Vrije Universiteit Amsterdam

Master's thesis

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in
Parallel and Distributed Computer Systems.

Generating Safe and Fast Coordination code in Rust with Reo

Christopher Esterhuyse
(ID: 2553295)

supervisors

Vrije Universiteit Amsterdam	**Centrum Wiskunde & Informatica**
dr. J. Endrullis	dr. F. Arbab

July 16, 2019

Abstract

TODO

Contents

List of Figures

List of Listings

List of Tables

Part I

Preliminaries

Chapter 1

Introduction

these days useful abstractions are everywhere. communications are still relatively primitive: they are solved in a case-by-case basis but there isn't a general approach. the problem is that machines work with actions, and we think in interactions. there are many approaches to representing one with the other. for example, session types allow you to represent actions, and then allow you to predict your PARTNER's actions.

Reo is neat in that its exogenous. it puts the coordination in one place so you extract the protocol from your code. this makes it easier to collect and reason about. you express your protocol in a high-level language and then use that as a specification. its explicitness makes it very useful for humans, but also for machines; the reo compiler is a tool for generating coordination glue code. the intuition is you abstract away the structure of your network and you no longer distinguish endpoints etc. you instead use PORTS as your interface. the compiler generates the details for you and then it behaves according to the protocol at runtime .

the reo compiler has support for numerous backends such as java. there are incentives for adding support for systems languages such as C: they represent a large swathe of the possible user space, and their low-levelness means that they can more effectively leverage the information that protocol descriptions provide in the first place.

Rust is a programming language related to C++, intended for a similar audience. aside from the comforts of modern programming languages (closures, generics, functional patterns, extensive macros) it is notable for its unique memory management system: it relies on affine types to statically manage variable bindings, implicitly freeing memory which goes out of scope in a predictible manner. its ownership rules also prevent the majority of data races and protect the programmer from undefined behavior such as accessing uninitialiedd memroy. its UNSAFE sub-language is very similar to C, and can be tapped-into explicitly to achieve optimizations that the compiler cannot prove are safe. Rust is also useful for its exceptionally expressive APIs, as the types themselves allow and require the OWNERSHIP of values to be specified.

-in this work we detail the development of a Rust back-end for the Reo compiler to generate protocol objects which can act as the communication mediums of 'compute components'. Chapter BLAH deals with the translation process itself. Chapter blah describes how the generated rust code performs the role of a coordinator at runtime, detailing significant optimizations, particularly focusing on those that take advantage of components co-existing in shared memory. Chapter blah investigates the development of additional tooling for automatically detecting deviations from protocols at compile-time using no extra compilation steps. Chapter BLAH investigates the runtime characteristics of these systems at runtime. Part BLAH reflects on the progress of the project and suggesting directions for future work.

Chapter 2

Background

2.1 Reo

Reo is a high-level language for specifying protocols. Here, we explore the motivation behind Reo's development, how the language is used, and (at high level) how it works. The Reo language has applicability whenever there is a benefit in being able to formalize a communication protocol. However, this work primarily focuses on Reo's role in automatic generation of glue-code for applications.

2.1.1 Motivation

TODO focus on safety properties

Modern software development involves the construction of large and complex projects. Owing to their scale and the heterogeneity of the tasks required, many people are involved in the development of a program at once, and over its development lifetime. The industry has long-since established paradigms for managing the scale of these projects. One tenet of good software design is *modularity*, which describes a structure that, instead of being designed monolithically, is built out of smaller constituent modules. In addition to isolating modules such that they can be re-used in other projects, this design philosophy allows contributors to concentrate on a subset of all the modules at a time. These ideas are well-established in practice; code re-use and separation-of-concerns have been prevalent for some time.

Reo's utility is not only its ability to facilitate modularity. Reo is designed such that properties of the individual modules are *preserved* when modules are *composed* into larger ones. This preservation marks the difference between *gluing* modules together (and hoping for the best), and *composing* them into something guaranteed to have the intended properties.

2.1.2 Language

Reo is a *coordination* language. This describes its focus on the specification of the *interactions* between distinct actors. This is in contrast to the usual *action-centric* model common to languages with their roots in sequential programming, where the programs or specifications describe *actions* of entities, relegating any associated interactions to requiring derivation from the actions. In a nutshell, Reo provides a language for describing the behavior of a *system* of actors by explicitly constraining the behavior of the *connector* which serves as their communication medium.

(TODO define connector. same as component just maybe structural?)

The Reo language is essentially graphic; each connector defines a relation over named *locations*. Complex connectors are defined as the composition of simpler connectors over its locations. This inherently visual language is also often seen in its textual form, usually in the context of machine parsing.

The simplest *primitive* connectors cannot be subdivided

by listing a set of constituent connectors. The simplest primitive Reo connectors are *channels*(TODO channel vs primitive). Channels by definition cannot be subdivided into constituent connectors, as they are defined by either (a) the model that provides Reo's semantics, or (b) opaque components defined in some target language such as C or Java. The nodes themselves are the other important aspect of the language. Ultimately, each node corresponds to a (logical) location which may hold up to one datum at a time. Reo is by default *synchronous*, and relationships between locations propagates that synchrony. *Locations* are divided into two classes according to whether

(TODO)

This motivates the Reo's metaphor of propagation of data and back-propagation of data-constraints, corresponding to its namesake, the Greek word for 'flow'. The compositional aspect is meaningful when locations are involved in multiple relationships.

forwards (by moving data several 'hops' at once) and backwards

In addition to re-using nodes inside a connector, connectors are able to expose these nodes for re-use in the connectors *above them* by exposing the node in the connector's *interface*. These exposed nodes are called *ports*, leaning on the metaphor of the connector *moving* some data in and out of itself.

(data flow corresponds with what happens at runtime, except its SYNCHRONOUS by default. Relate to TDS. Talk about replication and equality checks).

(In the context of applications, components that cannot be composed compile to things managed by different threads. at the boundaries, they communicate with ports. Here, there is a meaningful difference between putter and getter. Include example of how a protocol that uses some port A three times still results in a putter-getter pair)

2.1.3 Typical Channels

In principle, Reo does not enforce the use of any primitive channels in particular. Users are free to use channels that are best-suited to their use case. In practice, a small set of exceptionally simple channels are favoured in literature and in practice owing to their versatility. As such, this work presumes that these consititute the majority of the channels out of which our protocols are composed. Below, we enumerate this set of channels and their behavior.

1. sync(I_0, O_0)
2. fifo1(I_0, O_0)
3. lossy_sync(I_0, O_0)
4. exclusive_router(I_0, O_0, O_1)

2.1.4 Semantic Models

Reo took a number of years to take its present shape. It is recognizable as early as 2001, but was presented as a concept before it was formalized, leaving it as a task for future work [JA12]. Later, This several different approaches to formal semantics were developed. For our purposes, it suffices to concentrate only on the small subset of the semantics to follow. For additional information, the work of Jongmans in particular serves as a good entry point[JA12].

Starting with the fundamentals, a **stream** specifies the value of a variable from data domain D changing over the course of a sequence of events. Usually streams are considered infinite, and so it is practical to define them as a function $\mathbb{N} \mapsto D$. A **timed data stream** (TDS) takes this notion a step further, annotating each event in the sequence with an increasing *time stamp*. A TDS is defined by some tuple $(\mathbb{N} \mapsto \mathbb{R}, \mathbb{N} \mapsto D)$, or equivalently, $\mathbb{N} \mapsto (\mathbb{R}, D)$ with the added constraints that time must increase

$\mathbb{R}$	A	B
0.0	0	*
0.1	*	0
0.2	*	*
0.3	1	*
0.4	*	1

Table 2.1: Trace table comprised of TDS's for variables A and B. This trace represents behavior that adheres to the *fifo1* protocol with input and output ports A and B respectively.

toward infinity[ABRS04]. By associating one TDS with each *named variable* of a program, one can represent a *trace* of its execution. TDS events with the same time stamp are considered simultaneous, allowing reasoning about *snapshots* of the program's state over its run time. These traces can be practically visualized as **trace tables**, with variables for columns and time stamps for rows by representing the absence of data observations using a special 'silent' symbol *, referring *silent behavior*. In this work, we use 'trace tables' to refer to both the visualization and to a program trace as a set of named TDS's. The runs of finite programs can be simulated either by bounding the tables (constraining the TDS domain to be finite), or by simulating finite behavior as infinite by extending the 'end' forevermore with silent behavior. Table 2.1 gives an example of a trace table for some program with two named variables.

One of it's earlier *coalegebraic models* represented Reo connectors as **stream constraints** (SC) over such TDS tables in which variables are ports [Arb04]. Here, constraints are usually defined in first-order *temporal logic*, which allows the discrimination of streams according to their values both now and arbitrarily far into the future[1]. This model is well-suited for translating from the kinds of safety properties that are typically desired in practice. Statements such as 'A never receives a message before B has communicated with C' have clear mappings to temporal logic, as often it is intuitive to reason about safety by reasoning about future events. Table 2.1 above shows the trace of a program that adheres the *fifo1* protocol with ports A and B as input and output respectively.

SC are unwieldy in the context of code generation. In reality, it is easier

[1] Not all variants of temporal logic are equally (succinctly) expressive. It requires a notion of 'bounded lookahead' to express a notion such as 'P holds for the next 3 states' as something like $\Box^{1-3}P$ rather than the verbose $(\Box P \land \Box\Box P \land \Box\Box\Box P)$.

to predicate one's next actions as a function of the *past* rather than the future. Accordingly, **constraint automata** (CA) was one of the *operational models* for modeling Reo connectors that has a clearer correspondence to stateful computation. Where an NFA accepts finite strings, a CA accepts trace tables. Thus, each CA represents some protocol. Programs are adherent to the protocol if and only if it always generates only accepted trace tables. From an implementation perspective, CA can be thought to enumerate precisely the actions which are allowed at ports given the correct states, and prohibiting everything else by default. A CA is defined with a state set and initial state as usual, but each transition is given *constraints* that prevent their firing unless satisfied; each transition has both (a) the *synchronization constraint*, the set of ports which perform actions, and (b) a *data constraint* predicate over the values of ports in the firing set at the 'current' time step. For example, Listing 2.1 above is accepted by the CA of the *fifo1* connector with all ports of binary data type $\{0, 1\}$. Observe that here the automaton discriminates the previously-buffered value ('remembering' what A stored) by distinguishing the options with states q_{f0} and q_{f1}. As a consequence, it is not possible to represent a *fifo1* protocol for an infinite data domain without requiring infinite states.

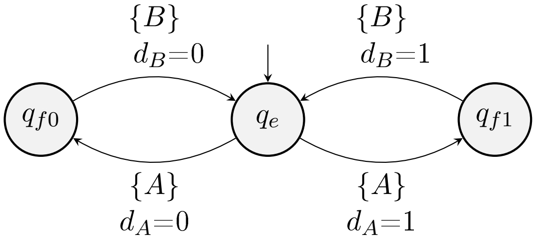

Figure 2.1: CA for the *fifo1* protocol with ports A and B sharing data domain $\{0, 1\}$.

Later, CA were extended to include *memory cells* (or *memory variables*) which act as value stores whose contents *persist* into the future. Data constraints are provided the ability to assign to their *next* value, typically using syntax from temporal logic (eg: m' is the value of m at the next time stamp). Figure 2.2 revisits the *fifo1* protocol from before. With this extension, the task of persistently storing A's value into the buffer can be relegated to m, simplifying the state space significantly. This change also makes it possible to represent connectors for arbitrary data domains, finite or otherwise.

For the purposes of Reo, we are interested in being able to compute the

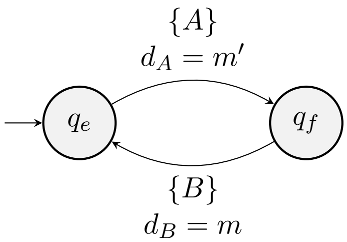

Figure 2.2: CA with memory cell m for Reo connector $fifo1$ with arbitrary data domain D common to ports A and B. Two states are used to track to enforce alternation between filling and emptying m.

composition of CAs to acquire a model for the compositions of their protocols. Figure 2.3 shows an example of such a composition, producing *fifo2* by composing *fifo1* with itself. This new protocol indeed exhibits the desired behavior; the memory cells are able to store up to two elements at a time, and B is guaranteed to consume values in the order that A produced them. Even at this small scale, we see how the composition of such CA have a tendency to result in an *explosion* if state- and transition-space. When seen at larger scales, a *fifoN* buffer consists of 2^N states. The problem is the inability for a CA to perform any meaningful *abstraction*; here, it manifests as the automaton having to express its transition system in undesired specificity. Intuitively, the contents of m_0 are irrelevant when m_1 is drained by B, but the CA requires two transitions to cover the possible cases in which this action is available. In the context of accepting existing trace tables, data constraints are evaluated predictably. However, in the case of code generation we are able to treat the data constraint instead as a pair of (a) the *guard* which enables the transition as a function of the *present* time stamp, and (b) the *assignment*, which may reason about the next time step, and which we are able to guarantee by *assigning* variables. As such, data constraints are broken up into these parts where possible. Figure 2.3 and others to follow formulate their data constraints such that the guard and assignment parts are identifiable wherever it is practical to do so.

Evidently, memory cells provide a new means of enforcing how data persists over time. In many cases, it can be seen that the same connectors can be represented differently by moving this responsibility between state- and data-domains. **Rule-based automata** (RBA) are the cases of CA for which this idea is taken to an extreme by relying only on memory cells entirely; RBAs have only one state. Figure 2.4 models the *fifo1* connec-

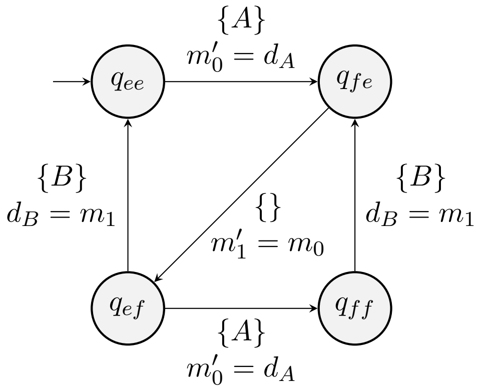

Figure 2.3: CA with memory cells m_0 and m_1 for the *fifo2* connector with an arbitrary data domain for ports A and B. Transitions are spread over the state space such that the automaton's structure results in the *first-in-first-out* behavior of the memory cells in series.

tor once again, this time as an RBA. Aside from the added expressivity, RBAs benefit from being cheaper to compose. As the state space is degenerate, RBAs may be easily re-interpreted into forms more easy to work with. **Rule-based form** (RBF) embraces the statelessness of an RBA as a single formula, the *disjunction* of its constraints. In this view, Dokter et al. defines their composition of connectors such that, instead of exploding, the composed connector has transitions and memory cells that are the *sum* of its constituent connectors[DA18].

RBAs have a structure more conducive to *simplification* of the transition space, such that one RBA transition may represent several transitions in a CA. Figure 2.5 shows how how this occurs for the *fifo2* connector. Where the CA in Figure 2.3 must distinguish the cases where A fills m_0 as two separate transitions, the RBA is able to use just one; likewise for the transitions representing cases where B is able to drain m_1. This 'coalescing' of transitions in RBAs is possible owing to the collapsing of their state space. Even without an intuitive understanding of why such transitions can be collapsed, such cases may often be identified only by inspecting the syntax of the data constraints. For another example of CA, a naïve translation to RBA might produce two transitions with data constraints $m = * \wedge X$ and $m \neq * \wedge X$ for some X, which are both covered by a single data constraint X. As both RBA and RBF share this property, we usually refer to RBA transitions and RBF disjuncts as *rules*, giving these models their name. By distinguishing CA

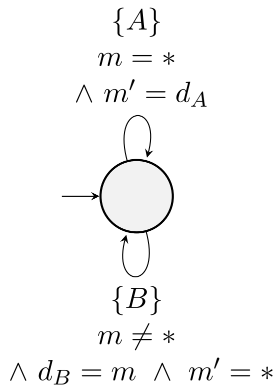

Figure 2.4: RBA of the *fifo1* connector for an arbitrary data domain common to ports A and B. Memory cell m is used both to buffer A's value, and as part of the data constraint on both transitions for *emptying* and *filling* the cell to ensure these interactions are always interleaved. Data constraints are formulated for readability such that the 'guard' and 'assignment' conjuncts are line-separated.

transitions from RBA rules in terminology, we are perhaps more cognizant of the latter's increased ability to *abstract* away needless data constraints.

Typically, Reo has used the *Data* domains in both CA and RBA as parallels to the data-types of the ports. In most of the languages in which Reo protocols are implemented, the discriminants of such types are not distinguished statically. For example, the C language lacks a way to statically enforce a that function `void foo(int x)` is only invoked when x is prime. Instead, checks at runtime are used to specialize behavior. On the other hand, the state-space is simple enough to afford a practical translation into the structure of the program itself, requiring no checking at runtime. For example, Listing 1 shows an intuitive representation of a connector that alternates between states A and B, getting data x from its environment in A, and emitting x when $x = 3$. Observe that there is no need to protect operations behind a runtime-check of *which* state the corresponding CA is in. This observation has implications for the behavior of implementations of RBAs, as they 'cannot remember' which state they are in and must thus perform more checking. In practice, the overhead of this checking is manageable, and does not *explode* under composition as the state space of CAs tend to do. The representation of automata in programming languages is explored in more detail in Section 2.2.2

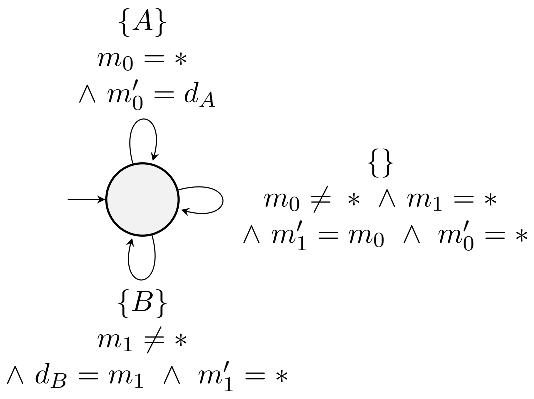
$\{A\}$
$m_0 = *$
$\wedge\ m_0' = d_A$
$\{\}$
$m_0 \neq *\ \wedge m_1 = *$
$\wedge\ m_1' = m_0\ \wedge\ m_0' = *$
$\{B\}$
$m_1 \neq *$
$\wedge\ d_B = m_1\ \wedge\ m_1' = *$

Figure 2.5: RBA of the *fifo2* connector for an arbitrary data domain common to ports A and B. Memory cells m_0 and m_1 are drained by B in the order they are filled by A, and have a capacity of 2 elements. Data constraints are formulated for readability such that the 'guard' and 'assignment' conjuncts are line-separated.

2.1.5 The Reo Compiler

TODO ask Sung to summarize the history of the Reo compiler. give a summarized story here.

The compiler aims to take the low-level implementation of a protocol out of the application developer's hands. Given a protocol specification, the compiler generates the *glue code* and

TODO focus on RBA

The steps from Reo specification to the generated glue code can be better understood when broken down into stages:

1. **Specification expansion**
   The composed definitions of Reo components are unrolled to the channel-level until the protocol is represented by one large automaton with many nodes.

2. **Minimization**
   Nodes not in the protocol's interface are *hidden* and the RBA is minimized. This step produces a new, simpler automaton with the same behavior and interface.

3. **TODO PUTTERS AND GETTERS**

```
void stateA() {
  this.value = get();
  if (this.value == 3) {
    stateB();
  } else {
    stateA();
  }
}
void stateB() {
  put(this.value);
  stateA();
}
```

Listing 1: An example of a program which implements a two-state automaton in the Java programming language. Observe that the behavior of states A and B are encoded implicitly in the *structure* of the program, while determining which of the two in A are available A requires a check ar runtime.

4. **Linking and Code Generation**
   The finished source code is generated from the resulting internal-representation. Those associated with functions in the target language are linked accordingly, and the rest are parsed and translated from the operational semantics of Reo to suitable target-language operations (such as data movement and duplication). The rules of the internal state are translated to the runtime definition of a protocol component object. An entrypoint to instantiating this protocol object is generated with the appropriate interface. The specifics of this step vary per target language.

2.2 Affine Types

[Wal05]. look for linear logic. proof theory. look into Rust's motivations. We can't pull things out of the blue.

talk about TYPE-STATE pattern (aka state machine pattern) https://hoverbear.org/2016/10/12/ru
state-machine-pattern/ http://cliffle.com/blog/rust-typestate/

```
pub struct X([u32;10]);
pub struct Y([u32;10]);
pub fn convert(x: X) -> Y {
Y(x.0)
}

pub fn do_thing_1(x: X) -> u32 {
x.0[0]
}
pub fn do_thing_2(x: X) -> u32 {
let y = convert(x);
y.0[0]
}
-------------------
example::convert:
mov     rax, rdi
mov     rcx, qword ptr [rsi + 32]
mov     qword ptr [rdi + 32], rcx
movups  xmm0, xmmword ptr [rsi]
movups  xmm1, xmmword ptr [rsi + 16]
movups  xmmword ptr [rdi + 16], xmm1
movups  xmmword ptr [rdi], xmm0
ret

example::do_thing_1:
mov     eax, dword ptr [rdi]
ret

example::do_thing_2:
mov     eax, dword ptr [rdi]
ret
```

Type systems exist for the sole purpose of constraining which programs can be built, and do not add any expressiveness in terms of what can be computed. However, we know that adding (sensible) restrictions provides us other important properties. By constraining ourselves in some part, we drastically increase the number of properties that other parts are safe to assume. Affine types are a type system that applies this reasoning to explicitly managing access to variables, thus, allowing us to statically reason about resource management in a more fine-grained way.

2.2.1 The Rust Programming Language

introduce Drop, Move, Clone, Copy

Aside from some unusual exceptions[2], all values in rust can be *moved*, which describes a *value* being transferred between variable bindings, or into functions as arguments, as demonstrated in Listing 2. Clearly, the Rust compiler tracks which values have been moved. Aside from preserving affine properties, this is necessary for determining when a value should be *dropped*. `drop` is the *destructor* function invoked by the compiler on a type when its binding goes out of scope and it has not been moved.

TODO we use snippets from rust, but omit clutter such as visibility qualifiers, imports and sometimes variable names TODo enum vs struct.

```
1 struct Foo;
2 fn func(x: Foo) {
3   // this function takes argument `x` by value.
4 }
5 fn main(){
6   let x = Foo; // instantiate.
7   func(x);     // Ok. `x` is moved into `func`.
8   func(x);     // Error! x is used after move.
9 }
```

Listing 2: Type `Foo` is affine. On line 7, x is moved into function `func`, consuming it. Accessing x is invalid, and so line 8 raises an error.

2.2.2 The Type-State Pattern

The *state* or *state machine* pattern refers to the practice of explicitly checking for or distinguishing transitions between and requirements of states in a stateful object[3]. Usually, these states are distinguished in the data domain of one or more types. Even the lowly `Option` type can be viewed as a small state machine as soon as some condition statement specializes operations performed with it.

The *type state* pattern is closely related, but as the name suggests, it is characterized by encoding states as types, which usually are distinct from

[2] TODO pinned objects

[3] Usually, we disregard the effects of terminating the program. Equivalently, this pattern only allows one to describe automata in which every 'useful' state reaches some final 'terminated' state.

data in their significance to a language's compiler or interpreter. A common approach is to instantiate one of the state types at a time. As an example, consider the scenario where a program wants to facilitate alternation between invoking some functions `one` and `two` which repeatedly mutate some integer n. Listing 3 gives an example of what this might look like as a *deterministic finite automaton* in the C language. In this rendering, the expression `two(one(START)).n` evaluates to the expected result of $(0+1)\cdot 2 = 2$. Even for this simple example, the encoding of states as types in particular has its benefits; the expression `one(two(START))` may appear sensible at first glance, but the compiler is quick to identify the type mismatch on the argument to `one`, making clear that the expression does not correspond to a path through the automaton:

```
note: expected 'DoTwo' but argument is of type 'DoOne'
```

The type state pattern can be applied in any typed language, but it is particularly meaningful in languages where the compiler or interpreter *enforces* its intended use. The example above demonstrates some utility, but a language such as C has no fundamental way to prevent the programmer from *re-using* values. If the programmer misbehaves, they can retain their previous states when given new ones, and then invoke the transition operations as they please. It's not much of a state machine if all states coexist, is it? This is not always a problem in examples such as the previous. Here, the types prevent the construction of mal-formed *expressions*, and perhaps this is enough. However, we cannot so easily protect a resource from any side effects of `one` or `two`; imagine the chaos that would result from these functions writing to a persistent file descriptor.

An affine type system overcomes the shortcoming illustrated above. Formally, *affine types* correspond with *affine* substructural logic, in which the structural rule for 'weakening' is absent; essentially, these logics do not consider terms to be idempotent. By treating instances of these types as affine *resources*, the programmer cannot retain old states without violating the affinity of the types. The example looks very similar when translated to Rust, but now a case such as that shown in Listing 4 will result in the compiler preventing the retention of the variable of type `DoOne`.

2.2.3 Proof-of-Work Pattern

Section 2.2.2 demonstrates how the type-state pattern can be used as a tool to *constrain* actions the compiler will permit the program to do. Indeed, this is a natural parallel to the affinity of the type system, which guarantees

```c
typedef struct DoOne { int n; } DoOne;
typedef struct DoTwo { int n; } DoTwo;
const DoOne START = { .n = 0 };

DoTwo one(DoOne d1) {
  DoTwo d2 = { .n = d1.n + 1};
  return d2;
}
DoOne two(DoTwo d2) {
  DoOne d1 = { .n = d2.n * 2};
  return d1;
}
```

Listing 3: An example of the type-state pattern in the C language. The alternating invocation of `one` and `two` is translated to type-checking the compiler can guarantee. This example guarantees that well-formed *expressions* can be interpreted as valid paths in some corresponding automaton, as the types must match.

that no resource is consumed repeatedly. The counterpart to affine types is *relevant* types, which defines correctness as each resource being consumed *at least once*. Type systems that are both relevant and affine are *linear*, such that all objects are consumed exactly once.

There is no way to create true relevance or linearity in user-space of an arbitrary affine type system; any program which preserves affinity is able to exit at any time without losing affinity. How are we able to enforce a behavior if it is correct to exit at any time? *Proof-of-work* is a special case of the type-state pattern which allows the expression of a relevant type *under the assumption that the program continues its normal flow*; ie. system exits are still permitted. The trick to enforcing the use of some object `T` is to specify that a type is a function which must *return* some type `R`, and to ensure that `R` can *only* be instantiated by consuming `T`. Clearly, we cannot prevent `T` from being destroyed in some other way, but we are able to prevent `R` from being *created* any other way.

Realistic languages have many tools for constraining what users may access. Java has *visibility* to prevent field manipulations. Rust has *orphan rules* to prevent imported traits from being implemented for imported types. Languages without any such features won't be able to prevent users from creating the return type `R` without consuming `T`. In these cases, another option is *generative types* which, among other things, allow us to further

```
fn main(d1: DoOne) {
  let d2  = two(d1);
  let d1  = one(d2);
  let d2  = two(d1);
  let d2_again = two(d1); // Error! `d1` has been moved.
}
```

Listing 4: A demonstration of how the type-state encoding shown in Listing 3 can leverage affine types to ensure that not only expressions, but *a trace through execution* can be interpreted as valid paths through some corresponding automaton. The compiler correctly rejects this example, which corresponds with attempting to take transition `two` twice in a row.

distinguish types with different origins. Here, generative types may be used to ensure not just *any* `R` is returned, but a particular `R` within our control. As this work uses the Rust language for concrete implementations, we will rely on its ability to prohibit the user from creating `R` by using *empty enum types* for types with no data nor type constraints, and by making its fields and constructors *private* otherwise[exo].

Consider the following illustrative scenario: We wish to yield control flow to a user-provided function. Within, the user is allowed to do whatever they wish, but we require them to invoke `fulfill` exactly once (which corresponds to 'consuming `R`'). How can we express this in terms the compiler will enforce? Listing 5 demonstrates a possible implementation (omitting all but the essence of 'our' side of the implementation). The user's code would then be permitted to invoke `main` with their own choice of callback function pointer. Our means of control is the interplay between dictating both (a) the *signature* of the callback function and (b) prohibiting the user from constructing or replicating `Promise` or `Fulfill` objects in their own code.

```
struct Promise;
struct Fulfilled;

fn fulfill(p: Promise) -> Fulfilled {
  // invoked once per `main`
  return Fulfilled::new();
}

fn main(callback: fn(Promise)->Fulfilled) {
  // ...
  let _ = callback(Promise::new()); // `Fulfilled` discarded.
  // ...
}
```

Listing 5: A demonstration of proof-of-work pattern. Here, the user is able to execute `main` with any function as argument, but it must certainly invoke `fulfill` exactly once.

Part II

Contributions

Chapter 3

Protocol Translation

In this section we describe the details that pertain to the process of translating reo connectors from one form to another. the notion of "runtime" becomes less useful when so many different tools are interacting, but this is everything but the runtime of the target application.

3.1 Two-Phase Generation

In previous works, reo generates the target code directly.

3.1.1 Motivation

Reo's Java back-end generates t

3.1.2 Imperative Form

In this work, we introduce a new intermediate representation for the reo compiler, which we will refer to as *imperative form*. based on what can be described in an RBA.

3.1.2.1 Data Actions

we implement synchronous interactions as a sequence of actions. this takes some care, as this translation gets hairy. eg: overwriting memory in the correct order.

3.1.2.2 Rules as Transactions

The Reo compiler has support for the TRANSFORM channel, which essentially allows the invocation of arbitrary functions in the circuit. eg $A = f(B)$ this gets complicated because these operations can be used in guards, assignments or both (as in $A = f(B) = m'$) the counter-intuitive thing is that the value must be computed to decide IF the rule may fire, yet we don't want the value if it does not. Instead, we model rule firings as TRANSACTIONS which can be rolled back. for every rule the last chance to roll back must precent the first VISIBLE EFFECT.

-we support this functionality with INSTRUCTIONS which are performed in sequence. these instructions can perform operations on data, such as creation of new elements, and it may trigger a ROLLBACK

3.2 Code Generation

3.2.1 Reo Side

we discuss the task of translating a declarative reo description into imperative form

3.2.1.1 Compiler Internal Representation

compiler internal repr. is very similar to RBA with some extra information such as the direction of ports. some information is provided, such as figuring out the ASSIGNMENT to ports and memory cells

3.2.1.2 Group by Resource

primarily want to find the mapping from putter to getters. this follows the FLOW of the DATA. ultimately we need

3.2.1.3 Type Constraining

once the flow has been determined, we can see which operations must be supported on the target thing: mostly equality and clone operations. we wish to generate GENERIC protocols. so this requires TYPE CLUSTERNG and CONSTRAINING

3.2.2 Rust Side

imperative form is still too high-level to be used directly. what happens next is very language specific, and can be approached many ways. the section to follow explains how the final transformed rust works at runtime. Here, we concentrate only on getting it into that form.

3.2.2.1 Runtime Interpreter

many approaches are possible for translating the imperative form. in this work we go with an approach that is relatively abstract by implementing a lightweight INTERPRETER that is essentially instantiated with a small program that corresponds strongly with the abstract interpretive form. This was done for several reasons: 1. extensive virtualization 2. pushes protocol instantiation later into the instantiation process so more decisions can be made at the last moment 3. represents more stuff as DATA relying less on monomorphization from generic types (as is the usual rust idiom), which makes it more inter-operable with C 4. a data-oriented representation is more friendly to FFI. 5. runtime configuration

3.2.2.2 Checking and Errors

Assuming that they are described using conventional data structures (sets, maps, lists, etc.) rba rules can be malformed. for example, checking a value not in the firiing set. As Imperative form orients its description according to TIME and not VALUE, the operations on the same values are decoupled and spread out, creating many new possibiltiies of ill-formed operations. for example, moving a value that is only constructed later. These problems are familiar to sequential programming, as expected. the parallel is trying to manipulate a variable out of scope. etc. To complement the throughline of "separating concerns", building a protocol object can FAIL, returning error. some failures are necessary for the build phase (eg: the builder does not know how to represent the movement of a value that has not been defined, as it does now know its data type), and other checks can result in sensible protocol objects but which would do something INVALID at runtime. for example, it would be possible to represent the consumption of a value whose name is in scope, but known to be EMPTY, but it would result in reading uninitialized memory at runtime.

3.2.2.3 Optimization

the INNER form of the rust protocol objects are indeed translated into a still more-complicated form for the purposes of optimization: 1. uses representations which are less convenient to use, but run more efficiently for their intended purpose (eg fusing port and memory readiness as they are usually checked togehter) 2. assumes that it is WELL FORMED to save the time that would be spent checking (eg: it can assume that a putter and getter expect the same data type. If it knows one, it does not need to look up the other)

3.2.3 Function Calls

Reo and RBAs in general support syntax for the TRANSFORM channel. these channels are particularly interesting because they blur the line between coordination and computation. in Reo, transform channels are intuitive: they output something which is a transformation of their input. done asynchronously, it is very clear to see how this would work. Unfortunately, Reo is synchronous which creates an interesting problem. Here, we see an instance of it in an RBA constraint: $f(P_0) = f(P_1) = C$. Let P_0 and P_1 be producers (they output into the protocol), and C is a consumer, and f is some unary function. This formulation has an interesting interpretation in the imperative realm: C gets some datum that is computed using f, but *only* if $f(P_0) = f(P_1)$. As f is some arbitrary function, we must invoke f before we know whether the result should be observed by C.

Chapter 4

Protocol Runtime

4.1 Examining the Java Implementation

The work of this project can draw from the efforts of previous work on the Reo Compiler. The Java implementation in particular has seen the most frequent and recent updates. This section treats the Java code generator as a touchstone for Reo-generated application code in general. We give a brief overview of the properties inherent to the generated code, and consider the effects of projecting the underlying ideas to the Rust language.

4.1.1 Structure: Ports, Threads and Components

Fundamentally, the generated code adheres closely to Reo's literature, revolving around the interplay between `Port` and `Component` objects. From the perspective of a developer looking to integrate a generated Java protocol into their application, the entry point is the `Protocol` component (where 'Protocol' is the name of the associated Reo connector).

Running a system requires an initialization procedure: (1) a `Port` is instantiated per logical port, (2) a `Component` is instantiated per logical component, and (3) pairs of components are linked by overwriting a port-field for both objects with the same instance of `Port`. To get things going, each component must be provided a thread to enter it's main loop; in idiomatic Java, this manifests as calling `new Thread(C).start()` for each component `C`. A simplified example of the initialization procedure is shown in Listing 6 for the simple 'sync' protocol which acts as a one-way channel. In this example, the ports are of type `String`.

In a sense, this implementation primarily hinges on `Port` as a communication primitive between threads, and equivalently, between components.

```java
Port<String> p0 = new PortWaitNotify<String>();
Port<String> p1 = new PortWaitNotify<String>();

Sender c0 = new Sender();
Receiver c1 = new Receiver();
Sync c2 = new Sync();

p0.setProducer(c0); c0.p0 = p0;
p0.setConsumer(c2); c2.p0 = p0;
p1.setProducer(c2); c2.p1 = p1;
p1.setConsumer(c1); c1.p1 = p1;

new Thread(c0).start();
new Thread(c1).start();
new Thread(c2).start();
```

Listing 6: A simplified example of initialization for a system centered around a `Sync` protocol object, which acts as a channel for transmitting objects of type `String`. Both ports and components are constructed before they are 'linked' in both directions: each port stores a reference to its components, and each component stores references to its ports. The system begins to *run* when each component is given a thread and started.

For matters of concurrency, operations on port-data involves entering a *critical region*. In contrast, `Components` are used only to store their ports and to be used as name spaces for their `run` function which implements their behavior (which corresponds to RBA rules in the case of the protocol component). Essentially, anything that interacts with `Port` objects can reify a logical component, whether or not this is done by an object implementing the `Component` interface.

4.1.2 Behavior: Rules

The representation of protocol rules is very intuitive; a rule is implemented as a block of code which operates on a component's ports. Once generated into Java, the only obvious sign that a component was generated from Reo is its linkage to multiple other components[1]. The (simplified) generated

[1] The distinction between 'protocol' and 'compute' components is tenuous at the best of times. If compute components are allowed to interact directly with one another, the distinction observed here disappears also.

Component code of the ‘sync’ protocol from the previous section is shown in Listing 7. This demonstrates that rules are indeed *commandified*, in that their behavior is encoded in discernible structures (appropriately called Command).

The behavior and structure of a component go together, and are generated by Reo at a relatively granular level. As such, the encoding of memory cells is natural also. Memory cells can be found next to ports in the fields of a Component.

4.1.3 Observations

It is very easy to see the correspondence between a generated Java protocol and its Reo definition. This carries over to how components and ports are used by an application developer. Next, we consider their higher-level properties that follow from the observations in the previous sections:

1. **Protocol Event Loop**
   Protocols are fundamentally *passive* in that they do not act until acted upon. Nevertheless, protocols each have their own dedicated thread that waits in a loop for a *notification* from its monitor. Notifications originate from a component’s own Ports in the event of a put or get invocation. For this reason, protocols and components are related in both directions, afforded by setting a port variable in one direction, and functions setProducer and setConsumer in the other.

   True to the intuition behind the RBA model, the protocol must *check* which (if any) commands can be fired, and keep spinning, trying rules while *any* guard is satisfied. This is unfortunate, as this approach requires guards to be evaluated repeatedly. As the protocol relies on the actions of other components to make progress, it is counter-productive for it to spend a lot of system resources evaluating guards to *false*. In cases where threads must share processor time, the excessive work of the protocol component will begin to get in the way of other components making progress, in turn leading yet more guards to evaluate to *false*.

2. **Reference Passing**
   Java is a managed programming language whose garbage collector is central to how the language works. To support the transmission of arbitrary data types, Port is generic over a type. The language only supports this kind of polymorphism for objects. Unlike primitives (such

as `int`), the data for objects is stored on the heap and is garbage collected by the Java Virtual Machine. Variables of such objects are therefore moved around the stack by *reference*. Moving and replicating values is cheap and easy, as they always have a small (pointer-sized) representation on the stack.

A minor drawback is the need for indirection when performing operations that need to *follow* the reference. For example, comparing two `Integer` objects requires that the `int` primitives backing them on the heap be retrieved and compared. Equality is an example of an operation that the Reo protocol thread can be expected to perform frequently. The cost of this indirection depends on a myriad of factors, but is at its worst when it results in new, spread-out locations each time. This case might arise, for example, if the `Sender` continuously created new `Integer` objects and sent them through its port. Another drawback is the *requirement* to allocate primitives on the heap before they can be sent through a port. This is not usually a problem in the case of Java, as in practice, almost everything is going to be stored on the heap with or without Reo.

This aspect of the generated Java code will require the most change for the Rust version, as Rust has a very different model for memory management; it does not use a garbage collector by default, and structures are stored first and foremost on the *stack* as in the C language.

3. **Two Hops for Data**
   As protocols are components like any other, even the most trivial of data-movements require values to hop at least twice: into the protocol, and out of the protocol. Fortunately, as stated above, the cost of the 'hop' itself is trivial, as it will always be a small reference. The problem is the time delay *between* the hops, as it will often involve actions of three distinct threads in series (with the protocol in the middle).

4. **Vulnerable to User Error**
   The construction and linking of components with ports is not something the protocol itself is concerned with. Indeed, *every* component assumes that their port-variables will be initialized by their environment. At the outermost level, this environment is in the application developer's hands. Components make no attempt to verify that they are correctly linked according to the specification; currently, there is not any infrastructure in place to support this checking if it were desired. As a result, it is possible make mistakes such as fusing two of

a protocol's ports into one. Whether this is a problem worth solving depends on the burden of responsibility that Reo intends to place on the end user. These difficulties cannot be completely avoided, but approaches exist to minimize these opportunities for mistakes.

While ports are clearly directional 'from the inside out' (ports store distinct references to their producer and consumer components), the same is not so 'from the outside in'. Neither of a port's components is prevented from indiscriminately calling `put` or `get`. The assignment of a port's values for 'producer' and 'consumer' component is in userspace also. As a consequence, these fields may not agree with the components that interact with the ports at all. In fact, any number of components may store a reference to a port, each arbitrarily calling `put` and `get`. If done unintentionally, this would lead to *lost wakeups*; the thread blocking for a notification after calling acting on the port is not the same as the thread receiving the notification. Solutions can be conceived to *wrap* ports in objects that constrain the API of a port to one of the two 'directions'. However, without affine types, there is no obvious way to ensure the *number* of components accessing a port is correct. In Rust, limiting these accesses becomes feasible.

5. **Port Data Aliasing**
   In Reo, it is common for connectors to replicate port data. Owing to the nature of Java, this is currently achieved by duplicating references, where replication is also known as *aliasing*. For immutable objects, aliasing has no observable side effects, and thus does not threaten Reo's value-passing semantics. However, Reo ports permit instantiation with *any* object-type. Even if the operations are thread-safe, this causes *incorrect* behavior, as a component might observe their data changing seemingly under their feet. Worse still, objects which are not thread-safe can cause undefined behavior. This is a result of Java's view on memory safety having inverted priorities to Rust. In Java, operations are unsafe by default, and the programmer must go out of their way to protect themselves from data races, access of invalid memory and corruption. In Rust, the *ownership system* is based on the prohibition of mutably-aliased variables. Achieving replication in Rust will require some effort to convince the compiler of safety before a program will compile.

6. **Non-Terminating Protocols**
   Currently, Reo-generated protocol objects loop forever unless they raise

an exception and crash. For protocols that can perform actions with observable side-effects in the absence of other components, this is perhaps a good idea. However, in the majority of realistic cases, protocols are indeed passive, and cannot do meaningful work as the only component. Reo semantics tend to reason about *infinite* behaviors. However, real programs often do *end*, and it is desirable that the program's exit is not held up by an endlessly-blocked protocol thread.

7. **Protocol Components Cannot be Composed at Runtime**
   (TODO is this the place to explain this?) Ports allow data to move from the putter (or 'producer') and getter (or 'consumer') components as an *atomic* operation by delaying `put` or `get` operations until their counterpart is called also. This causes problems for the implementation of RBAs with rules whose guards are predicated by the data they move. How can a protocol *decide* if it should fire as a function of values it can only obtain *by* firing? This ability to reason about the future is currently still a luxury limited to models such as TDS. The Java implementation gets around this problem by introducing *asymmetry* between 'compute' and 'protocol' components. Protocols are allowed to *cheat*. The `Port` object has additional operations to inspect a value without consuming it: `peek` and `hasGet`. However, this asymmetry means that composing two Java protocol components (by linking them with ports) does *not* result in a component with their composed behavior. Solving this problem in earnest requires continuously-connected protocols to reason about their distributed state, which is a problem beyond the scope of this work. Reo's relationship with *liveness properties* is explored in Section 5.

8. **Sequential Coordination**
   The Java implementation is structured such with *ports* being the critical region between components. As protocols have multiple ports, at first glance it may appear that coordination events could occur in parallel. However, no communication through protocol P happens without the single thread in P's `run` method. Indeed, `put` and `get` operations can be *started* in parallel by the boundary components, but P can only complete it's half of these operations sequentially.

4.2 Design Goals Defined

The Reo compiler's Java code generator were examined in Section 4.1, resulting in the extraction of some high-level observations, enumerated in Section 4.1.3. Below, the design goals of the Rust code generator are enumerated a list of *deviations* from the standard set by the Java version. Each goal is given a name such that it can be referred to in Sections **??** to **??** to follow.

G_1 Prohibit aliasing where it would result in behavior disallowed by Reo's value-passing semantics.

G_2 Relax the requirement of port-data to be heap-allocated.

G_3 Minimize the number of times values with large on-stack representations must be moved.

G_4 Minimize the overhead experienced as a result of protocol components repeatedly evaluating arbitrarily expensive guards.

G_5 Enforce that ports be acted upon by two components: one acting as producer, and the other acting as consumer.

G_6 Protect the end user from being able to run protocols with uninitialized or fused ports.

G_7 Facilitate termination detection as defined by all *non-protocol* components exiting.

4.3 Goals

4.3.1 Functional Requirements

features, safety

4.3.1.1 Features

4.3.1.2 Safety

4.3.2 Non-Functional Requirements

performance, maintainability

4.4 Runtime Properties

4.4.1 User-Facing

4.4.1.1 Protocol Construction

4.4.1.2 Port Construction

4.4.1.3 Destruction and Termination

4.4.2 Internal

4.4.2.1 Protocol Object Architecture

4.4.2.2 Rule Firing

4.4.2.3 Design Choices and Optimizations

```
private static class Sync implements Component {
    public volatile Port<String> p0, p1;

    private Guard[] guards = new Guard[]{
        new Guard(){
            public Boolean guard(){
                return (p1.hasGet() && (!(p0.peek() == null)));
    }   },  };

    private Command[] commands = new Command[]{
        new Command(){
            public void update(){
                p1.put(p0.peek());
                p0.get();
    }   },  };

    public void run() {
        int i = 0;
        while (true) {
            if(guards[i].guard())commands[i].update();
            i = i==guard.length ? 0 : i+1;
            synchronized (this) {
                while(true) {
                    if (p1.hasGet() && (!(p0.peek() == null))) break;
                    try {
                        wait();
                    } catch (InterruptedException e) { }
}   }   }   }   }
```

Listing 7: A simplified example of a Reo-generated Java protocol class for the *sync* connector. By convention, it is started by invoking `start`, which is a method inherited from the `Runnable` interface which `Component` extends. This method assumes that all ports are correctly initialized and linked to another 'compute' port. Its RBA-like behavior comes from an array of guards and commands which it iterates over in a loop, firing rules as possible forever.

Chapter 5

Generating Static Governors

A protcol's *governor* acts to ensure that all the actions of a component are *adherent* to the protocol with which it interfaces, guaranteeing that its actions will not violate the protocol. In this section, we develop a means of embedding governors into Rust's affine type system. As a result, an application developer may ergonomically opt-into checking protocol adherence of their own compute-code using their local Rust compiler, whereafter successful compilation guarantees adherence to the protocol.

In more precise terms, let protocol A be *protocol adherent* to protocol B if and only if the *synchronous composition* of A and B is language-equivalent to B; equivalently, A is adherent to B and A adheres to B. This can be understood as A contributing no constraints to the composed system that B did not have already.

5.1 The Problem: Unintended Constraints

A central tenet of Reo's design is the *separation of concerns*, part of which is the desire to minimize the knowledge a compute component must have of its protocol. In this view, coordinating the movements of data is not a concern relevant to the task of computation. A desirable balance is possible with the observation that protocol objects are able to partially impose protocol adherence on their neighbors; External ports may instigate a `put` or `get` at any moment, and the *coordinator* will complete them as soon as the protocol definition allows it. In this way, coordinators possess a crucial subset of the features of *governers*: aligning the *timing* of two actions that compose an interaction. Unfortunately, in the properties of the realm of sequential, action-centric programming itself *implicitly* imposes constraints on the behavior of the system: `put` or `get` block until their interaction is completed, and no subsequent code (potentially, other port operations) will occur until they do. This is beyond the capabilities of the coordinator to influence.

In the context of application development, this has an interesting consequence; the behavior of the system is influenced by the behavior of (potentially) all of its components. This is sensible in theory, but becomes unwieldy in practice. Even small changes to the behavior of a compute component influences the system's behavior in unexpected ways, as we are not used to thinking about synchronous code as a composable protocol, nor are we able to intuit the *outcome* of the composition. For example, Listing 8 gives the definition of a compute function which a user may write to interact with a protocol. When `p` and `g` are connected to a *fifo1* protocol (which forwards `p` to `g`, buffering it asynchronously in-between), it runs forever and the output will be something like: `I saw true. I saw false. I saw true. I saw false. (...)`. However, when connected with the *sync* protocol (which forwards `p` to `g` synchronously), the system has no behavior. The problem is that even though *fifo1* and *sync* have the same *interface*, `transform_not` is *compatible* (can be made to adhere) with the former and not the latter. By definition, *sync* fires when *both* `p` and `g` are ready, but `transform_rot` does not `put` until the `get` is completed. Once the intricacies of these programs grow beyond a programmer's ability to keep track of these relationships, the composed system may have *unintended* behavior. This property may be obvious at the small scale of this example, but it becomes more difficult the larger and more complex the program becomes. In the worst cases, an innocuous change makes an interaction becoming unreachable, manifesting at runtime as *deadlock*.

```rust
fn transform_not(p: Putter<bool>, g: Getter<bool>) {
    loop {
        let input: bool = g.get();
        print!("I saw {}.", input);
        p.put(! input);
}   }
```

Listing 8: A function in Rust which can be used as a compute component in a system, connected to a protocol component.

5.2 Governors Defined

In this work, we accept that it is necessary to write compute code that has blocking behavior. Rather than attempting to empower the coordinator with the ability to further influence its boundary components, we introduce explicit governors into our applications such that from the protocol's perspective, the components appear to manage themselves. A particular compute component requires a particular governor as the behavior permitted to the compute component is a function of its *interface* with the protocol.

Ultimately, all governors have in common that they enforce adherence to a given protocol on the components they govern. However, governors may differ on *when* and *how* this enforcement manifest. For example, a governor may intercept and filter network messages at runtime, while another checks for deviations *statically* and emitting compiler errors.

In this work, we leverage the unique expressiveness of the Rust language by creating a tool which generates protocol-specific governor code. When used by an application developer, these governors assess the protocol adherence of compute functions *statically*, and prevent compilation if deviations are detected. As such, these governor are absent from the compiled binary.

5.3 Solution: Static Governance with Types

TODO

5.4 Making it Functional

This section details the workings of the **Governor generator** tool which generates Rust code given (a) a representation of a protocol's RBA, and (b) the set of ports which comprise the interface of the compute component to be governed.

5.4.1 Encoding CA and RBA as Type-State Automata

The *type state* pattern described in Section 2.2.2 provides a means of encoding finite state machines as affine types. Their utility is in guaranteeing that all runtime traces of the resulting program correspond to runs in the automaton. For this class of machines, the encoding is very natural, as there can be a one-to-one correspondence between the states of the abstract automaton, and the types required to represent them. This is also the case for transitions and functions; in the worst case, this mapping is one-to-one also. For an arbitrary transition from states a to b with label x, a function can be declared to consume the type for a, return the type for b, and perform the work associated with x in its body.

The encoding is more complicated for CA, where not only states but data constraints must be encoded into types and must interact with transitions. One approach is to treat *configurations* as *states* were treated before by enumerating them into types. For example, the configuration of state q_0 with memory cell $m = 0$ is represented by type `q_0_0`, while state q_0 with $m = 1$ is represented by `q_0_1`. On a case-by-case basis, one might be able to represent several configurations using one type in the event these configurations are never *distinguished*. For example, a connector may involve positive integers, but only distinguish their values according to whether they are *odd* or *even* and nothing else; in this case {`q_0_0`, `q_0_1`, `q_0_2`, ...} may be collapsed to {`q_0_odd`, `q_0_even`}. For an arbitrary case unique types are needed for every combination of state with every value of every variable. As RBAs are instances of CA, we are able to represent them using the same procedure. As RBAs are used by both the Reo compiler and Reo-rs, they are the model of choice for governors also.

5.4.2 Rule Consensus

Thus far, we have reasoned about operations on RBAs that preserve their ability to simulate the non-silent port actions of the original. However, at runtime the protocol's state will follow a *particular* path, which may not be the only one possible. When two distinct paths branch out from the current configuration, which one should the governor follow such that it can enforce the correct actions? Consider the protocol *fifo2* once again, and observe that from type-state `(E,F)`, two rules may be fired next, one firing A and the other firing B. The governor must enforce A if and only if the protocol's state goes down the path for A, and likewise for B. This is an instance of the *consensus* problem; all RBAs in the system must agree on the path taken such that they can proceed in lockstep.

Many synthetic solutions are possible for creating consensus, as we can determine a meta-protocol ahead of time for both governors and protocol to follow such that consensus emerges at runtime. For example, by *ordering* the protocol's rules, and having all parties prioritize rules by ascending order, choice is statically eliminated. Even this simple solution is deceptively nuanced, as the normalization procedure breaks the 1-to-1 correspondence between the rules of the protocol, and the rules of a governor.

Instead, this work takes the approach of statically 'electing' the *coordinator* as the leader in every case, and having all governors follow the lead of its arbitrary choice. This approach is primarily motivated by its *flexibility*; by supporting an arbitrary choice on the part of the protocol, we make the choice itself an *orthogonal* concern for future work. Electing the coordinator in particular as the leader is also a natural choice, as it is the only RBA in the system with a complete view of the protocol's state, and thus can make the choice as a function of its *un-approximated* configuration, as well as the values of all port-putters dynamically.

In terms of implementation, the Rust function for a rule no longer returns a *particular* type-state token, but rather a `StateSet` object representing an indeterminate state-type which will *later* be chosen from the elements of the set. This type is entirely opaque other than a function to codedetermine it. At runtime, *determine* blocks until the governor receives a message from the protocol, communicating a particular choice. For cases where the `StateSet` is a singleton, `determine` simply unwraps the element. While the decision is made at *runtime*, the type-state automaton exists at runtime. To make this possible, the programmer must provide behavior for each *case*, corresponding to elements of the set, one of which will be chosen by `determine`. This use case describes a *sum type*, which is already present in rust as the `enum`, a union type with a set of *variants*. However, every new variant set would necessitate the definition of a new enum. This is impractical, as the number of combinations are large. Instead, we implement our own types which behave and appear to the user much like *anonymous sum types* (which do not currently exist in the rust language). This is achieved by relying once again on Rust's trait system to encode *lists* in type-space, using nested tuples. *Matching* of these nested tuple types translates to peeling away tuple layers.

5.4.3 Governed Environment

TODO

5.5 Making it Practical

5.5.1 Approximating the RBA

5.5.1.1 Motivation

TODO

5.5.1.2 Data Domain Collapse

The approach to generating a type-state automaton from an RBA was given in 5.4.1. A major contributor to the size of these state spaces is the size of the data domain. To proceed we abandon the goal of faithfully representing the entirety of the protocol's configuration space in favor of representing an *approximation* by assuming all data types to be the trivial *unit-type*. With this assumption, memory cells may be in one of two states: (a) empty, (b) filled with 'unit'. Converting existing RBAs may see large sub-expressions of *data constraints* becoming constant, including checks for equality and inequality between port values. This assumption is justified by its relation to Assumption **??** from Section **??**. In this context, it can be understood to mean that *usually*, two configurations that are only distinguished by having different *data values* in memory cells or begin put by putters satisfy precisely the same subset of the RBA's guards. Consequently, they do not need to be distinguished. This simplification greatly reduces the total number of types to encode an RBA's configuration space. However, it is still necessary to consider the possible *combinations* of all empty and full memory cells, requiring potentially 2^N types for N cells. Rather than enumerating these types explicitly, we can rely on the *structure* the RBA provides by simply encoding each automaton configuration as a *tuple* of types `Empty` and `Full`. In a sense, each tuple is indeed its own type, but neither the code generator nor the compiler need to pay the price of enumerating all the combinations eagerly. For example, a configuration of three empty memory cells would be represented by type `(Empty,Empty,Empty)`.

As before, we are able to represent an RBA rule as a *function* in the Rust language by encoding a configuration change from q to p determines its *declaration* such that it consumes the type-state of p and returns the type-state of q. The naïve approach of generating functions per type-state is susceptible to the same *exponential explosion* that plagued CAs in the first place. Fortunately, tuple-types have inherent structure which Rust's generic type constraints are able to understand. The use of generics to *ignore* elements of the tuple coincides with an RBA's ability not *ignore* memory values. Consequently only one function definition per RBA rule is required. The way the rule's data constraint manifests is somewhat different, as our function must *explicitly* separate the *guard* and *assignment* parts and represent them as constraints on the parameter-type and return-type respectively. As an example, Listing 9 demonstrates the type definitions and rule functions for the *fifo2* protocol first seen in Section 2.1.4 with the associated RBA shown in Figure 2.5. Observe that the concrete choices for tuple elements act as *value checks* for memory cells in either empty or full states. Omission of a check must be done explicitly using a *type parameter* such that the function is applicable for either case of `Empty` or `Full`, and to ensure the *new* state preserves that tuple element; this causes memory cells to have the expected behavior of *propagating* their values into the future unless otherwise overwritten by assignments. This serves as an example of a case where our simplification coincides with a faithful encoding of the original protocol as *fifo2* never discriminates elements of the data domains of A and B.

```rust
enum E {} // E for "Empty"
enum F {} // F for "Full"
fn start_state() -> (E,E);

fn a_to_m0<M>(state: (E,M)) -> (F,M);
fn m0_to_m1  (state: (F,E)) -> (E,F);
fn m1_to_b<M>(state: (M,F)) -> (M,E);
```

Listing 9: Type-state automaton for the *fifo2* protocol in Rust. The three latter functions correspond to the three rules seen for the RBA in Listing 2.5. Function bodies are omitted for brevity.

5.5.1.3 RBA Projection

When a protocol's interface is provided as-is to a compute component, its model itself (an RBA in our case) defines precisely what it is permitted to do, just with the *direction* of operations reversed; for the component to be compatible, it must put on port P whenever the protocol gets on P, and get on port Q whenever the protocol puts on port Q. In such cases, the procedure for encoding the RBA described in Section 5.5.1.2 can be applied directly. Otherwise, the interface of a compute component does not subsume the entirety of the interface of its protocol. In such systems, the protocol interfaces with several compute components. Indeed such cases form the majority in practice; compute components tend to only play a small role in a larger system.

The contents of Section 5.5.1.2 are sufficient to generate some functional governors. We consider a system containing protocol P and connected compute component C with interfaces (port sets) I_P and I_C respectively such that $I_P \supseteq I_C$. We wish to generate governor G_C whose task is to ensure that C adheres to P. As a first attempt, we translate P's RBA to Rust

rule	guard	assignment
0	$m_0 = *$	$\wedge\ m'_0 = d_A$
1	$m_0 \neq\ *\ \wedge m_1 = *$	$\wedge\ m'_1 = m_0\ \wedge\ m'_0 = *$
2	$m_1 \neq *$	$\wedge\ d_B = m_1\ \wedge\ m'_1 = *$

Table 5.1: RBF of the *fifo2* protocol, equivalent to the RBA in Figure 2.5. Formatted with an outermost disjunct per line such that guard and assignment parts per rule are discernible.

functions and types as-is. We would quickly notice that the RBA's data constraints represent port-operations that are excluded from I_C. These interactions involve no actions on C's part; from the perspectives of C and G_C, these actions are *silent*. Equivalently, we do not use the RBA of P directly, but consider instead its *projection* onto I_C, which *hides* all actions that are not in the interface projected upon.

```
fn a_to_m0<M>(state: (E,M)) -> (F,M) {
  // A puts
}
fn m0_to_m1  (state: (F,E)) -> (E,F) {
  // silent
}
fn m1_to_b<M>(state: (M,F)) -> (M,E) {
  // silent
}
```

Listing 10: Type-state automaton rules which govern the behavior of a compute component with interface ports $\{A\}$ for the *fifo2* protocol. Function bodies list the *actions* which the component contributes to the system. Observe that rules but 0 are silent.

As an example, we once again generate a governor for a compute-component with interface $\{A\}$ with the *fifo2* protocol. This time the protocol is represented as an RBF in Table 5.1 to make the correspondence to the generated governor in Figure 10 more apparent. Observe that all but one of its rule functions are *silent*, serving no purpose but to advance the state of the automaton by consuming one type-state and producing the next. As demonstrated here, this approach to generating governors is correct, but has two undesirable properties:

1. **API-clutter**

The end-user is obliged to invoke functions which correspond with rules in the protocol's RBA. In many cases, these rules will serve no purpose other than to consume a type-state parameter, and return its successor.

2. **Protocol Entanglement**
   The type-state automaton captures the structure and rules of the protocol's RBA in great detail. This is a failure to *separate concerns*, which further couples the compute component to its protocol. This has the immediate effect of making components difficult to re-use (their implementations are more protocol-specific), as well as making them brittle to *changes* to the protocol, making them difficult to maintain.

5.5.1.4 RBA Normalization

Section 5.5.1.3 introduced a procedure for generating governors, but also discussed a significant weakness; all governors are represented by type-state automata based on the original protocol's rules. In this section, we introduce a notion of *normalization* that intends to *specialize* the governors according to its needs such that it is still 'compatible' with the protocol's RBA in all ways that matter, but has greatly reduced *api-clutter* and *protocol-entanglement*.

Let an RBA be in normal form if it has no silent rules. We observe that the presence of silent rules contributes to both api-clutter and protocol entanglement. Ideally, we wish to abstract away the workings of the protocol as much as possible; at all times, the governor only needs to know which actions the component must perform *next*. To make this notion more concrete, we introduce some definitions which build on one another to define the term we need: our normalization procedure should generate an RBA with starting configuration which *port-simulates* the protocol's RBA in its starting configuration:

- $Act(r)$ of an RBA state r:
  The set of ports in r which perform actions (ie: are involved in interactions).

- *Rule sequence* from c_0 to c_1 of RBA R:
  Any sequence of rules in R that can be applied sequentially, starting from configuration c_0 and ending in configuration c_1.

- P*-final* wrt. port set I:
  A rule sequence of RBA R, with *last* rule r_{last} is P-final with respect to port set I if $Act(r_{last}) \cap I = \{P\}$ and for all rules r in the sequence, $r = r_{last} \vee Act(r) \cap I = \varnothing$.

- RBA R_1 in config. c_1 *port-simulates* R_2 in config. c_2 wrt. Interface I: If for every P-final rule sequence of R_2 starting in c_2, ending in c_2' there exists some P-final rule sequence of R_1 starting in c_1, ending in c_1' such that R_1 in c_1' port-simulates R_2 in c_2'.

The intuition here is that it does not matter how the governor's RBA structures its rules. It is unnecessary for governors to advance in lockstep with the protocol to the extent that they agree on the protocol's *configuration* at all times. It suffices if the protocol and governor always agree on which *actions* the ports in their shared interface do next. Figure 5.1 visualizes this idea; observe how the normalized RBA has entirely different transitions (different labels and configurations), but is ultimately able to pair actions of the protocol for ports in its interface with its own local actions.

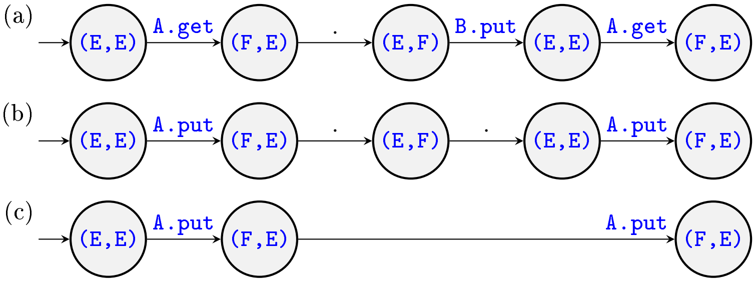

Figure 5.1: Rules being applied to walk three RBAs in lockstep, with time horizontally, showing the (simplified) configurations traversed, and annotating rules by showing which port actions they involve. (a) RBA of protocol *fifo2*. (b) RBA of *fifo2* projected onto port set $\{A\}$. (c) RBA of *fifo2* projected onto port set $\{A\}$ and normalized to remove silent rules.

The final normalization procedure is given in Listing 11 in the form of simplified Rust code. It works intuitively for the most part: silent rules are removed, and new rules are added to retain their contribution of moving the RBA through configuration space. The function `normalize` ensures that the returned rule set is in the same configuration as the protocol after matching a non-silent, but the configuration is allowed to 'lag behind' while the protocol performs rules which it considers to be silent. New rules must be added to 'catch up' to the protocol after any such sequence of silent rules. The procedure does this by building these *composed* rules from front to back, ie. replacing every silent rule x with a *set* of rules $x \cdot y$, where y is any other rule. Once completed, the RBA may contain rules that are subject

```
fn normalize(mut rules: Set<Rule>) -> Set<Rule> {
    let (mut silents, mut not_silents) =
↪   rules.partition_by(Rule::is_silent);
    while silents.not_empty() {
        let removing: Rule = silents.remove();
        if removing.changes_configuration() {
            for r in silents.iter() {
                if let Some(c) = removing.try_compose_with(r) {
                    silents.insert(c);
            }   }
            for r in not_silents.iter() {
                if let Some(c) = removing.try_compose_with(r) {
                    not_silents.insert(c);
    }   }   }   }
    return not_silents;
}
```

Listing 11: Normalization procedure, expressed in (simplified) Rust code. In a nutshell: while one exists, an arbitrary silent rule x is removed, and the list of rules is extended with composed rules $x \cdot y$ such that y is another rule.

to *simplification*. For example, $\{m = * \wedge n = *,\ m \neq\ * \wedge n = *\}$ can be represented by only $n = *$.

The normalization algorithm is **correct** as clearly it does not have silent rules once it returns (`not_silent` containing zero silent rules is invariant). Observe that for each silent rule removed, it does not consider composing with *itself*. The immediate result is that the algorithm never inserts some rule $x \cdot x$ for silent rule x. This is not a problem, as all *silent* rules of our approximated RBAs are *idempotent* with respect to their impact on the configuration. The algorithm is able to take for granted that the result any *chain* of silent rules $x \cdot x \cdot x \cdot ...$ is covered by considering x itself. Furthermore, the incremental removal of rules prohibits the creation of any silent cycles at all. This is due to the reasoning above being extended to any sequences also. (TODO PUMPING LEMMA).

The normalization algorithm is **terminating**. It consists of finitely many *algorithm steps* in which the RBA A is replaced by RBA $B = (A \setminus \{r\}) \cup \{r \cdot x | r \in A \setminus \{x\} \wedge \mathit{composable}(x, r)\}$ for some silent rule $x \in A$. Initially, A is the input RBA with silent rules. The algorithm terminates, returning B when A is replaced by B where B has no silent rules. Let $P(x)$ be the set of *acyclic paths* through RBA x's configuration space. Observe that initially,

rule	guard	assignment
0	$m_0 = *$	$\wedge\ m_0' = d_A$
2	$m_1 \neq *$	$\wedge\ d_B = m_1 \ \wedge\ m_1' = *$
$1 \cdot 0$	$m_0 \neq\ * \wedge m_1 = *$	$\wedge\ m_0' = d_A \wedge m_1' = m_0$
$1 \cdot 2$	$m_0 \neq\ * \wedge m_1 = *$	$\wedge\ m_0' = *$

Table 5.2: RBF of the *fifo2* protocol, projected onto port set $\{A, B\}$ and normalized. Rules 0 and 2 are retained from Table 5.1, and new rules $1 \cdot 0$ and $1 \cdot 2$ are composed of rules from the original RBF.

$P(A)$ is finite. It suffices to show that in each algorithm round, $|P(A)|$ strictly decreases. Within a round, for every 'added' p in $P(B) \setminus P(A)$, p contains a rule $m \cdot n$ such that there exists p' in $P(A) \setminus P(B)$ identical to p but with a 2-long sequence of rules m, n in the place of x. From this we know that $|P(A)| \geq |P(B)|$. However, the 1-long path of x itself is clearly in $P(A) \setminus P(B)$. Thus, $|P(A)| > |P(B)|$. QED.

To demonstrate the normalization procedure, Table 5.2 shows the result of projecting the *fifo2* connector's RBF onto port set $\{A, B\}$ and normalizing. The two additional rules can be understood to 'cover' the behavior lost as a result of omitting the silent rule 1 from the original RBF.

5.5.2 Unknown Memory State

5.5.3 Match Syntax

(TODO) 1. compare enum

Chapter 6

Benchmarking

6.1 Goal

6.2 Experimental Setup

6.3 Results

6.4 Observations

Part III

Reflection

Chapter 7

Discussion

7.1 Future Work

7.1.1 Distributed Components

7.1.2 Runtime Governors

7.2 Conclusion

Bibliography

[ABRS04] Farhad Arbab, Christel Baier, Jan Rutten, and Marjan Sirjani. Modeling component connectors in reo by constraint automata. *Electronic Notes in Theoretical Computer Science*, 97:25–46, 2004.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(3):329–366, 2004.

[DA18] Kasper Dokter and Farhad Arbab. Rule-based form for stream constraints. In *International Conference on Coordination Languages and Models*, pages 142–161. Springer, 2018.

[exo] Exotic sizes.

[JA12] Sung-Shik TQ Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science*, 22(1), 2012.

[Wal05] David Walker. Substructural type systems. *Advanced Topics in Types and Programming Languages*, pages 3–44, 2005.