, \mu, s, \checkmark)$$
If $\rho \subseteq \rho'$:
$$(\rho, \mu, s, S) \longrightarrow (\rho', \mu, s, S)$$

Figure 4. Small-step real operational semantics for PDL.

$$\frac{(\rho, \mu, s, S) \xrightarrow{\checkmark} (\emptyset, \emptyset, s', S'),\quad \mathcal{I}(c) \subseteq \mathrm{dom}(\rho), \forall p \in \mathcal{I}(c).\ \rho(p) \implies p \in \mathrm{dom}(\mu)}{\Sigma \overset{(\rho, \checkmark)}{\twoheadrightarrow} \{(\emptyset, \emptyset, s', S')\}}$$

where $\mathcal{I}(c) \subseteq P$ is the interface of component c. Note that the domain of the firing map ρ labeling a transition for a **sync** statement covers all ports in $\mathcal{I}(c)$. Observe, as well, that the firing map in the state of a component may refer to the firing of some ports outside of its interface.

Implicitly, the rule for the **sync** statement in the definition of $\twoheadrightarrow$ imposes the following characteristics on components sharing ports. First, if a component puts or gets from a port shared by other components, then other components must also perform compatible put or get operations. The behavior models a *all-or-nothing* transaction mode. Second, all puts and all gets on the same port have the same value. We leave as future work the change of granularity in the semantics to define alternative behavior in sharing of port variables. [1]

Observe that $\twoheadrightarrow$ models the parallel progression of a component, where each state in Σ corresponds to a *speculative*

[1]*Nodes* in Reo closely resemble shared ports in PDL. A node is an n to m relation that acts as a merger on its n putters, and as a replicator on its m getters. A PDL port acts as a consensus on its putters, and as a replicator on its getters. As with the constraint automata semantics of Reo [5], we can model a Reo node as a composition of an explicit n-input merger component whose only output port is in a 1-to-1 relation with a 1-to-m PDL port.

branch of the component. Note, however, that we still allow components to progress and receive arbitrary values at ports.

We define, in addition, a labeled transition relation $\longrightarrow$ on a set Λ of pairs of a firing map and a data value, such that:

- $\Lambda \xrightarrow{(\rho,d)} \Lambda \cup \{(\rho, d)\}$ if, for all $(\rho', d') \in \Lambda$, there exists $p \in \mathrm{dom}(\rho) \cap \mathrm{dom}(\rho')$ such that $\rho(p) \neq \rho'(p)$; and
- $\{(\rho, d)\} \uplus \Lambda \xrightarrow{(\rho',d)} \{(\rho', d)\} \uplus \Lambda$ if $\rho \subseteq \rho'$.

We say that Λ is *consistent* if, given a total firing map ρ, there exists at most one pair $(\rho', d) \in \Lambda$ such that $\rho' \subseteq \rho$. Then, if Λ is consistent, we write $\Lambda(\rho) \in \mathcal{O}(\mathcal{D})$ to denote the value d if there exists $(\rho', d) \in \Lambda$ with $\rho' \subseteq \rho$, and to denote the value $\star$ otherwise.

Lemma 3.1. *For all (ρ, d), if $\Lambda \xrightarrow{(\rho,d)} \Lambda'$ and Λ is consistent, then Λ' is consistent.*

We now define, on a list of components, the constraints of valid causality and one step look ahead. We use $\mathcal{M}$ to range over *configuration* where $\mathcal{M}(c) \subseteq \Sigma$ returns a set of states for a primitive component c, and $\mathcal{M}(p)$ returns a set of pairs of a firing map ρ and a port value $d \in \mathcal{D}$ for a port $p \in P$. Let $c \in C$ and assume that unless stated otherwise, $\mathcal{M}'(x) = \mathcal{M}(x)$ for all x. We define a configuration relation satisfying the following four rules.

A component may freely do an internal transition (rule 1):

$$\frac{\mathcal{M}(c) \twoheadrightarrow \mathcal{M}'(c)}{\mathcal{M} \Rightarrow \mathcal{M}'} \tag{1}$$

A component may put a value on a port (rule 2) if its firing map updates the current port configuration:

$$\frac{\mathcal{M}(c) \overset{(\rho,p!d)}{\twoheadrightarrow} \mathcal{M}'(c),\ \mathcal{M}(p) \xrightarrow{(\rho,d)} \mathcal{M}'(p)}{\mathcal{M} \Rightarrow \mathcal{M}'} \tag{2}$$

Rule 2 allows a component to put on a port if and only if the port has a corresponding valid transition. Note that if there is no valid transition for the port, then the component cannot put its value and blocks. The last operational rule in Figure 4 enables speculation on an arbitrary port for a component. It is, therefore, entirely possible for a component to keep speculating (adding firing information in its firing map) until a put operation succeeds.

A component may get a value (rule 3) that is currently stored in the port configuration only if its firing map occurs in the configuration of the port:

$$\frac{\mathcal{M}(c) \overset{(\rho,p?d)}{\twoheadrightarrow} \mathcal{M}'(c),\ (\rho, d) \in \mathcal{M}(p)}{\mathcal{M} \Rightarrow \mathcal{M}'} \tag{3}$$

Observe that rule 3 equates the firing map on the transition of the component c with the firing map in the store of port p. Similarly as for rule 2, a component may speculate on the firing of a port that is not in its interface with the last rule of Figure 4. Practically, as detailed in Section 4, an exchange of information occurs between the port and a component, to construct the smallest extension of a valid firing map.

Finally, a component may synchronize (rule 4) if and only if all other involved components synchronize with the same firing map. As a result, the port configuration is reset to the empty set and the global assignment is exposed as the label of the transition:

$$\frac{\exists \rho.\ \begin{array}{c} \forall c \in C. \mathcal{M}(c) \overset{(\rho,\checkmark)}{\twoheadrightarrow} \mathcal{M}'(c) \\ \forall p \in P. \mathcal{M}'(p) = \emptyset \wedge \neg\rho(p) \iff v(p) = \star \wedge \\ v(p) = \mathcal{M}(p)(\rho) \end{array}}{\mathcal{M} \overset{v}{\Rightarrow} \mathcal{M}'} \tag{4}$$

Observe that the assignment v is well defined since $\mathcal{M}(p)$ is consistent for every $p \in P$.

Lemma 3.2. *Let C be a composite component and $\mathcal{M}$ a configuration. Then, $\mathcal{M} \overset{v}{\Rightarrow} \mathcal{M}'$ with total firing map ρ as a witness of the synchronization if and only if for all $c \in C$ there exists $(\rho, \mu_c, s_c, S_c) \in \mathcal{M}(c)$ such that $(\rho, \mu_c, s_c, S_c) \xrightarrow{(\rho,\checkmark)} (\emptyset, \emptyset, s_c', S_c')$ and for all $p \in \mathcal{I}(c)$, $\mu_c(p) = \mathcal{M}(p)(\rho)$.*

Proof. See appendix. □

Rule 4 together with the small-step operational semantics of Figure 4 entails the property stated in Lemma 3.2, that a total firing map is sufficient for each primitive component to enter in communication and exchange valid messages on shared ports.

We use $\mathcal{M} \overset{r}{\Rightarrow}_* \mathcal{M}'$ to denote the n successive applications of $\Rightarrow$ whose sequence of labels is the sequence r. We write $\mathcal{M} \overset{r}{\Rightarrow}_*$ if there exists $\mathcal{M}'$ such that $\mathcal{M} \overset{r}{\Rightarrow}_* \mathcal{M}'$, and for all sequences of assignments r' there does not exist $\mathcal{M}''$ such that $\mathcal{M}' \overset{r'}{\Rightarrow}_* \mathcal{M}''$. We use $\mathcal{M} \overset{\sigma}{\Rightarrow}_\omega$ if, for any $n \in \mathbb{N}$, there exists $\mathcal{M}'$ such that $\mathcal{M} \overset{s}{\Rightarrow}_* \mathcal{M}'$ with $s = \langle \sigma(0), ..., \sigma(n-1) \rangle$. Given $C = S_1 \cap ... \cap S_n$, we use $[\![\![C]\!]\!]$ to denote the set:

$$[\![\![C]\!]\!] = \{s \mid \mathcal{M} \overset{s}{\Rightarrow}_*\} \cup \{\sigma \mid \mathcal{M} \overset{\sigma}{\Rightarrow}_\omega\}$$

where $\mathcal{M}$ quantifies over all initial configurations of the form $\mathcal{M}(c) = \{(\emptyset, \emptyset, \iota_c, S_c)\}$ and $\mathcal{M}(p) = \emptyset$ for all primitive component $c \in C$ and $p \in P$.

Theorem 3.3. *For any component C, $[\![\![C]\!]\!] \cap [\![\omega]\!] \subseteq [\![C]\!]$.*

As previously stated, the idealistic and realistic semantics differ in the class of behavior that each captures. Theorem 3.3 states that every *infinite* run operationally constructed with the realistic semantics is also an element of the idealistic semantics. There is a class of components C such that the converse holds, and $[\![\![C]\!]\!] \cap [\![\omega]\!] = [\![C]\!]$. Those components are such that every run satisfies the causality imposed by the realistic semantics, and all the put and get operations are paired with a corresponding receiver and sender. We call such component *causal* and *closed.* Section 4 proposes a runtime to distribute and execute such causal and closed composite component.

4 Distributed Runtime

In Section 3 we defined the realistic semantics of PDL, which generates runs given a protocol. It relies on trivial access to the configuration, making it suitable for an implementation in shared memory. In this section, we adapt the realistic semantics to a context in which the configuration is distributed over a physical network. This forms the basis of a distributed runtime which serves as the platform that drives the communications between a set of distributed applications given their distributed, shared protocol, specified just in time.

Distributed primitives. In this setting, each primitive in a protocol works autonomously within its own memory space, unable to directly act on the contents of its peers' memory. Primitives can work together indirectly by sending and receiving *control messages*; note that these are distinct from the messages that components put and get at ports. A system of distributed primitives consists of nodes in a *transport graph* whose edges characterize *neighboring* primitive pairs, sufficiently aware of each other to exchange control messages. We assume that every primitive can send itself control messages, i.e., each primitive is its own neighbor.

The transport of control messages is assumed to be *reliable*, i.e., all messages sent are eventually received after some finite time, but not necessarily in the same order as they were sent. Neighbors A and B can cooperate to realize the reliable transmission of control message m from A to B as follows. A repeats $msg(m)$ at regular intervals until it receives $ack(m)$. B sends $ack(m)$ whenever it receives $msg(m)$. Later in this section, we require that control messages sent during a round must be received within that round only, i.e., control messages from previous rounds are ignored. This can be achieved by numbering each control message with the round number in which it was sent. Recipients discard incoming messages with an old round number.

For convenience, we introduce *replication* as an abstraction over control message exchange. Neighbors A and B maintain an eventually-consistent replica of a set E as follows if (1) both replicas are initially consistent, and (2) elements are never removed from a replica. For each element e added by A to its replica, A sends a control message to B instructing B to likewise add e to its replica. The eventual consistency of E's replicas follows from reliability. To replicate a (partial) function, it suffices to replicate its set of input-output pairs.

Decision tree. As a given protocol may denote several acceptable runs, generating the next step requires a *decision*, selecting one in particular. In the case of shared memory, it suffices for the system as a whole to decide arbitrarily. However, distributed primitives arriving at the same decision presents a consensus problem. We opt to centralize the decision at a fixed *leader* primitive, whose arbitrary decisions

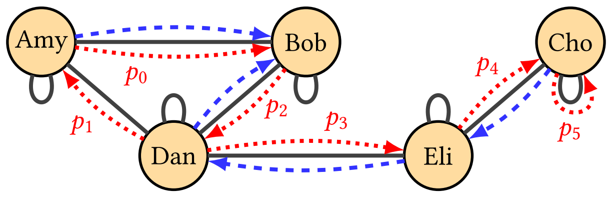

Figure 5. Example of a transport graph (black, solid edges) overlaid by a decision tree (blue, dashed edges), and linkage (red, dotted edges and port labels p_{0-5}).

are adopted by all other primitives. This can be understood as indirectly ordering steps by directly ordering primitives.[2]

We say a graph is *overlaid* atop another if both have identical nodes, and each edge in the former corresponds to an edge in the latter. We designate the root of a fixed *decision tree* G_D overlaid atop a transport graph as the leader of the latter graph. Note that a decision tree is necessarily *contiguous*[3], i.e., there exists a path in the tree between each pair of primitives. A decision tree orders the set of primitives by breaking the symmetry between parents and children, and defining a path for each primitive to and from the leader. Later, we take for granted that all primitives are able to come to consensus on a value following the leader's decision, propagated using the wave algorithm, centralized at the leader. [22] Figure 5 gives an example of an overlaid decision tree.

Linkage. Thus far, we imposed no restriction on primitives' access to ports. However, there is value in prescribing a unique *putter* and *getter* per port p. Concretely, a given *linkage* $L : P \to \mathbb{B} \to C$ *allows* a component c if and only if $\forall p \in \mathcal{I}(c), c' \in prim(c) : (p \in \mathcal{I}_p(c') \to L(p)(\bot) = c') \wedge (p \in \mathcal{I}_g(c') \to L(p)(\top) = c')$, where $prim(c)$ returns the primitives of component c. We say port p *links* its putter c_p to its getter c_g when $L(p)(\bot) = c_p \wedge L(p)(\top) = c_g$. A linkage can inform the distribution of port information over primitives. Figure 5 gives an example of a linkage depicted as a graph overlaid atop a transport graph, with each port p depicted as a p-labeled edge, directed from putter to getter.

Note that there may not exist a linkage that allows a given component c, specifically, if two of c's primitives both put or both get at some port. However, given a component c, we can derive a component c' and a linkage L, such that $\mathcal{I}(c') \subseteq \mathcal{I}(c)$, and c' has the same behavior after hiding all of its ports not in $\mathcal{I}(c)$. Intuitively, the procedure works by mapping a port with multiple putters and/or multiple getters into several ports, whose values are then kept equivalent by newly-added primitives. This is the same scheme used in the definition of the coloring semantics of Reo [16].

[2] Consensus follows if one directly orders steps. For example, each primitive selects the maximum. However, this requires that all steps be known, which is far less practical than requiring that all primitives be known.

[3] We use 'contiguous' for what is often called 'connected component' in the literature to avoid confusion with out notion of 'component'.

Given the protocol definitions of *Same* and *Rep* of Figure 1, let c' be initialized to c, and then modified as follows:

- While primitives c_1 and c_2 of c' **put** at port p:
  Take fresh ports $\{p_1, p'_1, p_2, p'_2\}$. Replace occurrences of p within puts in c_1 and in c_2 to p_1 and p_2 respectively. Finally, replace c' with $c' \cap Same(p'_1, p'_2, p)$.
- While primitives c_1 and c_2 of c' **get** at port p:
  Take fresh ports $\{p_1, p'_1, p_2, p'_2\}$. Replace occurrences of p within get in c_1 and in c_2 to p_1 and p_2 respectively. Finally, replace c' with $c' \cap Rep(p, p'_1, p'_2)$.

After each step, p has one fewer putter or getter in each case, respectively. Each fresh port always has one of each. Ultimately, each port in $\mathcal{I}(c')$ has at most one putter and one getter. As such, a linkage allowing c' necessarily exists.

As expressed by Lemma 4.1, the behaviors of c and c' are equivalent once the added fresh ports are hidden in c'. This is because the added primitives preserve the equality of values at ports that were previously not distinguished, and introduce causal dependencies only for gets on their respective puts, as is also the case in c.

For simplicity henceforth, we assume that primitive c puts at a port p if and only if c is p's putter, and likewise for get.

Lemma 4.1. *For all* $c \in C$, $[\![c]\!] = [\![\exists p_1. \dots \exists p_n.\ c']\!]$ *where* $\mathcal{I}(c') \setminus \mathcal{I}(c) = \{p_1, \dots, p_n\}$.

Step generation. A *session* $\mathcal{S} = (c, G_T, G_D, L)$ consists of a protocol c decomposed into primitives which are the nodes of a transport graph G_D, overlaid by a decision tree G_T and a linkage L, where L allows c, and all links are neighbors.

We adapt the realistic semantics of PDL to generate a run from the session's protocol. In this context, the execution at large emerges from the actions of its constituent primitives. A run is computed incrementally, through each primitive's participation in two concurrent step procedures: *distributed* and *centralized*. These procedures partition the task of applying rules 1–4 defined in Section 3.

Step generation: decentralized. The decentralized procedure applies rules 1–3, each of which requires access to only one primitive's state. As such, each primitive c applies only rules matching $\mathcal{M}(c)$ to explore only its own state space. In this procedure, each primitive interacts with its peers only via $\mathcal{M}(p)$, which may be replicated by a neighbor. Concretely, for each port p, $\mathcal{M}(p)$ is replicated by primitives $L(p)(0)$ and $L(c)(1)$. In this manner, neighboring primitives cooperate in the exploration of their respective state spaces; puts at p write elements to $\mathcal{M}(p)$ for p's getter to read.

Step generation: centralized. The centralized procedure aggregates information at the leader until it is sufficiently informed to apply rule 4. This occurs once per completed round, and results in all primitives updating their own states to reflect the newly-identified step in the run.

We say a firing map ρ *satisfies* a primitive c if and only if c has explored a state matching $(\rho', \mu, s, \textbf{sync}\ ;\ S)$ where $\rho' \subseteq \rho$. We say a firing map ρ *covers* a primitive c if and only if $\mathcal{I}(c) \subseteq dom(\rho)$. By a *solution* we refer to a firing map that satisfies and covers all primitives. By Lemma 3.2, each solution in a round corresponds to a particular step. Thus, it suffices for primitives to reach consensus on a solution; recall that this follows from the leader identifying and deciding on a solution. A firing map ρ is a *candidate* of a primitive c if ρ satisfies and covers each primitive in the decision sub-tree rooted at c. In the following, we give an algorithm such that the leader can discover its candidates. This suffices, as we show that the leader's candidates coincide with solutions.

A candidate is defined in terms of a global view on the decision tree, which is useful for characterizing solutions. However, c's candidates are defined in terms of information not always local to c. To proceed, we introduce an invariant, per primitive c, whose preservation requires only c-local information. Next, we extend the configuration to include $\mathcal{N}$, such that $\mathcal{N}(c)$ returns the set of c's candidates, for each primitive c. $\mathcal{N}$ is distributed such that $\mathcal{N}(c)$ is replicated at c and c's parent (if it exists). Observe that this lets primitives read the candidate sets of their children. For brevity, let $F(c)$ return the firing maps that cover and satisfy primitive c; this information is unfolded by the distributed procedure. Furthermore, let $Q(c)$ return a list of firing map sets, including $F(c)$, and $\mathcal{N}(c')$ for each c' child of c. $Q(c)$ can be understood as containing all the information from which $\mathcal{N}(c)$ can be derived. Let each primitive c continuously update c to preserve the following invariant equality:

$$\mathcal{N}(c) = \{\ \forall q \in Q, \exists \rho' \in q : \rho' \subseteq \rho\ \}$$

Lemma 4.2. *For all component* c, $\rho \in \mathcal{N}(c)$ *if and only if, for all* c' *in the decision sub-tree rooted at* c*, then* $\rho \in \mathcal{N}(c')$.

To see that Lemma 4.2 follows from the invariant for any primitive, it suffices to rewrite occurrences of $\mathcal{N}(c)$ for all c from left to right, and to expand the quantification of q. It becomes clear that by this inductive definition, candidates cover and satisfy the expected set of primitives. This is apparent for a primitive c with no children, as $\mathcal{N}(c) = F(c)$.

Each primitive c preserves the equation by monitoring $Q(c)$ and adding candidates to $\mathcal{N}(c)$ as follows. Initially, no primitive has a candidate, as no states have been explored; thus the lemma trivially holds at the start of the round. As the set of explored states only grows, each primitive's candidates only grows. It suffices for primitive c to monitor the sets in $Q(c)$ for new additions, and react by accordingly adding elements to $\mathcal{N}(c)$. Observe that is it never necessary to add ρ to $\mathcal{N}(c)$ if ρ' is already present, and $\rho \subseteq \rho'$. This is because no primitive can be satisfied and covered by the former and not the latter. This observation allows for a substantial reduction in the number of candidates of non-leader primitives, without affecting the leader's decision.

Application primitives. We extend the execution of a protocol in a session to include *application primitives*, each representing a user application as a participant in the session. Such a primitive is characterized by its protocol c_a being unspecified ahead of time. Rather, c_a unfolds up to the end of a round just as the round begins. This unfolding is facilitated by *synchronization*, a procedure exposed by the connector API, whose input specifies c_a up to its next **sync** statement. For example, below, defines $c_a^0 = c_a$ as round 0 begins, c_a^1 as round 1 begins, and so on:

$$c_a^0 = m_0 \longrightarrow p_0 \ ; m_1 \longrightarrow p_1 \ ; \ \textbf{assert} \ ¿p_2 \ ; \ \textbf{sync} \ ; \ c_a^1$$

As desired, the connector API can be simplified at the cost of application expressiveness. For example, the connector computes the definition of c_a^0 above, given only the subset of firing ports in $\mathcal{I}(c_a)$, and prescribing each message produced: $(\{p_0 \mapsto m_0, p_1 \mapsto m_1\}, \{p_2\})$.

Once the round is completed, synchronization returns the decided step projected onto $\mathcal{I}(c_a)$, such that the application can reflect on the outcome, e.g., by reading a port's message. This approach results in *cooperative scheduling* between an application and its connector: synchronization passes control flow back and forth. Primitives speculate 'during' a round, and applications reflect on previous rounds and prepares for the next round 'between' rounds.

Session setup. Initially, a session consists only of application primitives, isolated in the transport graph, and with no ports in the linkage. In the initial *setup* phase, applications can cooperate to add a fresh port, its link, and the underlying transport edge (if it does not already exist) all together. Concretely, each connector is given as input: (1) the identity of the other primitive, and (2) the direction of the link. An implementation may identify primitives using IPv4 or similar addresses; identifiers of some sort are necessary to facilitate consensus, as the literature shows that consensus in arbitrary contiguous networks is impossible otherwise. [22] To ensure that the applications have a consistent view on the link direction, each of their primitives informs the other of the expected direction in a control message; the procedure fails if the primitives learn that their expectations differ.

If the transport graph is contiguous, its primitives can complete the session setup together through the decentralized construction of the decision tree. First, a leader is elected using Chang's *echo algorithm with extinction* [14]. This requires that primitives' identifiers are ordered. Second, the *echo algorithm* [22] is initiated by the leader, identifying the parents and children of each primitive amongst its neighbors.

Session transformation. An application primitive c_a can introduce new ports and primitives without disturbing its neighbors. It can do so before or after the session setup. In the first case, it adds a fresh port p to $\mathcal{I}(c_a)$, and updates the linkage such that $L(p)(\bot) = L(p)(\top) = c_a$. In the second case, it adds a new component c_b to the session's protocol c, updating c to $c \cap c_b$. The c_b becomes c_a's (1) neighbor in the transport graph, and (2) child in the decision tree. In the process, c_a may choose to replace any subset of occurrences of c_a with c_b in its links, transferring access to a subset of c_a's ports to c_b. We assume the affected links still correspond to edges in the transport graph.[4] We extend this functionality such that applications can also add composite protocols by decomposing each into a set of primitives. The connector must take extra care to preserve linkage, e.g., by pre-processing the protocol as previously described.

Arbitrary transformations of the session are significantly more invasive and complex, necessitating delicate distributed procedures. Earlier work on dynamic reconfiguration of Reo circuits [11, 30–32] shows that such transformations are possible, laying the groundwork for the same in Reowolf. In future, we will investigate the application of graph rewriting techniques in general [20] such as PBPO$^+$ in particular [36] to manipulate regions of the transport graph. The power to alter the session dynamically adds a great deal of flexibility. As in [35], we are particularly interested in session transformations that have no effect on behavior observable to applications, but are otherwise more desirable. As a simple example, one edge in the decision tree is inverted, reducing the lengths of paths to the leaves, resulting in rounds completing more quickly. For a more realistic example, consider a session transformation that moves a filter component physically closer to the source of its incoming messages.

Distributed timeout. In general, a round may continue for an arbitrary duration without the leader making a decision. Whether or not a solution exists to be found, an application may wish to trigger a *distributed timeout* in order to restart the round, potentially providing their primitive with a different protocol specification.

During a round, a primitive can send a timeout request control message through the decision tree to the leader. Upon receipt, if no decision has yet been made, the leader decides on a timeout, which results in consensus as usual. The resulting distributed timeout restores the configuration to that of the start of the round. Although this distributed procedure may take an arbitrary duration, it is short in practice, as its involves very little work per primitive.

5 Evaluation & Future Work

Sections 1–4 define PDL and explain its usage for driving communications between networked applications. In this section, we evaluate the strengths and weaknesses of this contribution, and outline promising future developments.

Strengths. Connectors are sufficiently practical to afford a systems-level, distributed implementation. This is evidenced by the completion of a prototype implementation, along with

[4]To relax this assumption, replication must be extended to any pair of primitives. This can be achieved via the transitive closure of replication, i.e., a pair can replicate E if all primitives in a path between them replicate E.

a technical report which includes the results of experimental testing. These work products are publicly available in a persistent Zenodo repository [1].

By orienting their API around protocols, connectors narrow the gap between an application's implementation, and the specification of its high-level properties. This makes applications more high-level, thus, more maintainable and re-usable. Furthermore, one can reason about the high-level properties of sessions via their specifications. In future, we want middleware to automatically leverage the given protocols to apply optimizations at run-time. We are particularly interested in optimizations arising from the composition of multiple applications' protocols.

The connector API affords applications great flexibility, letting them interleave their communications with the addition of protocols to be preserved. Later, we want to increase this flexibility to enable transformations of an ongoing session's protocol. Applications are also free to form sessions by identifying only their neighbors.

The distributed procedures driving the runtime are largely decentralized, with primitives exchanging control messages with their neighbors concurrently. Furthermore, each primitive explores paths through its state space concurrently. As a result, rounds can progress quickly by the leader deciding on solutions found quickly; the existence of complex solutions does not impede progress of simpler ones.

Weaknesses. A small but crucial part of the distributed runtime involves a centralized decision event. Thus, the decision tree is a single point of failure. With some adjustment, the runtime can re-create the decision tree on demand to bypass any failed nodes and edges, using any of several distributed algorithms [8, 9, 23]. However, we expect that the decision tree cannot be dynamic without incurring significant overhead. In future work we will explore empowering applications to strike the balance themselves.

Currently, causal consistency is not preserved by protocol composition. As a result, not all desirable properties are characterized by a protocol without context. In future work we will further develop PDL, exploring changes that either make causality more explicitly expressed, or relax the need for runs to be causally consistent. In investigating the latter, we can continue to draw from work on constraint solving. [33]

Currently, connectors provide strong consistency guarantees, but use only one round look-ahead into protocols, and all primitives must participate to complete the round. In future, we want to generalize look-ahead, and let primitives be replicated over physical nodes, such that progress is robust to the failure of physical nodes and channels. Furthermore, we want to investigate relaxations of PDL that let some primitives progress, while leaving others behind, such that overall progress is not inhibited by slow primitives.

As protocols cannot be simultaneously mutable and immutable, session re-configuration and optimizations that leverage the preservation of protocols are mutually-exclusive features. In future work we will explore letting applications make this trade-off per protocol, as best suits their needs.

6 Related Work

This section compares the approach of Reowolf to that of several works with comparable problems or solutions.

Multi-party session types. Session types apply established type-checking disciplines to message-passing between networked processes. [18] The behavior of a process or channel endpoint is specified by a (local) session type used to check the correctness of the process's implementation. The trick is to assign types such that correctness of a session's behavior follows from that of its processes. Later work [26] introduced global session types ('GST') for characterizing communications between any number of peers. Projection of a GST onto each of a session's processes assigns it a local session type used to check local correctness as before.

GSTs and PDL have in common that they formalize the behavior of multi-party sessions, and are ultimately used to ensure that programs behave as specified. However, they differ in specificity, and in which context they are used. Both GSTs and PDL protocols can express choice by defining their behavior as a function of values chosen at runtime. PDL protocols express choice by reflecting on the messages they observe at their ports; they are able to constrain the choice made through assertions, but there is no specification of how the choice is made. In contrast, GSTs associate choices with message values originating at a specified sender, thereby fixing the sender as being solely responsible for making the choice. This demonstrates how PDL relies more extensively on its runtime system for its execution.

Ongoing work in session types muddies the aforementioned distinctions between PDL and GSTs, introducing GST variations that don't prescribe which process makes a choice.

Software Defined Networks (SDN). Software Defined Networks [27] distinguish between the control and data planes. On the control plane, messages are exchanged to update the configuration of some devices on the network; while the data plane deals with communication protocols. As an example, OpenFLow is a control protocol used for remote administration of switch's packet forwarding tables. Rules can be made separately on a controller, and dynamically pushed to the switches on a network, changing therefore the routing algorithm [34]. SDN comes also together with the virtualisation of network functions (also abbreviated NFV). The OpenStack, mainly maintained by Cisco, is a set of virtualized network services that can be deployed and configured remotely.

SDN and Reowolf mainly differ in their purpose. SDN eases the administration of networks, while Reowolf enables multi-party synchronous communications. They both, however, aim at taking networks and protocols as a first class concepts in their solutions (applications barely talk about sockets in SDN, but more about quality requirements).

Synchronous languages. Besides Reo [2–4], other works have been done on the design of synchronous languages. For instance, the imperative language Esterel [7], and the declarative language LUSTRE [12], are languages whose semantic models are similar to ours, in that they consider *histories* as infinite sequences of port assignments [6]. The difference is mainly in how each model generates such histories. Esterel and LLUSTRE use a clock synchronization mechanism. Our work differs in that in our model time is not explicit, but implicitly progresses via **sync** statements: only by performing **sync** all components synchronize.

Linda. Linda is a coordination language [10] whose primitives communicate asynchronously through a shared data space (called tuple space). Processes generate messages in the tuple space, which are eventually withdrawn by other processes. The operation of sending a message to the tuple space is non-blocking, while reading and removing messages may block. Synchrony in Linda is thus modeled as a sequence of send and receive operations between two processes.

Bulk Synchronous Parallel (BSP). BSP was an architecture suggested by Valiant in [39]. The idea was to build a conceptual bridge between software and hardware (analogous to Von Neumann architecture for sequential computation) for parallel computation. The architecture led to BSPlib, a library used for parallel computation [25], in which synchronization is separated from communication. A BSP computation consists of a sequence of parallel supersteps. A supersteps contains, in order, a phase of local computation at each process, a phase of communication between processes, and barrier synchronization among the processes. Reowolf has in common with BSP that multi-party synchronization is a feature of the language. However, BSP restricts to processes within the same machine and, to our knowledge, does not consider an implementation over an IP network. Moreover, BSP mainly does not consider data synchronization, while our runtime includes speculation and constraint solving.

MPI. Message passing interface (MPI) was developed incrementally throughout the 1990's. It is an interface for enabling a programming model for communicating synchronous [24], particularly popular in computational science. MPI and Reowolf's connectors have in common that they provide an abstraction over a multi-party session in which user applications exchange messages. MPI-2 lets processes dynamically instantiate other processes, much as Reowolf lets components instantiate other components. MPI offers variations of message-passing operations; in their applications, programmers effectively configure their usage of MPI's network abstraction to maximize runtime performance.

Reowolf differs from MPI in unifying the two features above into the activity of adding protocols to the session, (1) specifying behavior, and (2) delegating work to a new entity. These two activities coincide to enable reasoning about the latter in terms of reasoning about the former.

OpenMP. OpenMP is an API for introducing multi-threaded parallelism into sequential implementations with minimal impact on the source code. [13] For example, a C programmer annotates a for-block with the `parallel for` compiler directive, partitioning the work of the loop body over a set of worker threads. These directives accept keyword annotations on local variables, providing programmers control over how values are replicated and accessed by workers.

OpenMP and Reowolf have in common that they introduce a high-level language for coordinating concurrent processes, aiming to minimize the coupling between the computational task and inter-worker coordination. However, OpenMP differs from Reowolf in the task it aims to simplify. OpenMP eases static reasoning about a large code base. Reowolf eases reasoning about the behavior of modular components as part of a larger network context to be realized at runtime.

7 Conclusion

Connectors show promise as a multi-party session abstraction, interfacing the transport layer below with the application layer above. Like sockets, connectors facilitate message passing between their applications, distributed over physical networks such as the Internet. Unlike sockets, the connector API is oriented around applications dynamically adding PDL protocols to be preserved in the session. Two objects coincide in a protocol: (1) a specification of a session's properties, and (2) a distributed program a session can execute.

Via connectors, protocols become a powerful vehicle for capturing and communicating the application's requirements in the OS and further into the network. On the one hand, applications consequently have more flexible and abstract implementations, becoming easier to maintain and alter. On the other hand, the runtime gets insight into the applications' requirements. There is much future work to be done to continue to exploit this insight, for example, by transparently optimizing the efficiency of an ongoing session. This also includes developments to mitigate the current weaknesses. For example, the distributed runtime must be made more flexible to changes in the physical network. Further opportunities are expected to arise from the PDL, as it expands to capture new high-level protocol properties.

These contributions build on previous work to develop a paradigm in which network protocols are concrete artifacts. Our ambition is to cover as much of the OSI network stack as possible, such that communications over the Internet become more high-level, reliable, transparent, and efficient.

References

[1] 2020. Reowolf 1.0 deliverables Zenodo repository. (url and doi omitted to preserve anonymity).

[2] Farhad Arbab. 2004. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14, 3 (2004), 329–366. https://doi.org/10.1017/S0960129504004153

[3] Farhad Arbab. 2011. Puff, The Magic Protocol. In *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)*, Gul Agha, Olivier Danvy, and José Meseguer (Eds.). Springer, 169–206. https://doi.org/10.1007/978-3-642-24933-4_9

[4] Farhad Arbab and Jan J. M. M. Rutten. 2002. A Coinductive Calculus of Component Connectors. In *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 2755)*, Martin Wirsing, Dirk Pattinson, and Rolf Hennicker (Eds.). Springer, 34–55. https://doi.org/10.1007/978-3-540-40020-2_2

[5] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. 2006. Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.* 61, 2 (2006), 75–113. https://doi.org/10.1016/j.scico.2005.10.008

[6] Albert Benveniste, Paul Le Guernic, Yves Sorel, and Michel Sorine. 1992. A Denotational Theory of Synchronous Reactive Systems. *Inf. Comput.* 99, 2 (1992), 192–230. https://doi.org/10.1016/0890-5401(92)90030-J

[7] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

[8] Marc Bui, Franck Butelle, and Christian Lavault. 2013. A Distributed Algorithm for Constructing a Minimum Diameter Spanning Tree. *CoRR* abs/1312.1961 (2013). arXiv:1312.1961 http://arxiv.org/abs/1312.1961

[9] Marc Bui, Franck Butelle, and Christian Lavault. 2013. A Distributed Algorithm for Constructing a Minimum Diameter Spanning Tree. *CoRR* abs/1312.1961 (2013). arXiv:1312.1961 http://arxiv.org/abs/1312.1961

[10] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. 2000. On the Expressiveness of Linda Coordination Primitives. *Inf. Comput.* 156, 1-2 (2000), 90–121. https://doi.org/10.1006/inco.1999.2823

[11] C.) Krause Christian C. (born Köhler. 2011. *Reconfigurable component connectors*. Ph.D. Dissertation. https://doi.org/hdl:1887/17718

[12] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 178–188. https://doi.org/10.1145/41625.41641

[13] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

[14] Ernest J. H. Chang. 1982. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. Software Eng.* 8, 4 (1982), 391–401. https://doi.org/10.1109/TSE.1982.235573

[15] Behnaz Changizi, Natallia Kokash, and Farhad Arbab. 2012. A constraint-based method to compute semantics of channel-based coordination models. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA). IARIA.*

[16] Dave Clarke, David Costa, and Farhad Arbab. 2007. Connector colouring I: Synchronisation and context dependency. *Sci. Comput. Program.* 66, 3 (2007), 205–225. https://doi.org/10.1016/j.scico.2007.01.009

[17] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. 2011. Channel-based coordination via constraint satisfaction. *Sci. Comput. Program.* 76, 8 (2011), 681–710. https://doi.org/10.1016/j.scico.2010.05.004

[18] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. 2009. Sessions and Session Types: An Overview. In *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6194)*, Cosimo Laneve and Jianwen Su (Eds.). Springer, 1–28. https://doi.org/10.1007/978-3-642-14458-5_1

[19] Kasper Dokter and Farhad Arbab. 2018. Rule-Based Form for Stream Constraints. In *Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings (Lecture Notes in Computer Science, Vol. 10852)*, Giovanna Di Marzo Serugendo and Michele Loreti (Eds.). Springer, 142–161. https://doi.org/10.1007/978-3-319-92408-3_6

[20] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer. https://doi.org/10.1007/3-540-31188-2

[21] Christopher A. Esterhuyse and Hans-Dieter A. Hiep. 2019. Reowolf: Synchronous Multi-party Communication over the Internet. In *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)*, Farhad Arbab and Sung-Shik Jongmans (Eds.). Springer, 235–242. https://doi.org/10.1007/978-3-030-40914-2_12

[22] Wan Fokkink. 2018. *Distributed algorithms: an intuitive approach*. MIT Press.

[23] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. 1983. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.* 5, 1 (1983), 66–77. https://doi.org/10.1145/357195.357200

[24] William Gropp. 2011. MPI (Message Passing Interface). In *Encyclopedia of Parallel Computing*, David A. Padua (Ed.). Springer, 1184–1190. https://doi.org/10.1007/978-0-387-09766-4_222

[25] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin J. Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. 1998. BSPlib: The BSP programming library. *Parallel Comput.* 24, 14 (1998), 1947–1980. https://doi.org/10.1016/S0167-8191(98)00093-3

[26] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. https://doi.org/10.1145/1328438.1328472

[27] Fei Hu, Qi Hao, and Ke Bao. 2014. A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation. *IEEE Commun. Surv. Tutorials* 16, 4 (2014), 2181–2206. https://doi.org/10.1109/COMST.2014.2326417

[28] Sung-Shik Theodorus Quirinus Jongmans. 2016. *Automata-theoretic protocol programming*. Ph.D. Dissertation. Leiden University.

[29] Sung-Shik T. Q. Jongmans and Farhad Arbab. 2012. Overview of Thirty Semantic Formalisms for Reo. *Sci. Ann. Comput. Sci.* 22, 1 (2012), 201–251. https://doi.org/10.7561/SACS.2012.1.201

[30] Christian Krause, David Costa, José Proença, and Farhad Arbab. 2008. Reconfiguration of Reo Connectors Triggered by Dataflow. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 10 (2008).

[31] C. Krause, H. Giese, and E.P. Vink, de. 2013. Compositional and behavior-preserving reconfiguration of component connectors in Reo. *Journal of Visual Languages and Computing* 24, 3 (2013), 153–168. https://doi.org/10.1016/j.jvlc.2012.09.002

[32] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. 2011. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Sci. Comput. Program.* 76, 1 (2011), 23–36. https://doi.org/10.1016/j.scico.2009.10.006

[33] Vipin Kumar. 1992. Algorithms for constraint-satisfaction problems: A survey. *AI magazine* 13, 1 (1992), 32–32.

[34] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and

Jonathan S. Turner. 2008. OpenFlow: enabling innovation in campus networks. *Comput. Commun. Rev.* 38, 2 (2008), 69–74. https://doi.org/10.1145/1355734.1355746

[35] Nuno Oliveira and Luís S. Barbosa. 2013. On the reconfiguration of software connectors. In *SAC '13: Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 1885–1892. https://doi.org/10.1145/2480362.2480712

[36] Roy Overbeek, Jörg Endrullis, and Aloïs Rosset. 2021. Graph Rewriting and Relabeling with PBPO$^+$. In *Graph Transformation - 14th International Conference, ICGT 2021, Held as Part of STAF 2021, Virtual Event, June 24-25, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12741)*, Fabio Gadducci and Timo Kehrer (Eds.). Springer, 60–80. https://doi.org/10.1007/978-3-030-78946-6_4

[37] José Proença and Dave Clarke. 2013. Data abstraction in coordination constraints. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 159–173.

[38] José Proença and Dave Clarke. 2013. Interactive interaction constraints. In *International Conference on Coordination Languages and Models*. Springer, 211–225.

[39] Leslie G. Valiant. 2011. A bridging model for multi-core computing. *J. Comput. Syst. Sci.* 77, 1 (2011), 154–166. https://doi.org/10.1016/j.jcss.2010.06.012

Appendix

Proof sketch for Theorem 2.1. We proceed inductively on the structure of the statement, and show that for all $\sigma \in \Delta$, $\sigma \in [\![S]\!]$ if and only if there exist $s \in \Sigma$ and τ such that $((s,0),\sigma,\tau) \in [S]$. We give the proof for the branching, sequential, and loop constructs. We use s to denote an arbitrary state in Σ, and τ and λ for arbitrary streams of states in $\mathcal{O}(\Sigma)$.

Case **if** b **then** S_1 **else** S_2 **fi**. Suppose that $[\![S_1]\!] = \{\sigma \mid ((s,0),\sigma,\tau) \in [S_1]\}$ and $[\![S_2]\!] = \{\sigma \mid ((s,0),\sigma,\tau) \in [S_2]\}$. We fix $s \in \Sigma$, then

$$\begin{aligned}
&[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}]\!]\\
&\quad=\{\sigma \mid (\sigma, s, \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) \downarrow\}\\
&\quad=\{\sigma \mid (\sigma, s, S_1) \downarrow \text{ and } [\![b]\!](\sigma(0), s) = \textbf{true}\} \cup\\
&\qquad\{\sigma \mid (\sigma, s, S_2) \downarrow \text{ and } [\![b]\!](\sigma(0), s) = \textbf{false}\}\\
&\quad=\{\sigma \mid ((s,0),\sigma,\tau) \in [S_1] \text{ and } [\![b]\!](\sigma(0), s) = \textbf{true}\} \cup\\
&\qquad\{\sigma \mid ((s,0),\sigma,\tau) \in [S_2] \text{ and } [\![b]\!](\sigma(0), s) = \textbf{false}\}\\
&\quad=\{\sigma \mid ((s,0),\sigma,\tau) \in [\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}]\}
\end{aligned}$$

Case S_1 ; S_2. Suppose that $[\![S_1]\!] = \{\sigma \mid ((s,0),\sigma,\tau) \in [S_1]\}$ and $[\![S_2]\!] = \{\sigma \mid ((s,0),\sigma,\tau) \in [S_2]\}$. We fix $s \in \Sigma$, then $[\![S_1 \,;\, S_2]\!]$ is the set

$$\begin{aligned}
&\{\sigma \mid (\sigma, s, S_1 \,;\, S_2) \downarrow\}\\
&\quad= \{\sigma \mid (\sigma, s, S_1, \emptyset) \longrightarrow (\sigma_2, s', \checkmark, \emptyset) \text{ and} (\sigma_2, s', S_2) \downarrow\}\\
&\qquad \cup \{\sigma \mid (\sigma, s, S_1, \emptyset) = (\sigma_0, s_0, S_{1,0}, \emptyset) \text{ and}\\
&\qquad\qquad \forall n. \exists m. (\sigma_n, s_n, S_{1,n}, \emptyset) \longrightarrow_m (\sigma_{n+1}, s_{n+1}, S_{1,n+1}, \checkmark)\}
\end{aligned}$$

We first observe that if $(\sigma, s, S_1, \emptyset) \longrightarrow (\sigma_2, s', \checkmark, \emptyset)$ then σ_2 is a postfix of σ, and there exists a $j \in \mathbb{N}$ such that the j-th derivation of σ is σ_2, i.e., $\sigma^{(j)} = \sigma_2$. Therefore, $[S_1] \circ [S_2]$ is the set

$$\begin{aligned}
&\{(\tau, \sigma, \lambda) \mid (\tau, \sigma, (s', j)) \in [S_1] \text{ and } ((s', j), \sigma, \lambda) \in [S_2]\}\\
&=\{(\tau, \sigma, \lambda) \mid (\tau, \sigma, (s', j)) \in [S_1] \text{ and } ((s', 0), \sigma^{(j)}, \lambda) \in [S_2]\}\\
&=\{((s,0), \sigma, \lambda) \mid (\sigma, s, S_1, \emptyset) \longrightarrow (\sigma^{(j)}, s', \checkmark, \emptyset) \text{ and}\\
&\qquad (\sigma^{(j)}, s', S_2) \downarrow \text{ and } \lambda = \star \text{ or}\\
&\qquad\qquad \lambda \text{ is final state after } S_2 \text{ terminates.}\}
\end{aligned}$$

We then observe that the condition of *always eventually ticking* can be written as *always eventually the oracle progresses*, i.e., $\forall n. \exists m. (\sigma^{(k)}, s_n, S_{1,n}, \emptyset) \longrightarrow_m (\sigma^{(k+1)}, s_{n+1}, S_{1,n+1}, \checkmark)$. Therefore, given the initial state s and the statement S_1, the oracle stream σ is non-terminating but accepting, which corresponds to the elements $((s,0),\sigma,\star) \in [S_1]$. Thus,

$$[\![S_1 \,;\, S_2]\!] = \{\sigma \mid (\tau, \sigma, \lambda) \in [S_1 \,;\, S_2]\}$$

Case **while** b **do** S **od**. Suppose that $[\![S]\!] = \{\sigma \mid (\lambda, \sigma, \tau) \in [S]\}$. We distinguish two kinds of valid runs in **while** b **do** S **od**: either the run terminates; or the run does not terminate but synchronize infinitely often. Using standard proof methods, we can show that the first class of runs is captured by $\bigcup_{k=0}^{\infty}[(\textbf{while } b \textbf{ do } S \textbf{ od})^k]$, which contains some σ such that $(\sigma, s, \textbf{while } b \textbf{ do } S \textbf{ od}) \downarrow$. For the second class of accepting runs, we show that the class coincides with the set $\bigcap_{k=0}^{\infty} \text{prog}(S,k)\dagger$.

We use the syntax $S^{\underline{k}}$ as defined earlier. Let $[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!]$ be the set $\{\sigma \mid (\sigma, s, \textbf{while } b \textbf{ do } S \textbf{ od}) \downarrow\}$ defined as the union of two sets: F which is the set of finite runs, and I which is the set of infinitely productive runs. We have $[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!] = F \cup I$, with

$$\begin{aligned}
F &= \bigcup_{k=0}^{\infty} \{\sigma \mid (\sigma, s, (\textbf{while } b \textbf{ do } S \textbf{ od})^k) \downarrow\}\\
&= \{\sigma \mid (\lambda, \sigma, \tau) \in \bigcup_{k=0}^{\infty} [(\textbf{while } b \textbf{ do } S \textbf{ od})^k)]\}
\end{aligned}$$

and

$$\begin{aligned}
I &= \{\sigma \mid \exists s. \forall k. \exists t, t', j. (s, \sigma, S^{\underline{k}}, \emptyset) \longrightarrow (t, \sigma^{(j)}, \checkmark, \emptyset) \wedge\\
&\qquad \exists m, n > 0. (t, \sigma^{(j)}, S^{\underline{m}}, \emptyset) \longrightarrow (t', \sigma^{(j+n)}, \checkmark, \emptyset)\}\\
&= \{\sigma \mid \exists \lambda. \forall k. \exists t, t', j. (\lambda, \sigma, (t, j)) \in [S^{\underline{k}}] \wedge\\
&\qquad \exists m, n > 0. ((t, j), \sigma, (t', j+n)) \in [S^{\underline{m}}]\}\\
&= \{\sigma \mid \exists \lambda. \forall k. \exists t, t', j. (\lambda, \sigma, (t, j)) \in [S^{\underline{k}}] \wedge\\
&\qquad \exists m > 0. ((t, j), \sigma, \tau) \in [S^{\underline{m}}] \cap P\}\\
&= \{\sigma \mid \exists \lambda. \forall k. (\lambda, \sigma, \tau) \in [S^{\underline{k}}] \circ (\bigcup_{m=1}^{\infty} [S^{\underline{m}}] \cap P)\}\\
&= \{\sigma \mid (\lambda, \sigma, \star) \in \bigcap_{k=0}^{\infty} [S^{\underline{k}}] \circ (\bigcup_{m=1}^{\infty} [S^{\underline{m}}] \cap P)\}
\end{aligned}$$

where P is the set of progressive runs. Observe that $I \cap F = \emptyset$. Then, $[\![\textbf{while } b \textbf{ do } S \textbf{ od}]\!] = F \cup I$ with

$$\begin{aligned}
F \cup I &= \{\sigma \mid (\lambda, \sigma, \tau) \in \bigcup_{k=0}^{\infty} [(\textbf{while } b \textbf{ do } S \textbf{ od})^k)]\} \cup\\
&\qquad \{\sigma \mid (\lambda, \sigma, \star) \in \bigcap_{k=0}^{\infty} [S^{\underline{k}}] \circ (\bigcup_{m=1}^{\infty} [S^{\underline{m}}] \cap P)\}\\
&= \{\sigma \mid (\lambda, \sigma, \tau) \in [\textbf{while } b \textbf{ do } S \textbf{ od}]\}
\end{aligned}$$

□

Proof sketch for Lemma 3.2. First, observe that the message buffer of a component changes according to its put and get operation: in any state (ρ, μ, s, S), μ collects the assignments resulting from previous puts and gets. Then, the transition relation $\Rightarrow$ on configurations makes a put and a get operations to coincide with a transition on a port configuration. Therefore, after each put or get on a port p, if the resulting configuration $\mathcal{M}(p)$ of port p is consistent and the component is in a state (ρ, μ, s, S), we have $\mathcal{M}(p)(\rho) = \mu(p)$. By Lemma 3.1, and given that initially the configuration of each port is consistent, we can conclude that the memory buffer μ coincides with the port store after every operation, including after the synchronization. □