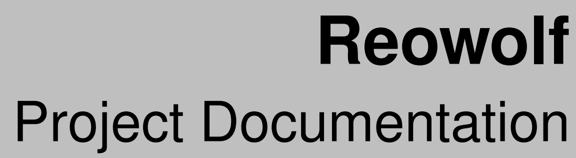

Reowolf

Project Documentation

DRAFT

Christopher A. Esterhuyse
Hans-Dieter A. Hiep
Centrum Wiskunde & Informatica

January, 2020
Draft version

DRAFT

Contents

1 Introduction **1**
1.1 Audience . . . 1
1.2 Vision and Mission . . . 1
1.3 Goals . . . 2
1.4 Milestones . . . 3

2 Design Overview **5**
2.1 Terminology . . . 5
2.2 Connectors . . . 7
2.2.1 Life Cycle . . . 7
2.2.2 Connector Setup . . . 7
2.2.3 Connector Communication . . . 8

3 Application Programming Interface **11**
3.1 Definitions . . . 11
3.2 Connector Programming . . . 12
3.2.1 Setup and Configuration . . . 12
3.2.2 Data Transfer and Synchronization . . . 14
3.3 Language-Specific API . . . 14
3.3.1 The C API . . . 14
3.3.2 The Java API . . . 15

4 Protocol Description Language **17**
4.1 Notation . . . 17
4.2 Context-Free Grammars . . . 18
4.2.1 Lexical Analysis . . . 18
4.2.2 Abstract Syntax Tree . . . 25
4.3 Grammar Rules . . . 33
4.3.1 Well-formedness . . . 33
4.3.2 Resolution . . . 35

5 Protocol State Representation **39**
5.1 The Table Model . . . 39
5.2 Protocol Semantics . . . 40
5.2.1 Internal Language . . . 40
5.2.2 Denotation . . . 41
5.2.3 Protocol Behavior . . . 43
5.3 Connector Semantics . . . 44

DRAFT

5.3.1 Setup Phase 44
5.3.2 Communication Phase 44
5.3.3 Correctness 45

6 Reference Implementation **47**
6.1 Goal 47
6.2 Implementation Overview 48
6.2.1 Protocol Components as Actors 48
6.2.2 Distributed Oracle Search 49
6.2.3 Speculative Component Execution 49
6.2.4 Speculative Message Passing 51
6.2.5 Oracle Consensus 52
6.2.6 Ternary Domain Port Predicates 54
6.3 Implementation Specifics 54
6.3.1 Setup procedure 54
6.3.2 Component Controllers 56
6.3.3 Cooperative Actor Scheduling 57
6.3.4 sec:cooperative$_{a}ctor_{s}cheduling$ 58
6.3.5 Meta Protocol Messages 59
6.4 Opportunties for Exploration 60
6.4.1 Actor Reconfiguration 60
6.4.2 Custom Transport Protocol 60
6.4.3 Component Precompilation 61
6.4.4 Shared Memory Value Aliasing 61
6.4.5 Dense Predicate Encoding 62
6.4.6 Hierarchal Speculative Branch Structure 62
6.4.7 Localized Synchrony 62
6.4.8 Arbitrary Oracle Variables 62
6.5 Example Connectors 63

DRAFT

Chapter 1

Introduction

This document serves as the documentation and specification of the Reowolf project, aiming to provide connectors as a generalization of BSD-sockets for multi-party communication over the Internet. This document is a work in progress, and is extended further as the project develops.

The Reowolf project started in November, 2019. It is supported by the Next Generation Internet Privacy & Trust Fund (NGI-Zero PET), a fund established by NLnet with financial support from the European Commission's Next Generation Internet programme, under the aegis of DG Communications Networks, Content and Technology under grant agreement No. 825310 (Horizon 2020). NLnet Foundation is a non-governmental public benefits organization that "has been financially supporting organizations and people that contribute to an open information society. It funds those with ideas to fix the internet."

1.1 Audience

Readers are expected to be somewhat familiar with Internet architecture and basic protocols such as IP, ICMP, TCP, UDP; existing routing infrastructure and services such as DHCP, DNS; systems programming using the C language with POSIX or UNIX-like operating systems; network programming experience using sockets and application layer protocols such as HTTP, FTP, SMTP, or any other application layer protocol typically implemented using sockets; and experience using a programming language similar to C or Java.

Chapters whose contents build upon existing literature include a final section on bibliographical remarks, giving readers an entry point to understanding more technical details for those missing relevant background knowledge.

1.2 Vision and Mission

As more Internet traffic is encrypted to enhance the privacy of its users, its nature is less insightful to network operators. This might lead to inefficient routing of traffic, the inability to monitor for abuse, and unfair networking practices. The vision of this project is to build new communication methods that increase user privacy and trustworthiness of Internet infrastructure, aimed at separating private message contents from the description of network communication. In the end, this might result in an alternative to deep packet inspection for network operators that better

DRAFT

protects privacy for users, and higher performance of complex data streaming applications by increasing throughput and decreasing latency.

Reowolf aims to replace BSD sockets with *connectors* for communication on the Internet. Connectors generalize sockets, facilitating multi-party communication according to an explicitly-expressed *protocol description* in Reowolf's Protocol Description Language (PDL). The use of these textual protocol descriptions allows applications to express their requirements on the nature of the traffic to the underlying network communication infrastructure at a higher level of abstraction. This abstract representation serves a dual purpose: (1) applications are able to relegate the implementation of coordination logic to Reowolf itself, becoming more modular and re-usable, and (2) protocol descriptions act as a vehicle for communicating the abstract requirements and intent of communication sessions to humans and machines alike, allowing for optimizations in middleware all along the network path, and the development of standard libraries for protocols with useful, verified properties.

From a more practical point of view, Reowolf allows the implementation of application-specific firewalls. In current practice, it is unknown to operating systems what network behavior applications do expect and wish to handle normally, versus what network behavior is unwanted or unsolicited. By using Reowolf, applications declare using the Protocol Description Language all permitted interactions over one or more data streams; other behavior is explicitly disallowed. Thus, traffic that does not conform to the protocol description is necessarily unwanted, and Reowolf implementations can use this information for increasing visibility in protocol violations that may signal network intrusion or defects. Moreover, this inspection of traffic can happen also within the network infrastructure, not only on the edge of the network. If applications apply this technique, a more reliable network infrastructure in the long run may result.

1.3 Goals

The Reowolf project's main objective is the design and implementation of connectors. We first focus on functional aspects to establish a baseline for correctness and functionality. Specifically, our core goals are:

1. designing the connector application programming interface (Chapter 3),
2. designing the protocol description language (Chapter 4),
3. defining and relating the semantics of protocols and connectors (Chapter 5),
4. designing and implementing a reference implementation,
5. demonstrating functionality through a number of detailed examples.

After we are effectively able to demonstrate functionality, we will focus on reliability, efficiency, and backwards compatibility. What follows explores various avenues of performance optimization and features for increased safety to demonstrate Reowolf's potential. More specifically, further goals are:

1. demonstrating the capability for performing per-application and protocol-based network traffic monitoring.
2. demonstrating a number of optimization opportunities, possibly applicable to only a subset of the full protocol description language.

DRAFT

3. maintain backwards compatibility with existing Internet infrastructure, and existing socket-based applications.

This project aims to produce a high-quality professionally engineered reference implementation, that will form the groundwork for future research assignments and development effort.

1.4 Milestones

The project has eight milestones listed below.

1. Design of API and protocol language
    - 1.1. Design of connector API
    - 1.2. Design of protocol description language
    - 1.3. Design of protocol state format and representation
2. Naïve implementation
    - 2.1. Design of naïve synchronous communication
    - 2.2. Implementation of distributed consensus
    - 2.3. Implementation of distributed synchronizer
    - 2.4. Revised API and PDL design
    - 2.5. Example programs
3. Interoperability
    - 3.1. Extend PDL with TCP and UDP primitives
    - 3.2. Reimplementation of socket API
    - 3.3. Testing interoperability
4. Deviation detection
    - 4.1. Design algorithm for deviation detection
    - 4.2. Design middleware for monitoring
    - 4.3. Implementation of deviation detection
5. Local optimization
    - 5.1. Collect relevant use-cases
    - 5.2. Implementation of shared-memory synchronization
    - 5.3. Implementation of reconfiguration
6. Middleware optimization
    - 6.1. Collect relevant use-cases
    - 6.2. Implementation of connector merging
    - 6.3. Implementation of dynamic routing
7. Benchmarking

DRAFT

   7.1. Experimental set-up and design
   7.2. Perform measurements
   7.3. Compare performance gains and overhead

8. Documentation

   8.1. Write tutorial
   8.2. Write guide for porters
   8.3. Collect comprehensive documentation

Each milestone has associated tasks, see Table **??** for the currently completed tasks.

ID	Deadline	Type	Detailed Description
1	2 dec 2019		
1.1		Document Chapter 3	**Design of connector API** The design of the API consists of an application programming interface (API): connector creation, configuration, data transfer, synchronization, error handling. Includes basic data structures and operations. Each operation includes an informal functional specification. The document describes the API for the C and Java language.
1.2		Document Chapter 4	**Design of protocol description language** The protocol description language is defined using a language grammar, describes the language parser, the intended language semantics. The document includes example protocols, and their intended behavior.
1.3		Document Chapter 5	**Design of protocol state format and representation** The protocol state and representation is the conceptual model for communication (the table format). This document describes how the table format and the API are related, and describes how protocols are represented in the conceptual model.

Table 1.1: Completed Tasks

DRAFT

Chapter 2

Design Overview

This chapter outlines the essential features and capabilities of Reowolf as a communication method for networked applications. Concepts are introduced in a top-down fashion, identifying their roles in the system at large. For topics warranting more detailed explanation, we refer the reader to the relevant chapters.

2.1 Terminology

Reowolf sits at the union of the domains of discourse of computer networking, coordination languages, and formal logic. By design, the conventions set by BSD-style sockets are followed. Reowolf uses familiar terminology for concepts already present in each of its domains. However, certain terms have a different meaning in each domain. To avoid such ambiguity, this section enumerates the most essential terms and gives a short definition.

- **Protocol**

  1. *Physical protocol*, protocols that enable electromagnetic signal exchange, for example Fast Ethernet, Gigabit Ethernet and Wi-Fi.
  2. *Internet protocol*, protocols that enable exchange of data on an established link, for example Internet Protocol (IP) and Internet Control Message Protocol (ICMP).
  3. *Transport layer protocol*, protocols that are implemented on top of the network layer, for example Transmission Control Protocol (TCP), User Datagram Protocol (UDP).
  4. *Application layer protocol*, protocols implemented on top of the transport layer, for example HTTP, FTP, SMPT.
  5. *Logical protocol*, a logical specification of the constraints on the observable behavior of data streams, defined in Chapter 4.

- **Port**

  1. *Physical port*, e.g. an Ethernet port as found on switching devices.
  2. *Transport layer port*, e.g. as used in transport layer protocols TCP and UDP.
  3. *Logical port*, a point of streaming data that can be constrained by a logical protocol.
  4. *Input port*, a logical port from which a protocol may causally receive data.

DRAFT

   5. *Output port*, a logical port to which a protocol may send data.

- **Component**
   1. *Physical component*, such as an electrical or optical wire, antenna, or device.
   2. *Logical component*, an interface with input and output ports and a logical protocol describing the observable behavior at the ports of its interface.
   3. *Primitive component*, a logical component where its logical protocol is described by a program specification (see Chapter 4).
   4. *Composite component*, a logical component constructed as a composition from other logical components, possibly recursively defined (see Chapter 4).
   5. *Native component*, a logical component of which the logical protocol is unknown. Typically implemented by a process running one some host (see Chapter 3).
- **Node**
   1. *Node*, a vertex in a mathematical graph structure.
   2. *Physical node*, a physical machine in a network.
   3. *Peer*, the entity that is uniquely identified by an Internet address and reachable via IP. A peer is not necessarily one physical machine, as the same physical machine can have multiple IP addresses, and the same IP address can be allocated to multiple physical machines (cf. DNS root server anycast).
   4. *Reo node*, a logical component with variable number of input ports and variable number of output ports, such that at most one input port fires at a time and its data is synchronously replicated to all output ports.
- **Connector**
   1. *Connector handle*, obtained by native applications when executing programs that make use of the application programming interface, defined in Chapter 3.
   2. *Connector*, an abstraction of the communication session between a set of peers. The behavior of a connector is prescribed by a logical protocol.
   3. *Connector component*, a logical component. After a connector is fully connected, we refer to the connector component to mean the composition of all configured logical protocols. The connector component is unique after a connector is fully connected, since all peers are known.
- **Synchronous**
   1. *Synchronous operations* block until some return event has occurred.
   2. *Synchronous events* occur within the same synchronous round of a connector.
   3. Two components must *synchronize* as soon as a port that is channeled from one component's output to the other component's input fires.
- **Imp**
   1. *Interface Message Processor* (historically) a distinct physical node used as a gateway to the ARPANET network.
   2. An entity that implements the interface between a native application and the rest of a connector at run-time. The behavior of a connector is realized through communication among imps.

DRAFT

2.2 Connectors

For applications making use of BSD-style sockets, sockets represent both (1) the application's 'handle' on the communication, and (2) a logical endpoint for data exchange with the peer. Applications that require multi-party communication or careful coordination with peers build this functionality into the application itself on top of sockets.

Reowolf introduces the *connector* as a stateful communication instance between an arbitrary number of peers. For groups it peers, it becomes necessary to reason about the nature of coordination. Reowolf takes this notion to the extreme, adopting the responsibility of implementing the coordination logic between a connector's ports. Applications express this logic as a *protocol description*, expressed in *Protocol Description Language* (PDL), and provide it to Reowolf during a connector handle's setup phase; this is referred to as *configuration*. Chapter 4 defines PDL, and explains the semantics of the protocols that define the specific behavior of connectors at runtime.

2.2.1 Life Cycle

Per communication peer, Reowolf's functionality comes to life as a cooperation between the application 'above' and the network 'beneath'. From an application's perspective, the functionality is always provided in the context of a particular *connector*. Both abstractly, and as it appears in the application code, connectors represent the communication in which the application plays a role. Each such connector has a two-phase life cycle over the program's runtime: setup and communication.

The setup phase creates and initializes the state of the connector. A local connector handle is instantiated paramaterized with a protocol description which describes the application's *local view* on the connector. Secondly, the connector is instantiated by mapping the logical names in the description to explicit domain names or network addresses. The connector handle is then connecting, in which it seeks out external nodes and joins other peers in establishing a shared connector. Setup phase blocks until all peers have communicated and established a consistent initial state with which they can begin communication. In this way, the entire distributed system can be seen to transition from setup phase to communication phase as a unit.

Once connected, the application is able to interleave their arbitrary computation with data exchange by sending and receiving data into *ports* of a connector. At this stage, the application code need not reflect the protocol description in any way; at runtime, Reowolf will enforce the adherence of the application to the protocol, collapsing the system to an error state if necessary. In this fashion, applications are able to delegate the task of coordination primarily to Reowolf, allowing the protocol and application-logic to be inspected and manipulated independently.

Chapter 3 provides Reowolf's API, detailing the setup of connectors and how connectors and ports are used in data exchange.

2.2.2 Connector Setup

PDL is compositional in nature; new protocols can be constructed from other, smaller protocols. Such protocols are marked explicitly in PDL as *composite* structures. Chapter 4 goes into detail about the relationship between a composite protocol and its constituents. Here, it suffices to understand that Reowolf's notion of composition is defined such that the behavioural constraints of protocols are preserved under composition. This design philosophy is important to Reowolf, as is allows actors (human or otherwise) to reason about the behavior of a complex system by extrapolation from its constituent components, and vice versa. This reasoning is applicable to

DRAFT

Reowolf connectors during their lifetime; applications define their *local* view of the connector in PDL, and rely on Reowolf to derive the *global* system by composing the descriptions supplied by all peers. While the global protocol is known to the Reowolf implementations of all peers, and is considered public information from the network perspective, applications can be implemented in terms of their local view of the protocol only. Section 5.2 defines the composition of Reowolf protocol descriptions more precisely.

2.2.3 Connector Communication

Once set up, a connector is able to act as the communication medium between its set of peers. Applications and the shared Reowolf runtime work together to bring the distributed system to life.

Application View

Chapter 3 explains Reowolf's API in detail; here, it suffices to provide an intuition behind the role the application plays in the communication.

A network of connected peers consist of a set of machines, each of which rely on Reowolf to organize network communications into protocol-adherent data flow. Between arbitrary sequences of local computation, applications may send data into, or receive data from the connector. Data exchange operations are paramaterized with *port handles*, corresponding to logical ports in the connector's configured PDL. By specifying a port, one application may maintain an arbitrary number of logical endpoints in the connector. The notion of *synchrony* is central to Reowolf, as is reflected in PDL. Applications are able to prepare batches of port operations for Reowolf to perform synchronously, i.e., within the same logical timestep. In this fashion. The distributed Reowolf instance cooperates with the various connected applications to move the state of the connector forward.

Reowolf View

During the setup phase of a connector, the Reowolf implementations of the set of peers establish a connection. Connection imposes a traditional synchronization barrier on peers, such that the system at large enters the connection phase as a unit. Reowolf models the state of the distributed system as a stepwise transformation, mutating the state of the shared connector one discrete *round* at a time. The manifestation of this cohesive, centralized view depends on the implementation. For example, a naïve implementation keeps all peers' rounds synchronized with explicit barriers each round. Furthermore in this explanation, we ignore Reowolf's distributed nature.

Section 5 defines the semantics of protocols, and goes into detail about the relationship between a protocol (as defined in PDL) and the observable behavior permitted to connectors instantiated from them. Here, it suffices that the connectors can be observed to transition through a sequence of states, each of which defines which values are observable at ports and the connector's local variables. Protocol descriptions essentially constrain which states may be observed as a function of those that were observed before. For example, a protocol may define that, always, port x and port y must observe the same value. At runtime, Reowolf might unfold a trace through the connector's state-space one round at a time: first $x = y = 6$, then $x = y = 0$, and then $x = y = 2$. Fundamentally, the task of Reowolf is that of a *constraint solver*, selecting valuations arbitrarily insofaras the semantics of the protocol are preserved. Applications are able to impose constraints of their own at runtime by performing port operations. For example,

an application putting value 5 at port x can be viewed as the inclusion of the constraint, $x = 5$. In cases where multiple valuations are possible, Reowolf makes a selection non-deterministically. A connector for which no valuation is possible has become inconsistent, and cannot continue to run. In some cases, this is simply the result of a programming error in the PDL itself, necessitating protocol validation and verification. In other cases, the actions of applications at runtime may lead to inconsistencies at runtime, necessitating real-time deviation detection and recovery.

DRAFT

DRAFT

Chapter 3

Application Programming Interface

Socket programming revolves around the creation, management, and use of sockets to exchange data with a single peer. Programmers take for granted that another application will create the counterpart socket, resulting in the manifestation of a shared, two-party communication medium at runtime, which we will call the 'socket-pair'. Applications rely on sockets as an abstraction of the rest of the socket-pair. Communication is represented abstractly, appearing to the application as the result of interactions with the socket itself. The underlying implementation is relied upon to translate these abstractions into management of underlying resources (e.g., maintaining distributed state, message-passing with IP packets, etc.). Applications are provided some control over the details of this implementation via the socket's configuration; typically, sockets can be configured with additional socket options to specialize the underlying behavior. For example, TCP supports the inclusion of the NODELAY flag to disable Nagle's algorithm for sacrificing round-trip-time on network messages to reduce messaging overhead.

Reowolf introduces *connectors* as a replacement for socket-pairs to generalize the number of connected peers (previously two) and the number of communication endpoints per application (previously one). The connector API is designed to be similar to that of sockets as much as possible. Applications represent their connector with a *connector handle* structure, and enable applications to perform abstract data exchange by reading and writing bytes to logical endpoints as before. Owing to the increased configuration space for these more complex connectors, Reowolf defines *Protocol Description Language* (PDL) for defining an application's requirements for the connector's behavior and communicating it to Reowolf during its configuration.

The API provided in this section is intended to be minimal; elaborations may be introduced to build more convenient and user-friendly APIs, using these simple procedures as their foundation.

3.1 Definitions

This section clarifies terminology to be used throughout the chapter.

1. *network address*
   This term generalizes the notion of a physical address over the internet. This generalizes over the available methods for sufficient capability to address host machines without ambiguity. For example, this may be implemented as an IPv6 address, or the tuple of (IPv4, port), where *port* is a 16-bit integer commonly used in the transport layer (e.g., TCP) for disambiguating beyond the limited capabilities of IPv4.

DRAFT

2. *port handle*
   Port handles are pervasive in Reowolf's API for representing logical Reowolf ports as they occur in the connector's configured PDL description. Concretely, they are non-negative integers, indicating the index of the port as parameter in the main component.

3. *port binding*
   During the setup phase of a Reowolf connector handle, the protocol's logical ports are *bound*, making clear their relationship to the application and how they are connected to peers' ports. A port given a *native* binding is exposed for data exchange to the application. *Passive* and *active* ports are exposed to the outside world, relying some application to provide a counterpart port to complete the coupling once the connector connects to its peers. Both passive- and active-bound ports are provided an address, and differ only in the means of resolving the connection: the passive port will listen passively for an incoming connection, while the active port will actively initiate an outgoing connection.

4. *message*
   The type of values being exchanged between the application and the Reowolf connector. Passed into ports with the `batch_put` procedure, and retrieved from ports with the `batch_get` procedure.

3.2 Connector Programming

Chapter 2.2 established that Reowolf connectors are inherently stateful, to reflect the stateful nature of the protocols that provide their definitions. In this section, we introduce an API for connectors such that the handles themselves are stateful also. For simplicity in this section, the API consists simply of a set of procedures; their relationship through effects on the connector's state are made explicit through text and error conditions. This approach is unlikely to be the best practice for all programming languages, requiring the design to be adapted. For example, procedures to do with communicating session-specific metadata of the configured protocol description may be exposed as the ubiquitous *builder pattern*, which creates a temporary builder object for incremental construction and applies all changes to the connector handle's state at once with a final `build` method.

Reflecting the distinct phases of a Reowolf connector outlined in Section 2.2.1, connector handles exist in one of four states: *uninitialized*, *connecting*, *communicating* or *closed*.

3.2.1 Setup and Configuration

Before becoming usable for communication, a connector handle must be created and set up using the `connector_configure` and `connector_connect` operations in succession.

As part of the setup phase, the application must invariably supply a protocol description, expressed in PDL. Sections to follow apply these concepts to particular programming languages, including an explanation of how textual PDL is represented in the application. Here, it suffices to assume that the language is able to represent some textual PDL loaded into program memory by a handle that can be passed to Reowolf for inspection.

1. `connector_create`
   A new connector handle is created and returned in the fresh state. An error arises if the system has insufficient resources to support a new protocol handle.

DRAFT

2. `connector_configure`
   An existing protocol handle in fresh state is configured with a textual protocol description, expressed in PDL. The procedure is parameterized by (1) a connector handle, (2) the set of logical port names local to the application, and (3) a mapping from logical port occurring in the connector's configured PDL to network addresses. The connector handle moves to the connecting state.

   An error arises if the provided PDL is not well-defined. Concretely, these errors correspond with the semantics of PDL itself. For example, if a component is defined with two arguments with the same identifier. For more detailed information, see Chapter 4.

3. `port_bind_native`
   An existing connector in the connecting state expresses that the given port is accessible to the application for data exchange. The procedure is parameterized by (1) a connector handle, and (2) a port handle.

   An error arises if the connector is not in the connecting state, the port handle is invalid in the connector's configured PDL, or the port handle is has previously been bound.

4. `port_bind_passive`
   An existing connector in the connecting state expresses that a given logical port is exposed for composition with that of another application, marked to passively wait for the counterpart point to initiate the coupling at the given address. The procedure is parameterized by (1) a connector handle, (2) a port handle, and (3) the network address to which the port will bind, i.e., analogous to a TCP connection.

   An error arises if the connector is not in the connecting state, the port handle is invalid in the connector's configured PDL, or the port handle is has previously been bound.

5. `port_bind_active`
   An existing connector in the connecting state expresses that a given logical port is exposed for composition with that of another application, marked to actively initiate the coupling at the given address with a passive counterpart. The procedure is parameterized by (1) a connector handle, (2) a port handle, and (3) the network address to which the port will connect, i.e., analogous to a TCP connection.

   An error arises if the connector is not in the connecting state, the port handle is invalid in the connector's configured PDL, or the port handle is has previously been bound.

6. `connector_connect`
   An existing connector in the connecting state is connected to its peers and completes its setup phase, and entering the communicating state. The procedure is parameterized by a connector handle. The connect call returns when all ports have been connected to some peer, and likewise with any of those peers' ports.

   Errors arise if there is a problem finding peers and composing with their connectors. Reasons for this include invalid, unresolvable, or unreachable network addresses used in port bindings.

7. `connector_close`
   A connector handle is closed, freeing its underlying resources. It is an error to use an invalid protocol handle, or one in closed state. The protocol handle moves to the closed state.

DRAFT

3.2.2 Data Transfer and Synchronization

A protocol handle in the communicating state may be used for the exchange of data. In addition to 'communicating', these connector handles maintain an internal state of a 'staging area' for synchronous port operations. Concretely, each protocol handle maintains a set of *batches*, where each batch is a set of staged port operations, and where one batch at a time is currently *selected*. Initially, a connector handle has a singleton set of batches whose only element is empty and selected. These batches provide a means for the application to express a non-deterministic choice; The `connector_sync` call finalizes all batches and blocks until Reowolf chooses one and completes the operations within. The returned result of `connector_sync` communicates which batch was chosen back to the application.

1. `batch_put`
   Inserts a tentative put operation for a given port into the connector handle's selected batch. The procedure is parameterized by (1) a connector handle, (2) the port handle, and (3) a message. An error arises if the provided port handle is not provided a native binding. An error arises if (1) the port handle is not defined for the connector at all, (2) not defined to have direction `in` for the connector, or (3) already occurs in the selected batch.

2. `batch_get`
   Inserts a tentative put operation for a given port into the connector handle's selected batch. The procedure is parameterized by (1) a connector handle, (2) the port handle. The procedure returns a value of type message. An error arises if the provided port handle is not provided a native binding. An error arises if (1) the port handle is not defined for the connector at all, (2) not defined to have direction `in` for the connector, or (3) already occurs in the selected batch.

3. `batch_next`
   Inserts an empty batch into the connector handle's batch set and selects it. The procedure is parameterized by a connector handle.

4. `connector_sync`
   The connector handle's batches are submitted to the connector, and the call blocks until the synchronous round has been completed. Afterward, all batches are reset to their initial state, i.e., a set of batches whose only element is empty and selected. The procedure returns the index of the chosen batch as an integer. The procedure is parameterized by a connector handle. If the connector encounters an error during runtime, it is propagated to this procedure and returned to the application. For example, a protocol in inconsistent state.

3.3 Language-Specific API

This section specializes the abstract procedures of connector programming described in Section 3.2 to concrete programming languages. Suggestive naming of procedures and methods preserves the mapping from the former to the latter.

3.3.1 The C API

Reowolf's C API follows the paradigms of the language by relying on a simple, procedural style. State management of resources is relegated to the application programmer, and they are ex-

DRAFT

pected to read and adhere to the textual documentation that accompanies the API. Connector handles are represented by file descriptors (of the `int` type) as is the convention for UDP and TCP sockets. Fallable procedures facilitate the propagation of errors to the application by use of error codes; consequently, the majority of procedures have the return type `int`.

```
#include <sys/socket.h>

/* setup */
int reowolf_connector();
int reowolf_configure(int connector, char* protocol_description);
int reowolf_bind_native(int connector, int port);
int reowolf_bind_passive(int connector, int port, sockaddr* address);
int reowolf_bind_active(int connector, int port, sockaddr* address);
int reowolf_connect();
int reowolf_close_connector(int connector);

/* communication */
int reowolf_put(int connector, int port, char* msg, int msg_len);
int reowolf_get(int connector, int port, char* msg, int msg_len);
int reowolf_close_port(int connector);
int next_batch(int connector);
int sync(int connector);
```

3.3.2 The Java API

Reowolf's Java API is more object-oriented than that of the C API, revolving around methods of the `Connector` class, with the exception of creation, which is represented by Java's canonical means for object instantiation. Fallable methods may throw exceptions for the application to handle.

```
import java.net.SocketAddress;

public class Connector {
    /* setup */
    public Connector();
    public void configure(String protocolDescription)
            throws ConfigurationException;
    public void bindNative(int portHandle)
            throws BindException;
    public void bindPassive(int portHandle, SocketAddress address)
            throws BindException;
    public void bindActive(int portHandle, SocketAddress address)
            throws BindException;
    public void connect() throws ConnectionException;
    public void close() throws CloseException;

    /* communication */
    public void put(int portHandle, byte[] message)
            throws CommunciationException;
```

DRAFT

```
    public byte[] get(int portHandle)
            throws CommunciationException;
    public void nextBatch();
    public void sync() throws CommunciationException;
}
```

DRAFT

Chapter 4

Protocol Description Language

This chapter defines Protocol Description Language (PDL) in a bottom-up fashion, oriented around the structure of its syntax. Section 4.3 and onward, the definition of well-formedness criteria for protocols is introduced. In Chapter 5 to follow, the semantics of protocols is explained in relation to that of connectors, laying the groundwork for the definition of a correct connector implementation.

4.1 Notation

For specifying syntactical structure we employ two notation formats: one for specifying the lexical grammar, another for specifying the abstract syntax tree.

Augmented Backus-Naur Form (ABNF) is a *context-free grammar* specified in RFC5234. It is used for specifying the lexical grammar of the Protocol Description Language. ABNF consists of a number of *production rules*. Each production rule is given a case-insensitive *name* and a defining *expression*.

Expression in ABNF are simple or complex. Simple expressions are formed by a *terminal value* or by a name of a production rule. Complex expressions are formed by the operations: *concatenation* or *alternative* of two expressions, *repetition* with optional minimum and maximum occurrences, *grouping* an expression in parentheses, or an *optional* expression.

```
CRLF           = CR LF
               ; Concatenation of CR and LF rules
HTAB           = %x09
               ; Horizontal tab (encoded value)
LF             = %x0A
               ; Line feed
CR             = %x0D
               ; Carriage return
SP             = " "
               ; Space character (literal value)
VCHAR          = %x21-7E
               ; Visible characters in 7-bit ASCII
WSP            = SP / HTAB
               ; Alternative of SP and HTAB rules
```


DRAFT

```
ALPHA           = %x41-5A / %x61-7A
                ; Value range for letters A-Z or a-z
DIGIT           = %x30-39
                ; Value range for digits 0-9
```

The terminal values are given either directly (e.g. `%x09`) or by specifying a *value range* (e.g. `%x21-7E`). The above example shows rules which are commonly used in ABNF. The sentence following a semicolon are comments that describe what the expression above it matches as terminal value.

We specify the abstract syntax tree using pseudo-code that should feel familiar to readers who have seen Java, C++ or the Interface Description Language (IDL) published by Object Management Group (OMG) before. We focus on specification in an object-oriented manner: elements in the abstract syntax tree are objects. Using this pseudo-code, we define *interfaces* which may inherit from (multiple) other interfaces, and *(abstract) classes* that may extend one super class and inherit from (multiple) interfaces, and have *methods* that provide access to its *attributes*.

```
interface InputPosition {
    String getFilename();
    int getLine();
    int getColumn();
    int getOffset();
}
class Element {}
class SyntaxElement extends Element {
    InputPosition getPosition();
}
```

Above is an example interface and two classes. These represent the position of a source file, and a syntactical element that has a position in a source file. The first interface, `InputPosition`, has four attributes: filename, line number, column number, and absolute offset. The second interface, `SyntaxElement`, has one attribute: a reference to an object that implements the first interface. We shall assume that we treat returned objects as *immutable*, not to be modified by users of these interfaces and classes.

In some superficial way, our interface definitions can also be seen as a context-free grammar. Each interface or class is a production rule. Inheritance specifies alternative choices. Interface attributes are the names of the concatenation of other productions.

4.2 Context-Free Grammars

4.2.1 Lexical Analysis

The lexical analysis of the Protocol Description Language consists of two conceptual layers: lower-level character input handling and higher-level syntax grammar. Processing of these two layers can happen in separate phases, or at the same time. We specify the lexical analysis by defining a grammar. Our goals in specifying this grammar are: unambiguous parsing and keeping implement to a minimum.

DRAFT

Input Handling

The following rules are used mainly to process character input data, filtering out comments and white space. We assume the input data is consumed per octet (8 bit). Input data matching these rules is ignored and not significant.

```
; The following terminals are ignored by the parser
cwb            = cw
               ; cwb must match a word boundary
cw             = comment / WSP / newline
comment        = line-comment / block-comment
newline        = CRLF / LF
               ; Special treatment of new lines
line-comment   = "//" *(WSP / VCHAR) newline
block-comment  = "/*" *block-no-eob "*/"
               ; A block comment may contain anything except "*/"
block-no-eob   = "*" (%x00-2E / %x30-FF)
               / (%x00-29 / %x2B-FF)
               ; Content of block may include non-visible characters
```

The difference between rule `cwb` and `cw` is that the former rule is a marker that serves as a reminder that we expect a word boundary; both rules consume the same terminals. Note that `cw` stands for *comment* or *whitespace*, but a newline is also accepted.

We treat new lines specially. We accept either a line feed, or a carriage return immediately followed by a line feed. Input handling also accepts just carriage returns as new line, but must greedily consume a line feed if it is immediately following a carriage return. With respect to the position in the input data, all three forms result in the line count to increase by one, and the column to be reset to zero.

A comment is either a line comment that is followed only by visible characters in the ASCII range, or a block comment that may contain arbitrary binary data except for the `"*/"` sequence that marks the end of the comment. A line comment ends after the end of line is encountered.

```
; Common symbols
binry-operator = "||" / "&&" / "|" / "^" / "&" / "==" /
                 "!=" / "<=" / ">=" / "<" / ">" / "<<" /
                 ">>" / "+" / "-" / "*" / "/" / "%"
assgn-operator = "=" / "*=" / "/=" / "%=" / "+=" / "-=" /
                 "<<=" / ">>=" / "&=" / "^=" / "|="
unary-operator = "++" / "--" / "+" / "-" / "~" / "!"

; Common tokens
char-constant  = "'" 1*(SP / %x21-26 / %x28-7E) "'"
               ; Character literals are arbitrary length
int-constant   = DIGIT *(DIGIT/"a"/"b"/"c"/"d"/"e"/"f"/"x")
               ; Integer literals are arbitrary length
ident          = 1*ALPHA *(ALPHA / DIGIT / "_")
               ; An identifier is an alphanumeric sequence
               ; that starts with an alphabetical character.
```

We commonly recognize certain characters as *binary operators*, *assignment operators* or *unary operators*. Later syntactic structures make sure to specify operator precedence and associativity. Remark that `"+"` and `"-"` can be confused, as both are binary and unary operators.

DRAFT

We also recognize constants and identifiers. *Character constants* may appear as at least one character, enclosed within single quotes and restricted to the visible characters in the ASCII range except for the single quote itself. Similarly, *integer constants* are recognized whenever a digit is encountered and hexadecimal constants such as `0xEA` and octal constants such as `076` can be recognized: but processing of the constant value itself happens later. Identifiers, representing names within a protocol description, must start with an alphabetic character, but may be followed by any alphanumeric sequence of characters including underscores.

```
keywords        = "import" / "composite" / "primitive" / "in" / "out" /
                  "channel" / "msg" / "boolean" / "byte" / "short" /
                  "int" / "long" / "null" / "true"/ "false" /
                  "if" / "else" / "while" / "break" / "continue" /
                  "synchronous" / "return" / "assert" /
                  "goto" / "skip" / "new"
builtin         = "put" / "get" / "fires" / "create"
```

We recognize *keywords* and *built-in* functions. Keywords are used to structure definitions, parameters, types and statements. The important keywords are `"composite"`, `"primitive"` and `"synchronous"`. Built-in functions are special. When displaying protocol descriptions, keywords and built-ins are typically shown in a boldface type.

```
type            = "in" / "out" / "msg" / "boolean" /
                  "short" / "int" / "long" / ident
                ; A type is a keyword or an identifier
method          = builtin / ident
                ; A method is a built-in or symbolic
field           = "length" / ident
                ; A field is length or an identifier
```

We further distinguish three classes of identifiers, each permitting different keywords. A type is either an *input*, *output*, *message*, *boolean*, integer of various sizes, or otherwise identified. A *method* is either a built-in function or symbolic: the meaning of the latter depends on its context, and can refer either to a symbolic function or a component. A field is an identifier, but we recognize `"length"` as a special field. `"length"` is not a keyword, however.

Besides the specified keywords and built-ins, identifiers should not be equal to other keywords or built-ins to avoid confusion. The grammar does not explicitly formulates this requirement, but a parser must avoid keywords or built-ins to be used as identifiers.

Syntax Structure

Parsing a protocol description begins with the *file* production rule.

```
file            = *cw *(pragma *cw) *(import *cw) 1*(symbol-def *cw)
                ; A file comprises one or more definitions
pragma          = "#" [VCHAR *(WSP / VCHAR)] newline
                ; Pragma until end of line
import          = "import" 1*cw (ident *("." ident)) *cw ";"
                ; Import declarations
```

Each file consists of a number of *pragmas* that begin with `"#"`. These are used to convey version information of the Protocol Description Language, to allow for future language changes.

DRAFT

Following the pragmas, zero or more *imports* of qualified identifiers, being identifiers separated by dots. A reverse domain name convention for imports is used, similar to Java packages. An application may register PDL files using these qualified identifiers, as a basic module system for protocol descriptions. Imports that are starting with the `std` identifier are reserved for predefined standard imports, and cannot be registered.

```
symbol-def      = fun-def / component-def
                ; Symbol definition
fun-def         = type-annot 1*cw ident *cw fparams *cw block
component-def   = composite-def / primitive-def
                ; Component definition is either composite or primitive
composite-def   = "composite" 1*cw ident *cw fparams *cw block
primitive-def   = "primitive" 1*cw ident *cw fparams *cw block
```

The rest of a protocol description consists of *symbol definitions*. Every symbol definition consists of an identifier (the *symbol*), *formal parameters*, and a *block*. There are two kinds of symbol definitions: *function definitions* and *component definitions*. A function definition additionally consists of a return type. A component definition is either composite or primitive. Identifiers of symbol definitions must not be equal to built-ins.

```
type-annot      = type [*cw "[]"]
var-decl        = type-annot 1*cw ident
                ; Variable declaration (optionally an array)
fparams         = "(" *cw [var-decl *(*cw "," *cw var-decl)] *cw ")"
                ; Formal parameter list
```

The formal parameters of a symbol definition are given as a list within parentheses. Each formal parameter is of some type. Each formal parameter declares a variable that is in the scope of the block of a symbol definition. Types may be arrays, as indicated by `"[]"` following the type, but arrays of arrays are not allowed.

```
block           = "{" *(*cw (channel-decl/mem-decl)) *(*cw stmt) *cw "}"
channel-decl    = "channel" 1*cw ident *cw "->" *cw ident *cw ";"
mem-decl        = type-annot 1*cw ident *cw "=" *cw expr
                  *(*cw "," *cw ident *cw "=" *cw expr) *cw ";"
```

A block consists of zero or more *local declarations*, followed by zero or more statements. A local declaration is either a *channel declaration* or a *memory declaration*. Channel declarations start by the `"channel"` keyword, followed by two identifiers separated by an arrow `"->"`. Memory declarations start by a type annotation, followed by one or more identifiers with an *expression* that designate the initial value of the memory. The local variable declarations declare variables that are in scope of the statements and expressions in the same block.

```
stmt            = block
                / ident *cw ":" *cw stmt
                / "if" *cw pexpr *cw stmt [*cw "else" *cwb stmt]
                / "while" *cw pexpr *cw stmt
                / "break" *cwb [ident *cw] ";"
                / "continue" *cwb [ident *cw] ";"
                / "synchronous" *cw (fparams *cw stmt / block)
                / "assert" *cwb expr *cw ";"
```

```
                / "return" *cwb expr *cw ";"
                / "goto" 1*cw ident *cw ";"
                / "skip" *cw ";"
                / "new" 1*cw method-expr *cw ";"
                / expr *cw ";"
```

A *statement* is either a block, a *skip* statement, a *labeled* statement, an *if* statement, a *while* statement, a *break* statement, a *continue* statement, a *synchronous* statement, a *return* statement, an *assert* statement, a *goto* statement, a *new* statement or an expression. Except for blocks, labels and expressions, the first keyword designates the kind of statement. The `"else"`, `"break"`, `"continue"`, `"synchronous"`, `"return"` and `"assert"` keywords must be at a word boundary, to prevent matching a following keyword or identifier without white space in between. To prevent confusion between variable declarations and expressions, synchronous statements without formal parameters are always followed by a block statement.

A statement may be labeled so break, continue and goto statements can refer to the label.[1] These statements are collectively refered to as control flow disruption statements. Not all control flow disruptions are valid. Labeled break and continue statements are also found in Java. The labeled goto statement is found in C.

There is an explicit skip statement, in contrast to languages such as C, C++ and Java, because in PDL empty blocks are not allowed. The *else-branch* of an if statement is optional, but is eagerly associated to avoid ambiguity.

```
if(true) if(true) put(x); else put(y);
if(true){if(true) put(x); else put(y);}
if(true){if(true) put(x);}else put(y);  // !!!

if(i<j) if(j<k) if(k<n) put(x); else put(y); else put(z);
if(i<j){if(j<k){if(k<n) put(x); else put(y);}else put(z);}
if(i<j){if(j<k){if(k<n) put(x);}else put(y);}else put(z); // !!!
```

Above example consists of two triples of similar statements. The first two statements are parsed in the same way, but this differs from the third statement. Eager association of the else-branch means that inner if-statements associate to the else branch; accolades can be used to change the standard grouping, as is done in the third statement.

```
pexpr           = "(" *cw expr *cw ")"
expr            = assgn-expr
assgn-expr      = cond-expr [*cw assgn-operator *cw expr]
cond-expr       = concat-expr [*cw "?" *cw expr *cw ":" *cw expr]
concat-expr     = lor-expr *(*cw "@" *cw lor-expr)
lor-expr        = land-expr *(*cw "||" *cw land-expr)
land-expr       = bor-expr *(*cw "&&" *cw bor-expr)
bor-expr        = xor-expr *(*cw "|" *cw xor-expr)
xor-expr        = band-expr *(*cw "^" *cw band-expr)
band-expr       = eq-expr *(*cw "&" *cw eq-expr)
eq-expr         = rel-expr *(*cw ("=="/"!=") *cw rel-expr)
rel-expr        = shift-expr *(*cw ("<="/">="/"<"/">") *cw shft-expr)
shift-expr      = add-expr *(*cw ("<<"/">>") *cw add-expr)
add-expr        = mul-expr *(*cw ("+"/"-") *cw mul-expr)
```

[1] See the Structured Programming with `goto` Statements article by Donald E. Knuth (1974) for technical discussion.

DRAFT

```
mul-expr       = unary-expr *(*cw ("*"/"/"/"%") *cw unary-expr)
prefix-expr    = *(unary-operator *cw) postfix-expr
postfix-expr   = primary-expr *(*cw postfix)
postfix        = "++" / "--" / index / select
index          = "[" *cw expr [*cw ":" *cw expr] *cw "]"
select         = "." *cw field
primary-expr   = pexpr/constant-expr/method-expr/array-expr/ident
constant-expr  = int-constant/char-constant/"null"/"true"/"false"
method-expr    = method *cw "(" *cw [expr *(*cw "," *cw expr)] *cw ")"
array-expr     = "{" *cw [expr *(*cw "," *cw expr)] *cw "}"
```

An *expression* is a C-like expression. Some expressions are surrounded by parentheses, but this is to avoid ambiguity or to override operator precedence. An *assignment expression* allows assignment of right-hand side expressions to left-hand side variables. A *conditional expression* tests some expression, and has a true expression after `"?"` and a false expression after `":"`. We consider, generally speaking, *binary expressions* and *unary expressions*. The binary expressions consists of a binary operator and left and right operand expressions, the unary expressions of a unary operator and an operand expression. *Index expressions* and *slice expressions* represent array access and *select expressions* represent field access.

See also Table 4.1 for operator precedence and associativity.[2] The arity of an operators determines the number of operands. Operators with a higher precedence are parsed as the operands of operators with a lower precedence. For example, `1+2*3` is parsed as `1+(2*3)`, where the expression `2*3` is an operand of the addition operator, since multiplication has a higher precedence than addition.

Conditional operators are parsed specially. The true expression of a conditional expression is parsed as if surrounded by parentheses. Otherwise, it is right-to-left associative. For example, `x?y?z:w:w` is parsed as `x?(y?z:w):w`, and `x?y:z?w:w` is parsed as `x?y:(z?w:w)` and *not* as `(x?y:z)?w:w`.

Constant expressions, *method expressions*, *array construction expressions* and *variable expressions* are primary expressions. Method expressions are parsed specially. A method consists of an identifier *m* followed by parentheses in which zero or more comma-separated argument expressions occur. Array construction expressions also consists of zero or more comma-separated element expressions, but surrounded by accolades instead of parentheses. Array constructions are used to give variable number of method arguments.

Harmless ambiguities

The statement `if (x) {}` is ambiguous, as it can be parsed either as an if statement with variable `x` as test expression and an empty block statement as true body, or with an expression statement as true body that is an array construction expression without any elements. A parser prefers the first alternative, but the second alternative is harmless as the array construction expression is side-effect free and has the same behavior as an empty block statement.

Other ambiguous language constructs may be present but currently unknown.

[2]Compare with `https://en.wikipedia.org/wiki/Operators_in_C_and_C++#Operator_precedence`

DRAFT

Operator	Name	Fixity	Arity	Level	Associativity
(_)	parentheses	circumfix	unary	1	n/a
_++	increment	postfix	unary	2	n/a
_--	decrement	postfix	unary	2	n/a
_[_]	array indexing	suffix	binary	2	left-to-right
_._	field access	infix	binary	2	left-to-right
++_	increment	prefix	unary	3	n/a
--_	decrement	prefix	unary	3	n/a
+_	positive	prefix	unary	3	n/a
-_	negative	prefix	unary	3	n/a
!_	logical not	prefix	unary	3	n/a
~_	bitwise not	prefix	unary	3	n/a
_*_	multiplication	infix	binary	4	left-to-right
_/_	division	infix	binary	4	left-to-right
_%_	remainder	infix	binary	4	left-to-right
_+_	addition	infix	binary	5	left-to-right
_-_	subtraction	infix	binary	5	left-to-right
_<<_	bitwise shift left	infix	binary	6	left-to-right
_>>_	bitwise shift right	infix	binary	6	left-to-right
_<=_	less or equal	infix	binary	7	left-to-right
_<_	less than	infix	binary	7	left-to-right
_>=_	greater or equal	infix	binary	7	left-to-right
_>_	greater than	infix	binary	7	left-to-right
_==_	equal	infix	binary	8	left-to-right
_!=_	not equal	infix	binary	8	left-to-right
_&_	bitwise and	infix	binary	9	left-to-right
_^_	bitwise xor	infix	binary	10	left-to-right
_|_	bitwise or	infix	binary	11	left-to-right
_&&_	logical and	infix	binary	12	left-to-right
_||_	logical or	infix	binary	13	left-to-right
_?_:_	conditional	mixfix	ternary	14	special
_=_	assignment	infix	binary	14	right-to-left
_*=_	multiplied "	infix	binary	14	right-to-left
_/=_	divided "	infix	binary	14	right-to-left
_%=_	remained "	infix	binary	14	right-to-left
_+=_	added "	infix	binary	14	right-to-left
_-=_	subtracted "	infix	binary	14	right-to-left
_<<=_	shifted left "	infix	binary	14	right-to-left
_>>=_	shifted right "	infix	binary	14	right-to-left
_&=_	bitwise and'd "	infix	binary	14	right-to-left
_^=_	bitwise xor'd "	infix	binary	14	right-to-left
_|=_	bitwise or'd "	infix	binary	14	right-to-left

Table 4.1: Operator arity, precedence, and associativity. Shown in ascending level. A lower level means a higher precedence and vice versa. The underscores are operands and can be complex expressions or simple expressions: a constant or variable.

DRAFT

4.2.2 Abstract Syntax Tree

During the parsing of input data, an abstract syntax tree is constructed. The resulting tree does not correspond 1-to-1 to the production rules of the grammar given in Section 4.2.1, and some information originally present is removed (such as parentheses and implicit precedence).

```
class Element {}
class SyntaxElement extends Element {
    InputPosition position;
}
class ExternalElement extends Element {
    Import external;
}
```

We shall describe the hierarchy of elements of the abstract syntax tree. Certain elements in the syntax tree are imported from an external source. Each element is either present from the original source at some position (see Section 4.1) or external.

```
interface Identifier {
    String getValue();
}
class ExternalIdentifier extends ExternalElement
implements Identifier {
    String value;
}
class SourceIdentifier extends SyntaxElement
implements Identifier {
    String value;
}
```

Identifiers are either externally provided or present in the source file. An externally provided identifier results from importing symbol declarations. If an identifier is present in the source file, it has a position through inheritance from syntactical element. This pattern appears for other elements too.

```
interface Type {
    byte TYPE_INPUT = 1;
    byte TYPE_OUTPUT = 2;
    byte TYPE_MESSAGE = 3;
    byte TYPE_BOOLEAN = 4;
    byte TYPE_BYTE = 5;
    byte TYPE_SHORT = 6;
    byte TYPE_INT = 7;
    byte TYPE_LONG = 8;
    byte TYPE_SYMBOLIC = -1;
    byte getSort();
}
interface SymbolicType extends Type {
    Identifier getIdentifier();
}
class BasicType implements Type {
```

DRAFT

```
    byte sort;
}
class SourceType extends SyntaxElement implements Type {
    byte sort;
}
class ExternalSymbolicType extends ExternalElement
        implements SymbolicType {
    ExternalIdentifier identifier;
}
class SourceSymbolicType extends SourceType
        implements SymbolicType {
    SourceIdentifier identifier;
}
```

There are ten types, the first nine are types with a corresponding keyword. The tenth type is symbolic and has the additional identifier attribute.

```
interface TypeAnnotation {
    Type getType();
    boolean isArray();
}
class BasicTypeAnnotation implements TypeAnnotation {
    Type type;
    boolean array;
}
class SourceTypeAnnotation extends SyntaxElement
        implements TypeAnnotation {
    SourceType type;
    boolean array;
}
```

In variable declarations, such as local variable declarations and formal parameter declarations, a type annotation is provided that specifies the type and whether the variable is an array.

```
abstract class Constant extends SyntaxElement {}
class NullConstant extends Constant {}
class TrueConstant extends Constant {}
class FalseConstant extends Constant {}
class CharacterConstant extends Constant {
    String value;
}
class IntegerConstant extends Constant {
    String value;
}
```

Constant expressions, as present in the source, are either a Boolean constant, a character constant, or an integer constant. The value attribute of the latter is the raw character string from the input file: its integer value has to be calculated later. Not all character strings are valid constants: for example, `093` would be accepted by the parser but is an invalid octal number.

DRAFT

```
abstract class Method extends SyntaxElement {}
class BuiltinMethod extends Method {
    final static byte METHOD_PUT = 0;
    final static byte METHOD_GET = 1;
    final static byte METHOD_FIRES = 2;
    final static byte METHOD_CREATE = 3;
    byte sort;
}
class SymbolicMethod extends Method {
    SourceIdentifier identifier;
    SymbolDeclaration declaration;
}
```

There are two kinds of methods as they appear in the source: built-ins and symbolic methods. The terminology of *method* refers to either a function or a component, depending on its context in which it is used. The declaration a symbolic method refers to is resolved after parsing and applying grammar rules.

```
class ProtocolDescription extends SyntaxElement {
    List<Pragma> pragmas;
    List<Import> imports;
    List<SymbolDefinition> symbolDefinitions;
    List<SymbolDeclaration> symbolDeclarations;
}
class Pragma extends SyntaxElement {
    String value;
}
class Import extends SyntaxElement {
    String value;
}
```

The root element of a protocol description consists of three attributes: a list of pragmatics, as they appear at the beginning of the file, a list of qualified imports, and a list of symbol definitions. The import value attribute is the qualified identifier. After resolving the imports, all symbol declarations are known, relative to which symbolic methods are resolved.

```
interface SymbolDeclaration {
    Identifier getIdentifier();
    List<TypeAnnotation> getParameterTypes();
}
interface ComponentDeclaration extends SymbolDeclaration {}
interface FunctionDeclaration extends SymbolDeclaration {
    TypeAnnotation getReturnTypeAnnotation();
}
abstract class ExternalSymbolDeclaration extends ExternalElement
implements SymbolDeclaration {
    ExternalIdentifier identifier;
    List<TypeAnnotation> parameterTypes;
}
class ExternalComponentDeclaration extends ExternalSymbolDeclaration
```

DRAFT

```
implements ComponentDeclaration {}
class ExternalFunctionDeclaration extends ExternalSymbolDeclaration
implements FunctionDeclaration {
    TypeAnnotation returnTypeAnnotation;
}
abstract class SymbolDefinition extends SyntaxElement
implements SymbolDeclaration, VariableScope {
    SourceIdentifier identifier;
    List<FormalParameter> formalParameters;
    BlockStatement block;
}
class FunctionDefinition extends SymbolDefinition
implements FunctionDeclaration {
    SourceTypeAnnotation returnTypeAnnotation;
}
abstract class ComponentDefinition extends SymbolDefinition
implements ComponentDeclaration {}
class CompositeDefinition extends ComponentDefinition {}
class PrimitiveDefinition extends ComponentDefinition {}
```

A symbol declaration consists of an identifier and a signature, being a list of type annotations of the parameters. A symbol declaration is either a component declaration or a function declaration. Function declarations moreover have a return type annotation. Symbol declarations are either imported from some external source, or are given in the source by a symbol definition. Symbol definitions consists of formal parameters, which includes the names of parameters. The signature of a symbol definition is obtained by stripping the formal parameter names, leaving only their types.

```
interface VariableScope {
    VariableScope getParent();
    List<? extends VariableDeclaration> getVariableDeclarations();
}
abstract class VariableDeclaration extends SyntaxElement {
}
class FormalParameter extends VariableDeclaration {
    SourceTypeAnnotation typeAnnotation;
    SourceIdentifier identifier;
}
class LocalVariableDeclaration extends VariableDeclaration {}
class ChannelDeclaration extends LocalVariableDeclaration {
    SourceIdentifier from_identifier;
    SourceIdentifier to_identifier;
}
class MemoryDeclaration extends LocalVariableDeclaration {
    SourceTypeAnnotation typeAnnotation;
    SourceIdentifier identifier;
    Expression initial;
}
```

A variable declaration consists of a type annotation and a name. Formal parameters and local variable declarations are variable declarations. There are two local variable declarations:

DRAFT

channel declarations, and memory declarations. The former consists of two identifiers, representing each channel end. The latter consists of an initialization expression that gives the memory its initial value. Moreover, we have the notion of variable scope: a symbol definition and a block statement introduce a new variable scope. A variable scope has an optional parent variable scope, and consists of a list of the variable declarations declared in that scope.

```
interface LabelScope {
    BlockStatement getParentBlock();
}
abstract class Statement extends SyntaxElement {}
class BlockStatement extends Statement
implements VariableScope, LabelScope {
    List<LocalVariableDeclaration> locals;
    List<Statement> statements;
    VariableScope parent;
    List<LabeledStatement> labels;
}
class SkipStatement extends Statement {}
class LabeledStatement extends Statement {
    SourceIdentifier label;
    Statement body;
}
class IfStatement extends Statement {
    Expression test;
    Statement trueBody;
    Statement falseBody;
}
class WhileStatement extends Statement {
    Expression test;
    Statement body;
}
class BreakStatement extends Statement {
    SourceIdentifier label;
    WhileStatement target;
}
class ContinueStatement extends Statement {
    SourceIdentifier label;
    WhileStatement target;
}
class SynchronousStatement extends Statement
implements VariableScope {
    List<FormalParameter> formalParameters;
    Statement body;
    BlockStatement parent;
}
class ReturnStatement extends Statement {
    Expression expression;
}
class AssertStatement extends Statement {
    Expression expression;
```

DRAFT

```
}
class GotoStatement extends Statement {
    SourceIdentifier label;
    LabeledStatement target;
}
class NewStatement extends Statement {
    MethodCallExpression expression;
}
class ExpressionStatement extends Statement {
    Expression expression;
}
```

A statement is either a block statement, a skip statement, a labeled statement, an if statement, a while statement, a break statement, a continue statement, a synchronous statement, a return statement, an assert statement, a goto statement, a new statement, or an expression statement.

A block statement consists of a list of local variable declarations and a list of statements. The parent variable scope in which the block statement is situated is either a symbol declaration, a synchronous statement or another block statement. The parent block of a block statement is nothing if the parent variable scope is a symbol declaration, just the block statement if that is its parent variable scope, or the parent variable scope of the synchronous statement if that is the blocks parent variable scope. Moreover, a block consists of a list of labeled statements that are directly nested under it. A block statement can have zero or more directly nested statements.

A skip statement can be used in place of an empty block statement.

An if statement consists of a test expression and a true body statement or a false body statement.

A while statement consists of a test expression and a body statement.

Break and continue statements have an optional label; by label resolution the while statement that they refer to is resolved.

A synchronous statement is a variable scope, as it may introduce zero or more formal choice parameters. A synchronous statement always has a block statement as its parent variable scope (see Section 4.3.1), not to be confused with the fact that synchronous statements are not necessarily directly nested under a block statement.

A return statement consists of an expression that describes the return value; and, an assert statement consists of an expression checked for truth.

A goto statement consists of a label, and the labeled statement it refers to is resolved by label resolution. In contrast to break and continue statements, goto statements may point to labeled statements that are in label scope.

A new statement consists of a method call expression; its corresponding symbol declaration is resolved by method resolution.

An expression statement consists of an expression.

```
abstract class Expression extends SyntaxElement {
}
enum AssignmentOperator {
    SET, MULTIPLIED, DIVIDED, REMAINED, ADDED, SUBTRACTED,
    SHIFTED_LEFT, SHIFTED_RIGHT, BITWISE_ANDED, BITWISE_XORED,
    BITWISE_ORED
}
class AssignmentExpression extends Expression {
```

```
    Expression leftExpression;
    AssignmentOperator operator;
    Expression rightExpression;
}
class ConditionalExpression extends Expression {
    Expression test;
    Expression trueExpression;
    Expression falseExpression;
}
enum BinaryOperation {
    CONCATENATE, LOGICAL_OR, LOGICAL_AND, BITWISE_OR, BITWISE_XOR,
    BITWISE_AND, EQUALITY, INEQUALITY, LESS_THAN, GREATER_THAN,
    LESS_THAN_EQUAL, GREATER_THAN_EQUAL, SHIFT_LEFT, SHIFT_RIGHT,
    ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER
}
class BinaryExpression extends Expression {
    BinaryOperation operation;
    Expression leftExpression;
    Expression rightExpression;
}
enum UnaryOperation {
    POSITIVE, NEGATIVE, BITWISE_NOT, LOGICAL_NOT, PRE_INCREMENT,
    PRE_DECREMENT, POST_INCREMENT, POST_DECREMENT
}
class UnaryExpression extends Expression {
    UnaryOperation operation;
    Expression expression;
}
class IndexingExpression extends Expression {
    Expression subject;
    Expression index;
}
class SlicingExpression extends Expression {
    Expression subject;
    Expression fromIndex;
    Expression toIndex;
}
class SelectExpression extends Expression {
    Expression subject;
    SourceIdentifier field;
}
class ConstantExpression extends Expression {
    Constant value;
}
class ArrayConstructExpression extends Expression {
    List<Expression> elements;
}
class MethodCallExpression extends Expression {
    Method method;
```

DRAFT

```
    List<Expression> arguments;
}
class VariableExpression extends Expression {
    SourceIdentifier identifier;
    VariableDeclaration declaration;
}
```

An expression is either an assignment expression, a conditional expression, a binary expression, a unary expression, an array indexing expression, an array slicing expression, a select expression, a constant, an array construction expression, a method call expression or a variable.

An assignment expression assigns the left-hand side expression the value described by the right-hand side expression. The assignment operator employed may be either setting or modifying. A set assignment operator indicates that the value of the right-hand side is stored to what the left-hand side expression describes, discarding the old value of that left-hand side expression. A modifying assignment operator moreover takes into account the old value of the left-hand side expression. A multiplied assignment takes the old value of the left-hand side and multiplies it by the value of the right-hand side; the result is stored to what the left-hand side expression describes. In a similar way, we have divided (stores the result of dividing the old value by the right value), remained (stores the remainder of dividing the old by the right value), added, subtracted, shifted left, shifted right, performing a bitwise AND operation, bitwise XOR operation, or bitwise OR operation.

A conditional expression consists of a test expression, and a true and false expression. The truth of the value described by the test expression determines whether the conditional expression evaluates the true or the false expression.

A binary expression consists of a binary operator and takes a left and a right expression. The value that a binary expression describes is determined by the value of its directly nested expressions. The binary operations are: array concatenation, logical and short-circuit OR, logical and short-circuit AND, bitwise OR, bitwise XOR, bitwise AND, equality or inequality, arithmetic comparison operations, bitwise shift operations, and the usual arithmetic operations.

A unary expression consists of a unary operator and an expression. The value of a unary expression is determined by the value of its directly nested expression. The unary operators are: numerical positive, numerical negative, bit flipping, logical negation, increment and decrement either giving back the old or the new value.

An array indexing expression takes a subject array and an index. It describes looking up the value of the subject array at the offset described by the index. An array slicing expression takes a subject array, and gives as result another array limited between the from index and the to index. The from index is inclusive, the to index is exclusive, meaning that the length of the resulting slice is $to - from$.

A select expression selects a field of its subject. Fields are identifiers. A good example is the lenght field of an array, indicating the number of elements of the array.

A constant expression describes the value that its constant represent.

An array construction expression takes a list of directly nested expressions and constructs an array out of it. A method call expression takes a method and a list of directly nested expressions as arguments. The method is either a built-in or a symbolic method. In the latter case, the corresponding symbol declaration is resolved by method resolution.

A variable expression consists of a variable identifier. As the result of variable resolution, the variable declaration corresponding to the variable is linked to the variable expression.

4.3 Grammar Rules

After lexical analysis an abstract syntax tree is constructed. However, the parsing process is not yet completed: we check a number of rules to determine the *well-formedness* of the protocol description. Moreover, we perform *symbol resolution* by processing the imported symbol declarations and resolving symbolic references. In a similar manner, but local to each definition, we also perform *variable resolution* for variable declarations. Finally, we perform *type checking*.

One can make use of a general recursor for traversing the abstract syntax tree. The recursor is specialized to implement each rule. As the recursor traverses the tree it maintains a state, based on which a decision can be made to explore the tree further, or to raise a syntax error.

4.3.1 Well-formedness

The first list of rules that we check is well-formedness of the protocol description. These rules are checked in the order as given here, so later rules can assume that the protocol description is well-formed according to the previous rules. Some checks are semantically motivated: the protocol description may have no clear meaning if these checks are not performed.

Nested synchronous statements

In the block of every composite definition or function definition, no synchronous statements may occur either directly or nested under other statements. Within a primitive definition, no synchronous statement occurs either directly or nested under any other synchronous statement. The example below demonstrates three violations of this rule:

```
composite main(in a, out b) {
    new other(a,b);
    synchronous skip; // illegal
}
int fun(int x) {
    synchronous skip; // illegal
    return x;
}
primitive other(in a, out b) {
    while (fun(1) > 1) {
        synchronous { // legal
            synchronous skip; // illegal
}   }   }
```

Invalid variable declarations

Node declarations must not occur within function definitions or primitive definitions. A formal parameter of a synchronous statement of input or output type is invalid. The example below demonstrates a violation:

```
primitive dir(in a, out b) {
    while (true) {
        channel x -> y; // illegal declaration
        synchronous skip;
}   }
```

DRAFT

Function return statement

Primitive and composite definitions do not have a return statement. The block statement of a function definition must return. A statement returns if it is a return statement, or if it is a goto statement, or if it is a block statement where its last statement returns, or if it is an if statement where both branches returns, or if it is a while or labeled statement where its statement returns. All other statements do not return. The following example demonstrates a complex function that does not return:

```
int myfun(int x) {
    if (x > 0) {
        while (x > 0) {
            x--;
            if (x == 0) skip; // illegal
            else return x;
        }
    } else {
        int y = 0;
        label:
        if (y >= 0) {
            goto label;
        } else {
            y = 5;
            return myfun(x + 1);
}   }   }
```

Valid occurrences of built-ins

Except for `create` are built-ins not allowed outside of synchronous blocks. This implies that these built-ins are not allowed inside composite or function definitions, since they do not have synchronous blocks.

```
primitive main(in a, out b) {
    int x = 0;
    msg y = create(0); // legal
    while (x < 10) {
        y = get(a); // illegal
        synchronous {
            y = get(a); // legal
}   }   }
```

Invalid assignment expressions

The left-hand side of an assignment expression must be an assignable expression. A variable expression is assignable. An indexing expression is assignable. A slicing expression is assignable. A field expression is assignable. All other expressions are not assignable.

Invalid indexing and slicing expressions

The subject of an indexing expression must be an indexable expression. The subject of a slicing expression must also be an indexable expression. A variable expression is indexable.

DRAFT

A concatenation expression is indexable. An array construction expression is indexable. A slicing expression is indexable. A select expression is indexable. A method call expression is indexable. A conditional expression is indexable if its true and false expressions are indexable. All other expressions are not indexable.

Invalid select expressions

The subject of a field selection expression must be a selectable expression. A variable expression is selectable. A concatenation expression is selectable. An array construction expression is selectable. A slicing expression is selectable. An indexing expression is selectable. A select expression is selectable. A method call expression is selectable. A conditional expression is selectable if its true and false expressions are selectable. All other expressions are not selectable.

4.3.2 Resolution

There are three kinds of resolution that link identifiers to declarations. The first kind of resulotion is method resolution that links method call expressions to corresponding symbol declarations: symbol definitions or imported external symbol declarations. The second kind of resolution is variable resolution that links variable expressions to corresponding variable declarations: formal parameters or local variable declarations. The third kind of resolution is label resolution that links identifiers of control flow statements to labeled statements.

Imports

A protocol description can import zero or more qualified identifiers. Each qualified identifier is either *registered*, or is a *standard import*. Below, we list standard imports and what symbol declarations are imported.

- `std.reo` — standard Reo connectors:
  component `sync(in, out)`
  component `syncdrain(in, in)`
  component `syncspout(out, out)`
  component `asyncdrain(in, in)`
  component `asyncspout(out, out)`
  component `merger(in[], out)`
  component `router(in, out[])`
  component `consensus(in[], out)`
  component `replicator(in, out[])`
  component `alternator(in[], out)`
  component `roundrobin(in, out[])`
  component `reonode(in[], out[])`
  component `fifo(in, out)`
  component `xfifo(in, out, msg)`
  component `nfifo(in, out, int)`
  component `ufifo(in, out)`
- `std.buf` — functions for manipulating message buffers in network byte-order:
  function `byte writeByte(msg, short, byte)`
  function `short writeShort(msg, short, short)`

DRAFT

function `int writeInt(msg, short, int)`
function `long writeLong(msg, short, long)`
function `byte readByte(msg, short)`
function `short readShort(msg, short)`
function `int readInt(msg, short)`
function `long readLong(msg, short)`

Method resolution

Occurrences of imports and symbol definitions result in a list of declarations per protocol description. Each method expressions that occurs anywhere in a protocol description must refer to a listed symbol declaration. Otherwise, it is unclear what the method refers to. The process of checking whether every method expression refers to a known declaration is called method resolution. It proceeds by listing external and source declarations of composite, primitive and function definitions. After listing all declarations, method expressions are linked to the external or source declarations they refer to.

Every symbol declaration consists of a signature. A signature consists of a list of type annotations, one for each corresponding parameter. A signature is either a component signature or a function signature. A function signature furthermore has a return type annotation.

Uniqueness of symbol declarations

Every listed declaration, being a source declaration resulting from a symbol definition or an external declaration from an import, must have a unique identifier.

Valid method call occurrences

Every method call expression must refer to a symbol which is declared. The result of resolving methods is that each such expression is linked to the corresponding symbol declaration. Within any symbol definition, no method call expression must refer to a component declaration, except for method call expressions that occur in a new statement. Moreover, new statements must only occur in composite definitions, and its method call expression must refer to a component declaration.

Variable resolution

In component and function definitions, formal parameters are declared. In block statements, local variables are declared. In synchronous statements, formal parameters are declared. Each variable expression must refer to a variable declaration or a formal parameter that is in *variable scope*. Otherwise, it is unclear what the variable refers to. The process of checking whether every variable refers to a variable declaration is called variable resolution. The first step in variable resolution is to link every scope to its parent scope. Component and function definitions have no parent scope. A variable is in scope whenever it is declared in any of its surrounding block statements, or occurs as a formal parameter of a surrounding synchronous statement or symbol definition. It proceeds by linking every variable occurrence to a variable declaration in one of its parent scopes.

Block statements can be nested to declare additional local variables. Only statements that occur within a block can refer to its declared local variables. Outside of the block, those variable declarations are inaccessible because they are out of scope. It is possible to have two sibling

block statements that declare the same local variable; these are different variable declarations. In other words, the scope of a block statement is restricted to its nested statements only.

Similar for synchronous statements: only statements that occur within a synchronous statement can refer to its formal parameters. Two sibling synchronous statements may declare the same formal parameter; these are different variable declarations.

Uniqueness of variable declarations

In every scope, every variable declaration must have a unique identifier. This implies that every formal parameter must be unique. Every local variable declaration in a block statement must be unique. Every formal parameter of a synchronous statement must be unique. Moreover, every variable declaration cannot overshadow identifiers already declared before by a formal parameter or a local variable.

```
composite main(in a, out a) { // illegal
    new lossysync(a, a);
}
composite main2(in a, out b) {
    channel a -> c; // illegal
    new lossysync(a, b);
}
primitive lossysync(in a, out b) {
    while (true) {
        synchronous (int a) { // illegal
            if (fires(a) && fires(b)) {
                msg x = get(a);
                put(b, x);
            } else if (fires(a)) {
                msg x = get(a);
            } else assert !fires(b);
}   }   }
```

In above example, formal parameters with the same identifier are illegal. Variable declarations that overshadow a variable already in scope are illegal. Sibling blocks that declare a variable with the same identifier, as seen in the two cases where both `msg x` is declared, is allowed. Although the identifier is the same, the two separate blocks declare a *different* variable.

Label resolution

Occurrences of labeled statements result in a list of labels per block statement. Similar to variable resolutions, labeled statements are bound to their *label scope*. The scope of the label of a labeled statement is its surrounding block statement. Each control flow disruption statement that occurs in a symbol definition must refer to a label that is in scope. The process of checking whether every disruptive statement refers to a label is called label resolution. We already assume that block statements are linked to a parent variable scope. Thus, for every statement, it is possible to find its surrounding block statements. We collect the labeled statements that occur within a block statement.

DRAFT

Uniqueness of labels

In every label scope, every label must have a unique identifier. It is possible for two sibling blocks to contain labeled statements with the same identifier, but these labels are considered different.

```
int main() {
    while (true) {
        dupl: skip;
    }
    dupl: goto dupl;
}
```

In above example the duplicated label is illegal.

Valid control flow disruption statements

A goto statement must refer to a label in scope. A break or continue statement has an optional label. If a label is provided, that label must be in scope and attached directly to a while statement. If no label is provided, there must be a surrounding while statement.

DRAFT

Chapter 5

Protocol State Representation

This chapter gives meaning to protocol descriptions, whose syntax is detailed in Chapter 4. Section 5.1 introduces the table model for executions in discrete, synchronous time steps. Section 5.2 provides an informal semantics of PDL, and relates it to an ideal definition of protocol execution as a set of infinite tables. Section 5.3 introduces semantics for the execution of a connector, relating it to that of protocols, and ultimately defining a correctness criterion to serve as the foundation for evaluating the correctness of connector implementations.

5.1 The Table Model

We introduce the table model for reasoning about stateful execution in PDL components and connectors. First and foremost, a table is a structure with a set of columns, each with a unique label, and a set of rows, labeled with a whole number. Every combination of column and row intersects at a table cell, which holds a value.

We say a table corresponds to the run through a primitive component if it wholly represents the primitive's values in an observation that can be explained by a trace through its control flow. The primitive's local variables (including port variables) are reflected in columns, and the rows reflect observations of these values in agreement with its definition, one synchronous round at a time at the end of a synchronous block (see Chapter 4). The semantics of PDL expressions, statements, etc., are provided more concretely in Section 5.2 to follow. One such table runs from its initial state in the first row, advancing downward through table rows one at a time. In a run whose primitive's control flow reaches the end of the component's definition, the table propagates the values of local variables and blocks all its ports (they have the no-message value) for synchronous rounds forevermore.

The table of a composite component is defined as the combination of the tables of its constituent components, with columns renamed as necessary to avoid unintentional overlap in variable or port names. As reflected in PDL, composite components are able to fuse a pair of its constituent components' ports (one input and one output port) by creating a channel between them. This fusion is reflected in the composed table if, for every row, the fused columns have equivalent values. Often, we refer to this as 'overlapping' the ports' columns, and imagine them as a single, new column. By construction, rows of constituent components are kept 'in sync' by being mapped by the same whole number. The restrictions of the control flow of all constituent primitives, complete with local variables, are retained in the composite table.

DRAFT

Conceptually, a protocol description denotes a set of *infinite tables*, each denoting a unique execution of the protocol component where there are infinite rows and infinite columns. In practice, we often reason about particular runs, or finite prefixes of an infinite table. For example, a practical definition of termination is that a given infinite table has only inactivity (silent port behavior) in all rows after a finite prefix of activity (non-silent port behavior).

5.2 Protocol Semantics

> It is not necessary for the semantics to determine an implementation, but it should provide criteria for showing that an implementation is correct. — Dana Scott

We shall use the framework of denotational semantics for giving a compositional semantics of protocol descriptions. The semantics of composite and primitive component definition differ: the former is declarative, the latter is imperative. The definition of the semantics of components is given last.

In giving this semantics, we shall assume there are no divergences: the programmer is assumed to be an expert to ensure that all function definitions and synchronous statements are terminating. This, obviously, simplifies the presentation of the semantics. In practice, one would likely count the resources such as space and time consumed, and halt after a certain threshold has been reached (out of memory, out of time).

5.2.1 Internal Language

We shall restrict ourselves to a subset of the protocol description language. This restricted subset is called the *internal language*. Conceptually, one can translate the full protocol description language into this restricted subset, without losing any of the intended meaning. In fact, the meaning of the full protocol description language is precisely the meaning of its image under this translation in the semantic definition that follows.

The purpose of the internal language is to allow for a simpler semantic definition than when working with the full language. The internal language has the following properties:

- Modified assignment statements, e.g. `x += 5`, are translated into assignment expressions, e.g. `x = x + 5`.
- Side-effectful expressions are assignments and calls to built-ins. All other expressions are side-effect free if they do not contain any nested side-effectful expression. In the internal language, there are two different expressions occurring in expression statements: assignment of a built-in and assignment of a side-effect free expression. Expression statements that involve only a built-in call or a side-effect free expression can be assigned to a dummy local variable.
- Statements such as if, while, return and assert have side-effect free expressions.
- During translation, local variable declarations are introduced where necessary, using fresh local variable identifiers that do not clash with other identifiers in scope. For example, the statement `x = y = 5;` is translated into two statements `y = 5; x = y`.
- Statements consisting of disruptive control flow statements are translated into a statement where no disruptive control flow statement occurs. That is, no `break`, `continue` or `goto` statements appear, and there are no labeled statements.

DRAFT

- The block statement directly under the symbol definition is the only one to have non-zero number of local variables declarations. All nested declarations are transformed by moving them up in scope, possibly renaming the identifiers if clashes would occur.

We shall, for now, leave out the details of how this translation is defined. The reader may consult existing literature on the topic for a more detailed discussion. More importantly, the translation of the full protocol description language into the internal language must preserve its well-typedness.

5.2.2 Denotation

Values of the Boolean domain are true and false. We assume standard Boolean operations. Just like in Java, the primitive types *byte*, *short*, *int*, and *long*, are signed integers of finite bit width of 8-bit, 16-bit, 32-bit, and 64-bit, respectively. Bitwise, logical and arithmetical operations on these domains are defined as usual.

Arrays of non-array values are values, i.e. there are no arrays of arrays. The domain of the elements of an array is uniformly fixed, i.e. we have byte arrays and int arrays, but not arrays of mixed bytes and integers. Every array has a fixed length, of the 32-bit signed integer domain. The domain of byte arrays is included in the domain of messages, i.e. every message can be treated as a byte array. Moreover, the domain of messages includes the *null* message, representing the absence of a message.

A frame is an assignment of formal parameters (of a function declaration or the component declaration that are not of input or output type) to values. Frames are treated in such way to remain constant for the duration of a function or a component, and in particular are not updated. Moreover, for each function definition, we assume there is a corresponding function, that maps frames to return values.

A *store* is a mapping from memory variables to values. As usual, stores can be updated to overwrite the value of a variable. An *oracle* is a mapping of port variables to a value of the message domain. We say that a port fires (with respect to an oracle or the head of an infinite sequence of oracles) iff its value is not the *null* message, i.e. a message is present.

For giving the semantics of a component, we assume we are given an infinite sequence of oracles. Conceptually, every time we synchronize an oracle is taken from the stream. This oracle is used to give the values of port variables, for the duration of synchronization. Before, in between, and after synchronization, no oracles are used and port variables are inaccessible (see Section 4.3.1).

During execution of a component, we record an input/output history. This history is a sequence of events. An event is either a *get* event, a *put* event, a *tick* event, or an *inconsistency* event. The purpose of keeping track of a history is to determine the consistency of an execution. A get event records a port and a message value that is received; a put event records a port and message value that is sent. A tick event records that the synchronous round finishes, and an inconsistency event records an impossible execution.

We consider the state of a component to be represented by a triplet of some store, an infinite sequence of oracles, and an input/output history. The store is variant and represents the *current values* of the local variables and parameters. The head of the infinite sequence of oracles represents the *current values* of the *ports*, i.e. the parameters of input or output type. The tail of the infinite sequence of oracles represent all *future* values of ports. Since time progresses one step after completing a synchronous statement, the infinite sequence of oracles is also variant. The input/output history tracks the *past* events.

DRAFT

We start at the lowest level of the abstract syntax tree and work upwards: first we give meaning to side-effect free expressions, secondly to statements, and thirdly to the primitive component definition.

Expressions

As usual, the meaning of an expression is obtained through evaluation. Evaluation of an expression takes a state. Evaluating an expression either results in a value, a divergence, or an inconsistency. Evaluation is defined inductively on the structure of expressions, as usual. For example, a variable expression denotes the value of that variable in the store. The built-in methods of *get*, *avail* and *ready*, however, denotes the value that the head oracle assigns to the port supplied as argument to these methods. Only the *put* built-in method can cause an inconsistency during evaluation, namely when the value assigned by the head oracle and the value denoted by its second argument expression are different.

Other expressions, such as binary expressions and unary expressions, are defined in the usual way. Without being too precise, we intend to mimic the semantics of C for evaluation of expressions. Method call expressions in primitive definitions always refer to function declarations; we evaluate a function call by creating a frame out of the values obtained from evaluating the arguments and applying it to the denotation of the function definition to obtain the resulting value or a divergence. Other than function calls, in no way can expressions in primitive definitions lead to divergence.

Statements

The meaning of a statement is a state transformer, viz. a mapping from state to state. The meaning of statements is defined inductively in a standard way. For example, sequential composition of two statements is function composition of the state transformers of the comprised statements. We make the following remarks:

- The meaning of a synchronous statement is defined as follows: take the state transformer of the nested statement, and take a state. We apply the state to the given state transformer to obtain another state. In the latter state, we replace the infinite stream of oracles by its tail. This ensures that after a synchronous statement completes, we progress to the next oracle. We further record in the input/output history a tick.
- A state is inconsistent if its history has a recorded inconsistency event. The meaning of statements on inconsistent states is undefined, i.e. any further next state is related to an inconsistent state.
- For an expression statement, there are two cases: an assignment of a side-effect free expression or an assignment of a built-in call. In the former, the side-effect free expression never cause an inconsistency, and the evaluated value of the expression is used to update the store. In the latter case, encountering a built-in call, we additionally record the call and its argument values in the input/output history. If the expression evaluates to an inconsistency, we record an inconsistency event (thus making the resulting state inconsistent).
- For an assert statement, if the evaluation of the test expression is false, an inconsistency event is generated.

DRAFT

We consider operational consistency and causal consistency of histories. Operational consistency is the lack of any inconsistency events. Causal consistency of a history means that messages being send and received correspond to each other, i.e. there is no received message that was never sent.

Operationally Consistent Executions

An execution of a primitive component starts with some initial state and proceeds by following the state transformer that the main block statement induces. During this process, the infinite sequence of oracles is consumed, and a history of input/output events generated. The resulting execution is operationally consistent if (1) the history of input/output events does not contain an inconsistency event, (2) in each round where an input port fires according to the oracle a corresponding get event is recorded, and (3) in each round where an output port fires, a corresponding put event is recorded.

In other words, if an input port fires, there must be at least one get operation in the same synchronous round. If an output port fires, there must be at least one put operation in the same synchronous round. And as a consequence of composition, when two primitive components share a port, such that one has access to the input side and the other has access to the output side, if the port fires then at least one put and at least one get operation must be recorded in the input/output history of the composed component.

Causally Consistent Executions

Causal dependency is a relation defined between component states. Within the execution of a component, every state transition is caused by a statement. We informally consider the causal dependency relation, by looking at how statements are causally related. A statement x is causally related to another statement y in the same primitive if x has a value which y observes. For example, `x=5;y=x;` where $y = x$ is causally related to $x = 5$. A statement containing a `get` expression of the value of a port P is causally related to all put statements of port P occurring within any synchronous blocks occurring in the same synchronous round. Note that this also introduces causal relations between primitives. The causal relation is transitively closed. Executions are causally consistent if for each synchronous round the dependency graph is acyclic. This captures the intuitive causal 'flow' of data from putters to getters; we prohibit protocol descriptions where the value of a message is caused by itself. Preserving causal consistency has a desirable consequence: any given message can be traced to definitive origins, which must be either (1) being put into the protocol at a boundary port, or (2) a message valuation provided by an oracle.

5.2.3 Protocol Behavior

The ideal meaning of a protocol is a set of infinitely long tables. This set characterizes the behavior of a protocol. For each protocol description, we interpret it as describing such a set. The meaning of a protocol description is the set of infinite tables induced by consistent executions of its main component. A consistent execution of a component is an execution that is both operationally consistent and causally consistent. An infinite sequence of oracles can be treated as an infinite table, where each oracle provides the valuation of the ports of the table. A set of consistent executions induces a set of infinite tables as follows: take for each consistent execution the infinite sequence of oracles and collect them in the induced set of infinite tables.

DRAFT

A protocol is consistent if it consists of at least one infinite table, i.e., there exists at least one consistent execution.

5.3 Connector Semantics

This section defines the semantics of connectors. Intuitively, connectors instantiate a particular communication session with respect to a configured protocol. As such, the semantics of protocols and connectors are tightly related.

5.3.1 Setup Phase

In Chapters 2 and 3, it was explained that connectors are configured with a protocol description, that describes a set of executions. Ports of connectors are then bound, by either giving applications access to native ports, or passively or actively bind them over the Internet. At connection time, a set of peers is gathered, until the connector is fully connected.

During connection time connectors exchange configured protocol descriptions, composing the protocol descriptions. In the end, the connector consists of a single, shared protocol description that is composed out of the protocol descriptions submitted locally by each application. After the connector is fully connected, no further peers can join. Thus the meaning of the shared protocol description becomes fixed, and the communication phase can begin.

There is an important distinction between no protocol behavior, and silent protocol behavior. A connector has no protocol behavior if always eventually runs into an inconsistent execution. An inconsistent execution is not continued; there does not exists a next synchronous round after encountering an inconsistency. However, silent protocol behavior indicates that ports are always silent: further synchronous rounds can be constructed but every port blocks. Idealistically, silent protocol behavior consists of a single infinite table where all ports block, whereas no protocol behavior is an empty set.

5.3.2 Communication Phase

A connector runs through connector states while collecting constraints from native applications. A run is a sequence of connector states alternated by a constraint. The constraint is provided by the participating applications. A connector state consists of a finite table it has constructed so far.

The only permissible runs of a connector are those where its table is a prefix of one infinite table in the behavior of its associated protocol. In other words, a run consists of a finite table that must be a prefix of some infinite table specified by the composed protocol description. The infinite table is an ideal object, and the run is its finite approximation. As a connector runs through an execution, it maintains a state and records its past observations reflecting the run. Sometimes a choice can be made in which execution to follow through: two executions that are allowed by the protocol might share a common prefix with the current run of the connector.

Connectors are also restricted by native applications: its choices must not only be consistent with some behavior of the protocol, it must also be consistent with the constraints put forward by the native applications. Native applications submit constraints using the Application Programming Interface, which has an effect on the run. In Chapter 3 we further explained how the application does this, and how it is related to data flowing in and out of the connector.

An application's *mode of operation* with its connector defines how the connector organizes the application's submitted batches into synchronous rounds.

DRAFT

1. In *Cooperative Mode*, the connector guarantees a one-to-one correspondence between an application's `sync` invocations and synchronous rounds. The connector cannot proceed to the next synchronous round without the cooperation of the application.

2. In *Preemptive Mode*, the connector imposes *delay insensitivity* on the application; the application's constraints per `sync` are applied to synchronous rounds in the same order as in cooperative mode, but the connector is permitted to inject rounds in-between in which it presumes the application blocks all its ports. This mode can be viewed as an optimization, allowing the connector to progress its state if the application is slow. However, the application loses the ability to reason about the current round the connector is in.

Unless otherwise specified, we presume applications to be in cooperative mode with their connector. This is motivated by it being 'conservative', with fewer possible executions for the given behavior. Observe that an application in cooperative mode is able to simulate one in preemptive mode by spawning a thread which will eagerly invoke `sync` with empty batches, i.e. a batch where all native ports block.

5.3.3 Correctness

We distinguish two notions of connector correctness: total correctness and partial correctness. Total correctness of a connector means that the finite table of the last state of every run of a connector must be a prefix of a consistent execution of its corresponding protocol. Partial correctness of a connector means that the finite table of the last state of every run must be a consistent prefix of an execution of its corresponding protocol.

DRAFT

DRAFT

Chapter 6

Reference Implementation

This section describes the design and implementation details of the *reference implementation* of the connector semantics for multi-party communication between user applications atop IP. The part the implementation has to play in the Reowolf project is described in Section 6.1. With this in mind, Section 6.2 gives an overview of the design of the implementation, making more concrete the relationship between the connector semantics defined in Section 5.2 and their concrete manifestation in the implemented runtime. Section 6.2 follows, elaborating on some interesting particulars of the implementation, including the description of identified design alternatives for connectors, as well as elaborations on some implementation details.

6.1 Goal

The reference implementation aims to provide the minimum viable product, providing connectors as a facility atop IP for participants to exchange data with their peers in discrete synchronous rounds in accordance with the protocol definitions with which their connectors were configured. For now, it is paramount to provide functionality such that protocols are sufficiently expressive and that the runtime is correct. The implementation is designed to lay the groundwork for development later in the project, such that the runtime becomes better optimized and is able to support more safety mechanisms such as deviation detection. Concretely, the reference implementation has the following properties which are not intended to persist throughout its development:

1. **Protocols are only interpreted locally**
   Protocol descriptions supplied in a connector's configuration are not interpreted by peers in the distributed system. Indeed, the existence of these descriptions for facilitating optimization is an important part of the project. However, the reference implementation makes no such attempt, illustrating that even in the absence of any protocol description exchange, the system is functional.

2. **Protocols are interpreted just-in-time**
   Protocol descriptions supplied during configuration are parsed and linked into decorated abstract syntax trees. However, the runtime makes no attempt to interpret or optimize the behavior of components eagerly. The runtime's interpretation of a component's definition is contemporary with the synchronous round in which the effects will be observable. Pro-

tocol descriptions are available for inspection and prepossessing during the setup phase, but the reference implementation demonstrates that this is not necessary for functionality.

3. **Transmission control is delegated to TCP**
   The algorithms behind the reference implementation are designed with discretized message passing in mind, suitable for (1) The IP layer on which it is built, (2) Reowolf's port message-passing semantics, and (3) the various distributed algorithms on which the system relies. For the sake of simplicity, the reference implementation realizes network channels a pairs of connected TCP sockets, delegating various necessary tasks of transmission control including (1) re-transmission on packet loss, (2) backpressure and flow control, (3) message integrity. TCP is in fact an over-approximation of the implementation's requirements for network channels; for example, the implementation does not require the preservation of message ordering. Later implementations might build the needed transport layer for network channels atop UDP, or even just IP.

6.2 Implementation Overview

This section is intended to bridge the gap between the semantics of connectors and the particular design decisions made to facilitate a concrete implementation.

6.2.1 Protocol Components as Actors

The *actor model* is characterized by describing complex system as the concurrent execution of a set of *actors*, each of which given a sequential task. As with PDL, the actor model uses these isolated units to encapsulate accesses of a program's persistent state, and allows controlled information exchange between encapsulated units via explicit message passing. Information crosses the boundaries between actors in only two ways: (1) an actor creates a new actor, (2) one actor sends a message, which is asynchronously received by another. Applications of the actor model vary in many ways, including the specifics of how actors route their messages. Often, actors are able to send messages to any other actor whose *identifier* they know.

The reference implementation realizes stateful, executable protocol descriptions as networks of interlinked state machines, comparable to actors. Each actor defines a sequential control flow, and must create new actors to express concurrency. Actors that create other actors corresponds with *composite* protocol components, whose behavior is the composition of other components.

Unlike the typical actor model, actors cannot send and receive messages at any time. Communication is only possible in a synchronous *round*, a discrete phase within its control flow expressed in PDL as a synchronous statement block. For the duration of a synchronous round, actors are able to send or receive at most one message per port. All message exchange is synchronous, so both send and receive only succeed if the peer on the other end of the port participates. If only one of the two actors sharing a port attempt to exchange data, the system becomes *inconsistent* and the system exits, citing the cause of the protocol deviation. To avoid this, actors are permitted to reflect on the *oracle*, a structure available for the duration of the round which promises to know whether a given port fires. In this fashion, actors are able to *attempt* data exchange and distinguish success from failure. Actors are able to introduce inconsistencies as a way of constraining the provided oracle. In this fashion, the actor can express desirable outcomes as a function on which ports exchanged which messages during the round.

DRAFT

6.2.2 Distributed Oracle Search

Recall how Section 6.2.1 explains that a component's actor is able to rely on an oracle for each synchronous round to be provided such that no inconsistencies can be observed. Let such an oracle *satisfy* the component. For every synchronous round, the task of the runtime is to identify an oracle such that it satisfies all components. When done correctly, the system runs with the states of all components' actors advancing one round at a time, all coordinating as if by magic on the sending and receiving of messages which satisfy the constraints of all participating components. In this fashion, the oracle provides an unambiguous *result* of the synchronous round such that all component's information exchange is coordinated, arranged into linearized rounds, and guaranteed not to contradict the constraints of any components.

For particular protocols in particular configurations at runtime, there may not exist a satisfactory oracle. By design, the system promises to satisfy all components, and creating an unsolvable satisfaction problem is as simple as just one component asserting a contradiction: *assert*(*false*). The reference implementation parameterizes each synchronous round with a timeout, promising to identify a satisfactory oracle if it exists, or time out. Unsatisfiable configurations result in a timeout. Later in the project, the intent is to facilitate preprocessing of protocol descriptions to identify unsatisfiable configurations before they are encountered, or to identify the cause of an unsatisfiable constraint.

6.2.3 Speculative Component Execution

Section 6.2.2 explained that, each round, the distributed system comprising the connector runtime searches for an oracle that satisfies all components' actors. Many approaches to this kind of search problem can be understood as being decomposed into distinct phases:

1. enumerating a set of candidate oracles, and
2. filtering out candidate oracles which do not satisfy all components.

There exists a naïve algorithm for the search. It is simple to reason about, and retains a clean separation between the two phases. Consider that it suffices to enumerate all conceivable combinations of port variables, for all combinations of message exchanged during the synchronous round. It is easy to see how the resulting candidate oracle set must necessarily contain a solution if it exists, as it will contain all conceivable oracles. Candidates can be inspected and filtered out in phase 2 until one is found to satisfy all components, whereupon the search can stop. If all candidate solutions are filtered out, there is no satisfactory oracle and the system can respond by raising an exception or whatever is appropriate. Recall from Chapter 4 that only *primitive* components are permitted to participate in synchronous message exchange. As such, this section concentrates on primitive components and their actors only.

While correct, the naïve approach is prohibitively expensive, and ignores a lot of available information. To proceed, we make the search process more intelligent by avoiding regions of the search space for which any candidate solutions are certain to be filtered out. In so doing, the distinction between the two phases of the search algorithm become blurred; candidates are effectively filtered by avoiding them during enumeration.

Firstly, observe that it suffices to constrain the set of port variables to union of all ports of components in the connector in the synchronous round. In essence, we make the set of port variables finite, and exclude those that belonged to components that have *exited*, as there will be no component able to access those port variables.

DRAFT

Next, the generation of candidates is performed *per component*, such that the set of candidates can be determined as a function of the states of components at the start of the synchronous round. Alone, this is not a fruitful optimization. The benefits of this approach necessitate another change: candidate oracles are represented *abstractly*, with large sets of candidate solutions represented by terse oracle *predicates*. The intuition is to enumerate the constraints each component imposes on a satisfactory candidate, and identify an oracle that satisfies the constraints. In practice, the multiplicative increase in the number of candidates as a result of searching per component is overshadowed by the multiplicative decrease resulting from enumerating predicates rather than oracles. To build a component's predicate, the runtime simulates the control flow of the component's actor through the synchronous block, collecting constraints in response to the actor reflecting on port variables. As a simple example, consider a component with a synchronous block containing: `assert(fires(port_a));`. The set of candidate solutions for this components is large, containing those that assign to `port_a` the messages `0`, `1`, and so on. The predicate is able to represent this set more tersely, encoding something that can be understood as $a \neq *$, where `*` represents the special *silent* value, encoding the absence of a message. It is easy to see how having access to the actor's state is able to significantly reduce the set of candidate solutions in a manner that is not possible otherwise. Consider how the synchronous block `assert(msg < port_a);` expresses a very meaningful constraint given the knowledge that `msg == 2` in the actor's state.

Finally, we avoid the need for predicates to represent logical disjunction, instead generating a set of predicates per component. The utility of this is that, per predicate, the satisfiability of the component becomes a boolean property, and we are able to distinguish true from false implicitly by simply discarding predicates representing unsatisfiable disjuncts. In practice, it suffices to create the set of predicates incrementally, *forking* an initial predicate whenever a disjunction is introduced. The most obvious example arises in components with conditional branching. For example: `if(firing(port_a)) assert(firing(port_b));`. Rather than having to formulate a predicate over `port_b` in terms of `port_a`, we recognize the disjunction from the statement's structure. Two predicates are created, $a \neq * \wedge b \neq *$ in the event the if-condition evaluates to true, and $a = *$ for the other. In this fashion, each component's description is explored recursively, creating a small search tree rooted at the start of the synchronous block. Branches are pruned if they are found to necessitate contradictory predicates or diverge. Branches that reach the end of the synchronous block are retained, each predicate expressing the constraints the branch imposes on the oracle; oracles that satisfy these constraints satisfy the component. Note that there may be many such branches that reach the end of the synchronous block. Observe that, by construction, their predicates have empty intersections in the oracle space. For this reason, a given oracle satisfies *at most* one branch. We refer to this branching search of actor simulation as *speculative execution*, as each branch represents the runtime speculating on the real execution of the component. A sensible optimization presents itself: the resulting actor states of all satisfied branches are retained; once the oracle is determined, the corresponding branch *commits*, replacing the state of the actor with its own. In effect, the disjunctive branches per components are similar to *transactions*, and the runtime ultimately lets at most one commit.

In summary, the reference implementation forgoes enumerating and filtering candidates for the connector at large. Instead, a set of branch-and-predicate pairs are computed for each of the connector's primitive components. The branches enumerate disjunct subsets of the candidate oracle space, representing the set of oracles that satisfy the component. To proceed, it is necessary to discover an oracle such that it occurs in such a set of *all* components. The reference implementation distributes the components of the connector, and so what remains is a problem of distributed consensus, explained in Section 6.2.5.

DRAFT

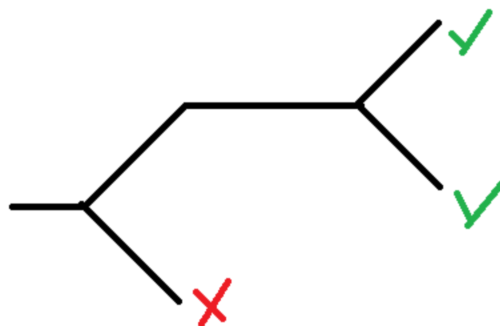

Figure 6.1: Todo

6.2.4 Speculative Message Passing

The speculative execution procedure described in Section 6.2.3 has two shortcomings, both solvable by one modification to the speculative execution procedure:

1. Without information about the contents of received messages, oracle predicates must remain abstract, representing large sets of candidate solutions.

2. By computing oracle predicates per component individually, the task of determining which candidate oracles are *causally consistent* requires an additional filtering phase.

Both problems are the result of reasoning about components in isolation, resulting in unwanted generality of each component's predicates. The first problem results from the missed opportunity of communicating the constraints a *sender* applies on the message to its recipient. Such reasoning is conceivable, as the topology of ports and connectors is fixed during the synchronous round.

The second problem results from neglecting to reason about the causal relationships between messages, necessitating that it be reconstructed. Recall that the intention of the causal consistency property defined in Section 5.2 intends to enforce a unique origin for messages, prohibiting that messages be causally dependent on themselves. To motivate its necessity, consider the example of two components that synchronously send and receive the same message; concretely: `put(port_a, get(port_b));` and `put(port_b, get(port_a));`. Without a check for causal consistency, predicates for both components may (understandably) conclude that any message exchanged at ports a and b are acceptable whenever $a = b$, and the system may reach consensus on an oracle assigning, say, $a = 3, b = 3$. The message flowing through these ports has no original sender, and thus can only be explained when reasoning 'backwards'. The resulting valuation is sufficiently unconstrained that the chosen oracle is almost entirely subject to the whims of the runtime's implementation. As it is impossible to reason about causality from only the *result* of speculative execution (i.e., pairs of predicates and actor states), either the speculative execution procedure must be changed, or an additional filtering phase will have to inspect the actors' components' definitions a second time.

The solution of the reference implementation is to perform the speculative execution procedure for all components concurrently, and using channels to send *speculative messages* through

DRAFT

ports; information about one component's predicate can be transmitted through a port. This necessitates that speculative execution is able to *block* and *resume* in response to the availability of speculative messages. Recall that speculative branches of a component carry a predicate over the candidate oracles that satisfy the component. As the nature of this predicate is a function of the branching control flow of the component in particular, it is non-trivial for one component to make sense of the predicate information of another component. To solve this, speculative messages are sent by a component's particular *branch*, sent to another component and received by the *set* of branches for which the predicate is consistent with its own. As an example, consider a connector with port *a* and *b* both allowing component *x* to send messages to component *y*. The former has as part of its synchronous block: `if(foo  fires(port_a)) put(port_b, m1);`, and the latter has `m2 = get(port_b);`. The runtime crafts *x*'s speculative message through *b* such that the expression within its if-block is represented. Without realizing it, in receiving the message, *y*'s branch has inherited the constraints on *x*'s branch. This change enforces that only causally consistent branches reach the end of the synchronous block. To demonstrate, consider the case of the connector whose components induce a causal loop: `put(port_a, get(port_b));` and `put(port_b, get(port_a));`. Here, both speculative executions will have single branches, both of which block in wait for a message from their peer. The cyclic dependency is made manifest in the runtime itself, and both branches *starve*. Section 6.3.3 explains how the starvation of speculative branches do not impede the liveliness of the runtime itself.

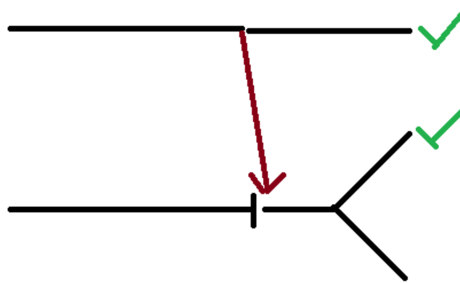

Figure 6.2: Todo

6.2.5 Oracle Consensus

The speculative execution procedure described in Section 6.2.3, and elaborated to guarantee causal consistency in Section 6.2.4 generates a set of predicates per primitive component, such that any satisfactory oracle must satisfy exactly one predicate per component. What remains is to identify and *decide* on such an oracle, such that the entire distributed system is aware of the decision unambiguously and can end the synchronous round with the decision in effect. Let this oracle be called the *decision*. Assuming all predicates of all components are available at once, this decision process is a matter of identifying a combination of component predicates such that (1) each component contributes one predicate, and (2) the decision satisfies all of its constituent predicates. If done correctly, the resulting decision would (1) correspond with one branch per component, making unambiguously clear in which state the component's actor is at the end of the synchronous round, and (2) characterize a synchronous round in which all components

observe the same port variable valuations without observing any inconsistencies. Observe that in some cases there are multiple oracles which satisfy all components (each distinct oracle with a different combination of components' branches). Such cases arise when the outcome of the synchronous round results in a nondeterministic choice between the available options. This nondeterminism is fundamental to component compositionality, as it allows components to allow choices to be made by other components without necessarily knowing their constraints. By design, the protocol does not constrain which outcome becomes the decision.

We introduce a consensus procedure, allowing the connector's set of primitive components to exchange control information with their peers until one has the necessary information and authority to decide, and announce its decision to its peers. To this end, we introduce the *consensus tree*, an overlay network built atop the existing network between components during the setup phase. As the network of components connected by channels is an arbitrary connected graph, the consensus tree is constructed by omitting some channels. To be clear, these channels still exist, and are necessary for the speculative message passing described in Section 6.2.4. The consensus procedure of the reference implementation takes the definition of a satisfactory oracle (an oracle which satisfies one branch per component) and applies it recursively for all subtrees of the consensus tree. Effectively, every component solves a sub-problem: identify a predicate for which satisfies one branch each for all components in the subtree of which it is the root. The base case occurs for leaf components, those with zero children. Observe that their sub-problem is trivial: all branches yielded by their local speculative execution procedure satisfy all components in their consensus subtree (a singleton set of components). All components continue to find solutions for their subproblem, forwarding the solution to their parent if they exist, and *deciding* otherwise. In this fashion, only the root of the subtree has the capability to decide, and it does so the first chance it gets. The decision event is followed by the chosen decision predicate being announced to all components, flooding through the consensus tree from parent to child. Necessarily, each component has exactly one branch satisfied by the announced predicate, and is able to *commit* that branch's state, ending the synchronous round.

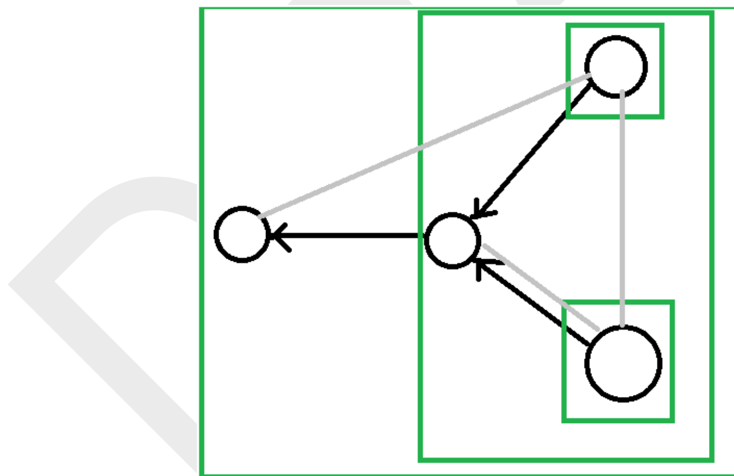

Figure 6.3: Todo

6.2.6 Ternary Domain Port Predicates

Recall from the previous sections that control messages of various kinds are transmitted between connectors carrying oracle *predicates*, structures which communicate abstract constraints on an oracle's port variables. In Section 6.2.4, predicates are used to propagate causal dependency and constraints on the oracle between two components. In Section 6.2.5, predicates are used again to represent the aggregation of components' predicates from children to parents in the consensus tree.

Conceptually, all components are provided access to the oracle, and are able to reflect on its valuations so long as they are able to *express* the reflection; a component cannot query the value of a port it does not have. This restriction is made obvious in the syntax of PDL by exposing reflection on the oracle as built-in procedures `isfiring`, `put` and `get` all parameterized by a port variable in the local scope. Components are able to reflect on the precise value of *received* messages, i.e., at ports for which the component is a 'getter'. Concretely, components may use procedures `fires` to reflect on whether a message is received, and then `get` to retrieve the message value. While they are provided an oracle in the same manner, putters can only reflect on *whether* a port exchanges a message using `isfiring` (this can be thought of as synchronously checking if their peer is receiving). Putters cannot reflect on the precise message, and can only *prescribe* the value by providing a message-type expression into the `put` statement. This is in place for the reasons mentioned in Section 6.2.4; the ability to reflect on a message before it is sent can only introduce causal cycles, and so it is syntactically prohibited. Consequently, speculative messages transmit concrete *contents* (the value of the message-type expression as it appears to the actors).

This restriction permits a significant optimization: all oracle predicates need to only represent the messages exchanged at ports using the ternary value domain: $\{\textit{some}, \textit{none}, \textit{unspecified}\}$, or equivalently, a partial mapping from port variables to the binary domain $\{\textit{some}, \textit{none}\}$. To understand why, observe that it is impossible for two distinct speculative messages to be sent in the same round, through the same port, but carrying different contents. For such a case to arise, it would be necessary for one branch to have the same oracle predicate in the same state, but yet send two different messages. The guarantee that the component's control flow is deterministic given an oracle forbids this. As a consequence, although a predicate does not represent the domain of port variables, it is still uniquely encodes it.

The end result is a vast simplification in the representation of predicates, allowing for a terse representation, reducing the size of control messaging overhead, and reducing the cost of the many operations comparing, modifying and composing them. Further optimizations of the encoding of predicates is desribed in Section 6.4.5.

6.3 Implementation Specifics

This section expands on the design for the reference implementation overview in Section 6.2, expanding on procedures that are important to the functioning of the implementation, but not vital to the overall concept.

6.3.1 Setup procedure

The procedures outlined in Section 6.2 took for granted that the distributed system had been connected and set up appropriately to afford communication. In this section, we expand on the setup procedure. Chapter 3 introduces explains that applications instantiate their handle on a

DRAFT

connector at runtime, (1) configuring it with a protocol description, (2) annotating the ports of the protocol's main component with *port bindings*, and finally (3) connecting the connector before communication can begin.

Distributed Node Identifiers

In Section 6.3.1 to follow, the need for an election procedure during the setup phase is explained. In his book on distributed algorithms [Fok13], Wan Fokkink succinctly summarizes the impossibility of Las Vegas algorithms for election or consensus in anonymous networks. In other words, there can be no algorithm that always finishes correctly for election or consensus in networks in which nodes are not provided unique identifiers. The proof revolves around the notion of *symmetry*; in networks in which nodes are initialized with identical algorithms, nodes must find a way to distinguish themselves to achieve inherently assymetric goals, e.g. election and consensus.

Although difficult in general, in practice there are many robust ways of generating unique identifiers. For example, a centralized authority may distribute identifiers; IP and MAC addresses are examples of this which already exist, and may be sufficient in many cases. If all else fails, one may rely on nodes to *guess* their identifier from a large space, effectively relying on the uniqueness of the choice of their random number generator of choice. This work assumes the existence of unique node identifiers, relying on the generation of a 64-bit number by the operating system at startup. The topic of identifier selection is scheduled for exploration later in the project.

Port Identifiers

Section 6.2 described several procedures on which the runtime system relies to facilitate synchronous communication. Therein, it was taken for granted that when one component sends a message with information about its state, the requirements on its oracle, states of ports and so on, the information is in a format comprehensible to the recipient. Ensuring this property is non trivial, as there is not initially a globally-unique identifier for all port variables. While components in PDL name their variables, this is a strictly orthogonal naming scheme, local to the component's scope. What is required is an identification scheme that maps ports to identifiers and vice versa such that two identifiers are the same if and only if they refer to the same logical ports. The two directions of this implication can be understood separately as (1) there cannot be 'collisions', i.e., one identifier used to refer to multiple ports, and (2) there cannot be 'aliases', i.e., multiple identifiers for the same ports.

Many schemes exist for allocating the names of ports such that property 1 is preserved. Section **??** describes various schemes and how they trade off algorithmic complexity during the setup and communication phases. Owing to its robustness and flexibility, the reference implementation employs a scheme whereby port identifier are derived from node identifiers. Concretely, port identifiers are tuples of (N, I), where N is the identifier of some node in the distributed system, and I is an *index* which the node identified by N guarantees to only allocate to one port variable. Avoiding collisions is far easier in this manner, as each node must only guarantee that the indices it allocates locally do not collide. Uniqueness of port identifiers follows from the uniqueness of node identifiers. The preservation of property 2 follow naturally with this approach. Observe that only two components at a time refer to any given port directly, precisely the components that can exchange messages through the port. These two components must come to a consensus on the naming of the channel they have in common. This procedure is easy in practice, as the symmetry between the components must often be broken anyway. The

reference implementation has two simple methods for identifying a port when creating its underlying channel: (a) for network channels, the underlying TCP connection requires the distinction between *passive* peer (the peer that accepts the incoming connection), and the *active* peer (the peer that initiates the connection to a known IP address); the convention is for the passive peer to identify the port, and communicate the identifier to the active peer by means of a control message. (b) for channels created to link two components in one node's shared memory, there is no symmetry to break, and the node chooses and provides the identifier to both components.

Section **??** to follow explains alternative port identification schemes that make different tradeoffs. The chosen scheme was used for the reference implementation owing to its simplicity and robustness.

Consensus Tree Construction

Once all nodes are identified and the network channels are establishes and their ports itentified, the distributed system is running, but it is not yet ready for communication. What remains is the construction of the *consensus tree*, the overlay network taken for granted in Section 6.2.5 for the communication phase. Initially, all nodes know only their own identifiers, and a set of port-ends that connect them to their peers. The sink tree is constructed in two phases (1) decentralized leader election, and (2) centralized tree construction.

The first phase relies on the *echo algorithm with extinction* [Fok13]. In a nutshell, all nodes initiate a wave and become a leader once it completes, returning to the initiator. All these waves occur concurrently, with messages of different waves distinguished by being tagged with the identifier of its initiator. Whenever a node encounters a message whose initiator has an identifier larger than that of their current wave, they abandon the current wave and join the new one. Here, it suffices to say that only the wave of the node with the largest ID completes.

The second phase begins, initiated by the elected leader. What follows is the construction of a sink tree rooted at the leader; nodes choose their own parent, and therefore must rely on messages from their neighbors to determine whether the edge connecting them to the neighbor is in the sink tree at all (if so, that neighbor is their child). The leader initiates a wave of *announcement* messages to all its neighbors. Non-leaders forward the announcement to all neighbors but the one from whom they first receive it, to whom they send a distinct *acknowledgement* message instead, which communicates to the recipient that the sender considers the recipient their parent. Ultimately, every node receives exactly one message from every neighbor. Network edges included in the consensus tree carry an announcement from parent to child in one direction, and an acknowledgement in the other. Edges not included in the sink tree are characterized by an announcement being sent in both directions. Once every node has received its messages, the algorithm terminates and the tree is complete.

6.3.2 Component Controllers

The reference implementation introduces the notion of a *controller*, an agent in the connector whose role is to manage a set of components. Effectively, the set of controllers partitions the connector's set of components. A controller is characterized by *not* distributing its state, such that its internal decisions do not require distributed consensus.

The introduction of controllers allows the granularity of the protocol's component logic to be decoupled from the granularities of (1) parallelism, and (2) the nodes in the consensus tree. A smaller number of controllers simplifies several of the runtime's control algorithms, primarily those for constructing and using the consensus tree. The algorithms described thusfar do not change significantly with the introduction of controllers. Speculative execution and

speculative message passing does not change at all. Oracle consensus changes only in that, where previously each node had one ‘local’ component, at most one parent component and a set of children components, now each node has a *set* of local components, at most one parent controller, and a set of children controllers; the property of each node of the consensus tree representing a subtree containing a set of components remains unchanged.

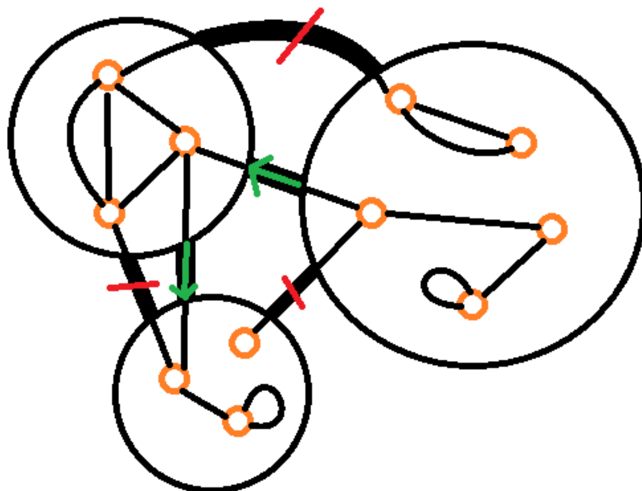

Figure 6.4: Todo

The introduction of controllers lays the groundwork for flexibility and optimization later. Consider, for example, the benefits of being able to alter the consensus tree without otherwise altering the behavior of the connector. For example, a smaller number of larger connectors has the advantages of (1) reducing the probability of identifier collisions, or allowing the identifier space to be reduced safely, and (2) reducing the number of needed concurrent threads for machines with limited resources, and (3) reducing the control messaging incurred by the consensus procedure. On the other hand, a larger number of smaller connectors exposes opportunities for increased parallelism.

For now, the reference implementation uses one controller per connector handle, and it manages the application’s native component, as well as the components defined by the application’s protocol configuration. This has the benefit of requiring no additional threads whatsoever; the work of each synchronous round is driven by the application thread itself.

6.3.3 Cooperative Actor Scheduling

Section 6.2.3 describes the speculative execution of each component as largely isolated (and therefore, concurrent) procedures. With the introduction of controllers in Section 6.3.2, it becomes necessary for a single entity to multiplex the speculative execution of a set of such components without imposing an ordering on their execution. To this end, the reference implementation leverages a custom cooperative scheduler.

DRAFT

Beginning the Synchronous Round

The order in which a controller performs the part of its components' actors' speculative execution prior to the synchronous block is irrelevant, as these actors can only act upon their own internal states and cannot exchange messages. As such, the controller starts the synchronous round by performing the speculative execution of each component's actor until it enters the synchronous block or exits. In this time, it is also possible for new actors and channels to be created. If any inconsistency is encountered here, the entire connector becomes inconsistent, as without access to any oracle, there can be no branching in the speculative execution (and thus, until entering their synchronous blocks, the execution of actors is not 'speculative' at all). As components can be neither created nor destroyed within the synchronous round, the set of the components that settles at the beginning of the round remains fixed until the end.

Cooperative Actor Scheduling

6.3.4 sec:cooperative$_a$*ctor*$_s$*cheduling*

It is beneficial to imagine the work of the runtime during a synchronous round and the reciprocation between two systems, characterized by their role in the system:

1. **The Controller** multiplexes the speculative execution of a set of component actors, routes incoming messages to the appropriate actor and reasons about the distributed state of the system such that it can reach consensus on a satisfactory oracle to end the round.
2. **The Component Actor** is able to inspect and manipulate its local store, and interact with its environment by (1) assigning and inspecting an oracle's port valuations, (2) interact with the runtime by signaling the discovery of an inconsistency, or request the creation of components or channels.

The reference implementation preserves the separation of these concerns by representing the work of an actor as a *coroutine*, logical units of work which are cooperatively scheduled with the runtime of the controller at large, resulting in a control flow that appears to bounce back and forth between the controller itself and various actors. Execution of an actor alternates between the controller's and actor's views, minimizing the overlap between the concerns. Actors are permitted to run in a context, given a structure to facilitate callbacks to the controller such that less intrusive operations may be performed on-the-fly by way of a callback. This includes the creation of new channels and components, which do not require the actor to halt its execution. In other cases, it is necessary for the actor to halt, yielding control back to the controller such that it may perform more intrusive managements of actors at large. For example, when an actor wishes to inspect the contents of a received message, the actor must *block*, cooperatively yielding its scheduling quantum for to the controller until the message is available and the actor is scheduled once again.

The cooperation between actors and controllers is necessary for the speculative execution described in Section 6.2.3; events requiring the introduction of speculation on oracle valuations often requires an actor's state to be replicated and subsequently distinguished, resulting in the exploration of separate speculative branches. The controller relies on actors to report the discovery of inconsistencies, resulting in those speculative branches being pruned. In this way, the system responsible for an actor's execution is entirely oblivious of coordination tasks such as consensus, predicates and speculation, while the controller system is oblivious of the internals of the evaluation and state of each actor.

DRAFT

Controller Event Loop

Recall that a controller sees to it that all of its components' actors enter their synchronous blocks together. This is necessary for the correct initialization of the consensus procedure described in Section 6.2.5, as it is impossible to reason about solutions until the set of participating components is established. Before handling the receipt of any messages at all, actors are scheduled to run until a decision is found, or only actors *blocking* for the reception of a message remain. Subsequently, an invariant holds: no progress can be made without the receipt of a message from the network. Concretely, this may be because either (1) a decision can be made, but the consensus procedure is halted until a control message arrives from some controller (see Section 6.2.5), or (2) some actor has not finished its speculative execution, as it awaits a speculative message from another actor (see Section 6.2.4).

For the remainder of the synchronous round, the controller maintains this invariant by responding to incoming messages. The round ends either with a timeout, or when the consensus procedure yields a decision.

6.3.5 Meta Protocol Messages

Here, we make explicit the message domain used by the reference implementation. These correspond exactly to enumeration types used to represent them in memory. Messages are expressed as contiguous structures, designed to be parsed unambiguously by greedily matching one token at a time from left to right, corresponding with their in-memory representations of tagged unions. In transmission over the network, an extra step is required at either end to serialize and de-serialize the structures into and from byte streams respectively. Below, the grammar for recognizing a serialized message is provided in ABNF. The production for $var-int$ is omitted; it refers to a 64-bit unsigned integer encoded in variable-length integer encoding[1]. This encoding is used to compress numbers expected to usually have a long null-byte prefix if encoded using a fixed size. Other distributions of integer values within their bounded domain are better represented using a more conventional fixed length. For these values, (e.g., controller-id), network byte ordering is

```
message        = setup / communication
setup          = (port-setup / echo / elected / ack-parent)
communication  = round-index (payload / elaborate / decided)

port-setup     = %x00 port-info
echo           = %x01 controller-id
elected        = %x02 controller-id
ack-parent     = %x03
payload        = %x04 predicate payload
elaborate      = %x05 predicate
decided        = %x06 predicate

port-info      = port-id ("P" / "G")
predicate      = length *assignment
payload        = length *byte
port-id        = controller-id id-suffix
assignment     = port-id ("T" / "F")
```

[1] https://developers.google.com/protocol-buffers/docs/encoding

```
controller-id  = 8 byte
round-index    = u64-var-int
length         = u64-var-int
id-suffix      = u64-var-int
byte           = %x00-FF
```

6.4 Opportunties for Exploration

6.4.1 Actor Reconfiguration

The reference implementation makes very little use of the protocol descriptions available at runtime. In a sense, the protocol descriptions are interpreted verbatim, without regard for optimization of the descriptions given to actors in response to the descriptions of *other* actors. While they don't change the protocol's behavior, these alterations may change the behavior of individual actors, and thus they can be thought of as instances of systemic reconfiguration.

Speculative Branch Pruning

Speculative branches which a component may describe in isolation may be identified as unreachable when considered in the context of the actor's environment in a particular connector. As a trivial example, consider the composition of components which (1) may not may not consume an incoming message every round, with one that (2) never sends any messages. An obvious optimization opportunity exists to replace the former actor with one that does not consider the possiblity of receiving messages at all.

This optimization results in a reduction in speculative message passing, and in the branching of the speculative execution of actors.

Composition Flattening

In many cases, components may express the composition of components resulting in a set of actors whose work could be easily represented by only one. There are many reasonable explanations for these apparently overcomplicated protocols being used in practice: (1) the subcomponents were specified independently and composed only at runtime, perhaps by different users on different machines, (2) a composite component allowed for better PDL code-reuse either for clarity or terseness, (3) the existance of the simplification is not apparent to the humans or machines that submitted the protocol descriptions, as it may emerge from subtle combinations. Often, the resulting 'flattened' primitive component results in a more efficient connector at runtime. For example, consider a set of identical components which continuously forward a single message from one port to another, composed into a long chain. Clearly, the composition of these components is in some sense idempotent, and can be easily replaced with a single component with the same description.

6.4.2 Custom Transport Protocol

The reference implementation relies on properties of its communication channels for its algorithms for speculative execution, election, consensus and so on. Concretely, channels are required to preserve *integrity*, ensuring that messages are received with the same contents with which they were sent, and *reliability*, ensuring that every message sent is eventually received.

DRAFT

Although they are not distinguished in PDL, the reference implemention requires two distinct kinds of channel depending on the circumstance: (1) **Memory channels** connect components and controllers on the same machine. (2) **Network channels** connect components and controllers over the network, and they are implemented atop TCP. Memory channels provide both guarantees trivially; the system relies on the coherence and integrity of the machine's memory and nothing else. Network channels rely on TCP's error detection and timeout-based re-transmission features to preserve the required properties.

Both kinds of channel provide a guarantee on the *ordering* of messages sent per logical channel. The reference implementation is built not to depend on this property such that it may be relaxed later. While the preservation of ordering comes cheaply for memory channels, the same is not true for network channels, which must work to re-establish the ordering that arises from the asynchronous network medium.

In future versions, the intention is to implement a transport protocol which better suits the requirements of the reference implementation, relaxing the unnecessary properties to reduce overhead. For example, the reference implementation benefits from being able to conditionally relax the requirement of re-transmission in response to the progress in the consensus procedure. Concretely, it suffices to preserve reliability for all messages in a synchronous round until a decision is reached, whereupon all of the round's speculative messages may be discarded (as they are no longer needed). Furthermore, given information about the state of the consensus procedure, the transport layer may tailor the *frequency* of re-transmissions per outgoing message. In this fashion, the reference implementation is able to implement a more informed form of flow control, guaranteeing eventual delivery, but biasing its efforts of re-transmitting messages that are more likely to be needed to reach the decision.

6.4.3 Component Precompilation

The reference implementation has a lazy approach to interpreting the user-provided protocol specification. Later in the project, in combination with various efforts to preprocess actors to alter their behavior at a logical level, it is the intention to also investigate *compilation* of the protocol description to a form better-suited for efficient execution rather than human readability. This step is also the ideal opportunity for applying various optimization strategies well-explored for other compiled languages such as C.

6.4.4 Shared Memory Value Aliasing

As described abstractly in Section 6.2.3, and more concretely in Section 6.3.3, the runtime must circumstantially replicate the states of executing actors to explore different speculative branches of the actor's execution space. The cost of this replication is very implementation-specific, and is expected to change in response to the representation of actors in their state and behavior. However, in all cases it can be expected that components will frequently store *messages*, byte sequences of arbitrary length transmitted through ports. It is expected that there will be many cases in practice where messages stored by actors are replicated more often than they are mutated. For this reason, a promising optimization is to use the *copy-on-write* for these larger structures, allowing for the resources underlying these values to be transparently aliased between actor replicas until the moment the resources themselves must be replicated such that the replicas can diverge. By performing the replication lazily, it is expected that fewer replications of messages will be necessary overall. In combination with more specialized implementations of allocators for these kinds of aliased data types, it can be expected that future versions of the

implementation will be able to use the system's resources more efficiently without altering any observable behavior.

6.4.5 Dense Predicate Encoding

Predicates over oracles are the primary means by which controllers and actors aggregate and compare information at runtime. As such, predicates are represented in most control messages; they are sent, received, serialized and deserialized, and compared frequently. Section 6.2.6 explained that, per port variable, only three possible values are possible, and thus must be represented in predicates. This section explores futher means by which the representations of predicates may be altered to optimize these frequent operations.

Contiguous Port Indices

Recall that in Section 6.3.1, the concrete representation of port identifiers was described as a tuple of controller identifier, and port index. With this approach, controllers are free to provide identifiers to a set of ports in isolation, as the BITVECTORS. Cid -> (Cidx -> bool)

Hierarchal Controller Identifiers

childen and themselves. The sink starts with a trivial prefix. Eg: sink given prefix [] and has two children a,b gives prefix [00] to self gives prefix [01] to a gives prefix [10] to b (possibly saves [11] for future use) —– a given prefix [01] and has one child c gives prefix [010] to self gives prefix [011] to c. Worst case bits is linear with controllers Best case bits is log2 with controllers

Contiguous Port Identifiers

flatten predicates entirely. needs centralized though!

6.4.6 Hierarchal Speculative Branch Structure

MORE EFFICIENT QUERYING for FEEDING MESSAGES

6.4.7 Localized Synchrony

for some protocols, it is not necessary for synchronous blocks to advance in lockstep. EXAMPLE. we can apply a powerful optimization: components only synchonize with those reachable by firing ports? still lots of questions to answer

6.4.8 Arbitrary Oracle Variables

nondet branches. dynamically allocated port variables

DRAFT

6.5 Example Connectors

the idea is to (1) clarify how the system can be used to give the reader confidence that we are the real deal

TODO describe (1) a use case, (2) the set of main applications, (3) the PDL.

1. firewall: filter messages 2. leader election in a ring problem in 1 synchronous round VS async 2. 2-matching "tinder" problem for arbitrary

Bibliography

[Fok13] Wan Fokkink. *Distributed algorithms: an intuitive approach*. MIT Press, 2013.

DRAFT