Shared Memory Implementations of Protocol Programming Languages

DOI:
10.1145/3242947.3242952

Document status and date:
Published: 17/07/2018

Document Version:
Peer reviewed version

Shared Memory Implementations of Protocol Programming Languages, Data-Race-Free*

Micha Hergarden
Open University of the Netherlands

Sung-Shik Jongmans
Open University of the Netherlands
and Imperial College London

Abstract

Protocol programming languages are domain-specific languages that offer higher-level abstractions for programming of synchronization and communication protocols among participants. However, most implementations of protocol programming languages on shared memory architectures use pointer passing to exchange data in communications, so programs can still run into data races. We report on our ongoing efforts toward the first shared memory implementation of a protocol programming language that guarantees freedom of data races, without excessive copying, by leveraging the programming language Rust and its type system.

CCS Concepts • **Software and its engineering → Concurrent programming languages**;

Keywords protocol languages, Rust, Reo

ACM Reference Format:
Micha Hergarden and Sung-Shik Jongmans. 2018. Shared Memory Implementations of Protocol Programming Languages, Data-Race-Free. In *13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'18), July 17, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3242947.3242952

1 Introduction

With the advent of multicore processors, concurrent programming has become an indispensable skill for many general-purpose programmers to master. However, concurrent programming remains difficult: despite new features in general-purpose programming languages that offer higher-level abstractions on top of bare threads and locks (e.g., the fork/join framework in Java; actor-based concurrency in Scala and Erlang; channel-based message-passing in Go and Rust), programmers continue to struggle with classical concurrency errors, such as deadlocks and data races.[1]

A major challenge that programmers of concurrent programs face, pertains to the implementation of *protocols* (i.e., synchronization and communication patterns) among *participants* (i.e., concurrent computations): while general-purpose programming languages offer concurrency primitives to program the local *actions* of participants (e.g., `lock`/`unlock`; `send`/`recv`), they lack linguistic support to ensure local actions truly result in the global *interactions* of the protocol (e.g., "a synchronization between threads T_1 and T_2 is followed by a communication between T_2 and thread T_3"). Aggravated by the many possible interleavings in which threads can be scheduled, purely *action-centric protocol programming* techniques are hard to reason about and error-prone to use.

In recent years, several *interaction-centric protocol programming* techniques have been developed that offer several advantages. The idea is that programmers continue to use an existing *base language* (e.g., Java, C, etc.) to program the sequential computations of a program. Complementary, programmers are also provided a *supplemental language* specifically for protocols (i.e., a domain-specific language), in which they can program the interactions of protocols directly and explicitly. Using such a supplemental language, specifically, programmers can program the desired data exchanges using higher-level and more appropriate abstractions, and automatically generate lower-level code that uses concurrency primitives in the base language. Thus, protocols programmed in the supplemental language are ultimately compiled into code in the base language, after which the whole program can be compiled/run using the base language's standard tools.

This way of working has several key advantages: the protocol code can be considered modularly from the actual computation code, enabling protocol code and computation code to be formally verified separately (e.g., model-checking protocol code [8], or type-checking computation code against local protocol specifications [4]). Modularity also simplifies reuse of both computation code and protocol code in other programs, as the sequential parts can be replaced by other algorithms. Premier examples of interaction-centric protocol programming languages are Reo [1] and Scribble [9].

*Work-in-progress paper

[1] E.g., Gartner (a leading IT advisory company in industry) recently reported that "multicore programming is generally seen as a hard-to-achieve and time-consuming task, so many programmers avoid it as far as possible" [3].

ICOOOLPS'18, July 17, 2018, Amsterdam, Netherlands

2 Research Questions

To avoid data races, a key assumption in the *designs* of many supplemental languages for protocols (including Reo and Scribble) is that every participant has private memory—even if the base language supports shared memory—and that all interaction proceeds via message-passing; under this assumption, data races by definition cannot occur. In the *implementations* of these languages, however, this assumption is not always upheld. Specifically, for base languages with shared memory, the following two approaches have been used to implement communications between participants:

- **ALWAYS-COPY** *The runtime system for the protocol programming language always makes a copy of every value communicated.* The advantage is that freedom of data races is statically guaranteed, because the private memory assumption is faithfully "simulated" by always copying. The disadvantage is that excessively many copies of data may be created (e.g., if a sender does not use a value after sending, no copy is necessary).
- **NEVER-COPY** *The runtime system never makes a copy, relying on the programmer to make a copy upon send and/or receive.* The advantage is that the programmer can fine-tune the number of copies to improve performance; the disadvantage is that the programmer may make too few copies—intentionally (e.g., to maximize performance) or by mistake—so freedom of data races is no longer statically guaranteed.

For instance, the most recent Java implementation of Reo uses only the second approach [6]; the Java implementation of Scribble works with both approaches [5].

In an ongoing research project, we aim to find a middle ground between these two approaches, consolidating their strengths, while alleviating their weaknesses. Specifically, taking the more practical NEVER-COPY approach as our basis, we seek answers to the following research questions:

- **Q1a** How to statically guarantee, using NEVER-COPY, that if a participant P sends a value v, P will not use v after it has sent v? (I.e., P can only use a copy of v.)
- **Q1b** How to statically guarantee, using NEVER-COPY, that if a participant P receives a value v, no other participant will use v after P has received it? (I.e., every other participant can use only copies of v.)
- **Q2** What is the trade-off between freedom of data races and maximal performance (i.e., use NEVER-COPY and knowingly run the risk of data races)?

To resolve **Q1a** and **Q1b**, we need an analysis tool to reason about usage of heap data and aliasing. The type system of the Rust programming language does exactly this. Our approach is, thus, to adopt Rust as a base language, compile an existing supplemental language for protocols to Rust, and leverage Rust's type system to statically guarantee freedom of data races. Doing so, we aim to develop the first shared memory implementation of a protocol programming language that guarantees freedom of data races, without excessive copying.

(a) Alternator **(b)** BufferReplicator

Figure 1. Example Reo graphs

3 Rust

The Rust programming language was initiated at Mozilla and has, for instance, been used to reimplement the rendering code in Firefox. The syntax is similar to C++, but the memory management model is based on a linear type system (the Rust terminology is *ownership* and *borrowing*), without the aid of a garbage collector, as explained next.

- **Ownership** Rust gives strong guarantees about the relationship between values and variables: each value is assigned to a unique variable, called its *owner*. A value can be reassigned to a different variable, thereby *moving* it to a different owner, but the type system forbids further mutations or accesses of that value through the original variable after the move (statically checked). The memory occupied by a value is automatically freed whenever its owner goes out of scope.
- **Borrowing** Although every value has a unique owner, Rust's type system does allow other variables to temporarily *borrow* mutation or access rights to a value from the owner, without moving ownership, using references (akin to pointers and references in languages like C++ and Java). However, references are bound to rules: at any one time, either there is exactly one reference that allows mutation of the value, or there are zero or more references that allow (read) accesses.

Essentially, ownership and borrowing remove the root cause of data races, namely having a shared mutable state.

The ownership model enables the refinement of NEVER-COPY implementations, providing a third alternative: whereas in the existing NEVER-COPY implementations in Java, C, etc., a sender transfers *only* a reference, in our new implementation in Rust, a sender transfers *also* ownership during the transfer to the receiver. Thus, in the event that a sender must access or modify a value after sending it, it *must* create a copy before sending, or it will be in violation of the rules set out by Rust's type system. Such a violation is treated by the Rust compiler as a programming error and will cause the compiler to emit an error message instead of a binary executable.

4 Reo in Rust

We are currently developing a Rust implementation of Reo, a premier example of a supplemental language for protocols.

(a) Alternator **(b)** BufferReplicator

Figure 2. Example automata

Background. Reo [1] is a graphical language to *draw* protocols among participants as graphs. Figure 1 shows examples. To send a value, a participant can perform a blocking `put` operation on an *input vertex* of a graph (e.g., A or B in Figure 1a). Operation `put` initially *suspends* the participant: only once the graph is ready to accept the value, the `put` will complete, and the participant *resumes*. Similarly, to receive a value, a participant can perform a blocking `get` operation on an *output vertex* of a graph (e.g., C in Figure 1a). Once a graph has accepted a value through input vertices, it transports that value along its edges, possibly through one or more anonymous *internal vertices* (e.g., the middle vertex in Figure 1b), and dispenses it through one or more output vertices. Every edge has a *type* that determines its local transport behavior. The graphs in Figure 1 feature edges of three different types: a sync edge has synchronous channel semantics (e.g., the edge between A and C in Figure 1a); a fifo1 edge has asynchronous channel semantics, with an internal buffer of capacity 1 (e.g., the edge between B and C; the box signifies an internal buffer); a syncdrain edge has synchronous drain semantics (e.g., the edge between A and B in Figure 1a); other channel types appear in the literature [1].

Recent implementations of Reo are based on its operational semantics [2, 7]. The idea is to model the behavior of a Reo graph as a finite-state *automaton*, where states model configurations of the graph (e.g., buffer emptiness/fullness), and transitions model synchronous value transports along the edges. Figure 2 shows examples. Every transition label consists of two elements: the set of input and output vertices that collectively participate in the transition (e.g., $\{A, B, C\}$), called the *synchronization constraint*, and a specification that states how values are transported from input vertices to output vertices (e.g., [C := A; x := B]), called the *data constraint*. For instance, the bottom transition in Figure 2a states that a value accepted through A is dispensed through C, while synchronously, a value accepted through B is stored in local variable x (i.e., the internal buffer of the diagonal fifo edge in Figure 1a); the top transition states that the value previously stored in x is dispensed through C.

To compile a Reo graph to code in a base language, in its most basic form, the Reo compiler takes the following steps. First, the compiler determines for every constituent of the graph (i.e., vertices and edges) a "small automaton" that models the local transport behavior only of that constituent. Next, the compiler composes the small automata into one "large automaton", using a synchronous product operator; in this step, the compiler also abstracts away all internal vertices. Finally, the compiler translates the automaton to a piece of state machine code in the base language.[2]

`Automaton` An `Automaton` holds the complete set of `States` and the current `State`.

`State` A `State` consists of a list of `Transitions` and a label to help the Reo programmer to relate it to the graph.

`Transition` A `Transition` contains the associated synchronization and data constraints (i.e., its label) that is to be met in order to let the automaton fire it. Furthermore, the `Transition` contains the target `State`.

Figure 3. Core structs in generated code

Compilation to Rust. The final compilation step, when Rust is used as the base language, is implemented as a code generator. This generator takes a tuple representing an automaton as input. The output consists of a Rust application in source code form. This application can then be compiled into a binary executable, using the standard Rust toolchain.

Every participant is programmed as a sequential Rust function, executed in its own Rust thread. The moment the thread is started, the function is called and passed a generated `Automaton` struct,[3] which offers the following interface:

```
pub fn put(&mut self, vertex: usize, val: Message)
pub fn get(&mut self, vertex: usize) -> Message
```

Whenever `put` or `get` is called, the calling thread "enters" the generated protocol code, tries to find an *enabled* transition from the current state, and if one exits, actually *fires* that transition. A transition is enabled iff every vertex in its synchronization constraint has a pending `put` or `get`; a transition has fired iff values have been distributed according to its data constraint. Specifically, to distribute data, the generated code has separate variables to temporarily store values to be exchanged, namely one for every vertex and local variable controlled by the automaton (e.g., A, B, C, x for the automaton in Figure 2a); it simply transfers data from inputs to outputs *according to the data constraint*. For instance, [C := A; x := B] in Figure 2a is morally translated to:[4]

```
... // puts write to aut.val_A and aut.val_B
aut.val_C = aut.val_A; // move ownership
aut.val_x = aut.val_B; // move ownership
... // get reads from aut.val_C
```

If there are no enabled transitions (e.g., `put`s have been issued on vertices A and B, but a `get` has not been issued yet on vertex C in Figure 2a), the thread "leaves" the generated protocol code, and suspends; it resumes whenever another

[2]We omit a number of optimizations from this overview, which are essential to improve performance, but beyond our current scope.

[3]Specifically, `Arc<Mutex<Automaton>>`, to allow mutably sharing the same protocol among all participant functions.

[4]The assignments in the actual implementation are a bit more involved, to ensure the `aut` struct is left in a valid state (i.e., fields have values).

thread successfully fires a transition later on (e.g., after a get on C). A mutex ensures that only one thread can fire a transition at a time. Figure 3 summarizes the structs.

Resolving Q1a and Q1b. Both the put function and the get function make use of *pass-by-value* to pass in a message-to-send, or to retrieve a message-to-receive via the return value. In Rust, passing by value implies a transfer of ownership as the type system prescribes that a value can only be assigned to a unique variable. Indeed, after transferring ownership, the Rust compiler will emit a detailed error message if the participant tries to mutate or access the value. For instance:

```
error[E0382]: use of moved value: `v`
  --> producer_consumer.rs:25:30
   |
24 |  aut.put(self.port, v);
   |                     - value moved here
25 |  println!("{:?}", v);
   |                   ^ value used here after move
```

Thus, we statically guarantee that if a participant sends a value, it loses ownership (i.e., it cannot use that value in the future), and that if a participant receives a value, it gains ownership (i.e., no other participant can use that value in the future). The former resolves **Q1a**; the latter *almost* resolves **Q1b**, but special care is needed to support multi-casts.

The problem with multi-casts pertains to our translation of data constraints. Specifically, by assigning the value of one vertex to another, ownership of the value is transferred. This allows for a copy-free transport of the value, but because the type system guarantees that the value cannot be assigned to multiple variables, it does not work with multi-casts. This situation arises if a vertex in a Reo diagram is attached to multiple edges (e.g., the middle vertex in Figure 1b). In that case, the message value needs to be transported to all receiving vertices, which cannot be achieved by means of transferal of ownership. The solution is to make an explicit copy of the value and assign ownership of the copies.[5] For instance, [B := x; C := x] in Figure 2b is translated to:[4]

```
... // prev. transition wrote to aut.val_x
aut.val_B = aut.val_x.clone(); // explicit copy
aut.val_C = aut.val_x; // move ownership
... // get calls read from aut.val_B and aut.val_C
```

Our code generator recognizes such multiple assignments of a value and inserts the appropriate clone calls in the data constraint statements. Importantly, the copies are made internally by the generated code, transparent to the programmer. With this extra multi-cast care, **Q1b** is resolved as well.

Toward resolving Q2. We are currently setting up experiments to study the performance trade-offs between ALWAYS-COPY, NEVER-COPY, and NEVER-COPY+ownership. Our plan is to use the Rust code that is generated for Reo graphs as described above for NEVER-COPY+ownership, to simulate ALWAYS-COPY by adding additional copying to the generated code, and to simulate NEVER-COPY by always passing the same tiny value around (1 byte, so the costs of copying are negligible). In this way, we can compare the performance of the three approaches within the same framework.

We are planning two kinds of benchmarks. In *protocol benchmarks*, we aim to measure purely the overhead of copying for a representative set of Reo graphs, by running the generated code among "zealous" participants (i.e., participants that try to put/get as often as possible, without performing any real computations). In *whole-program benchmarks*, we aim to measure the effect of copying in real(istic) concurrent programs, such as pipelined computations where the same large data is processed by multiple threads in sequence.

[5] Such an explicit copy is created by requiring the message type to implement the Clone trait built-in defined in the Rust language.

5 Conclusion

We reported on our ongoing efforts toward the first shared memory implementation of a protocol programming language that guarantees freedom of data races, without excessive copying, by leveraging the programming language Rust and its type system. To this end, we briefly explained how protocol programming language Reo can be implemented in Rust to resolve **Q1a** and **Q1b**, and we outlined our plans to resolve **Q2**. Other future work includes:

- *Improve static guarantees.* We are curious to study how, and to what extent, Rust's type system can also be leveraged to implement linear *session type* systems [4] (i.e., Scribble's theoretical foundation).
- *Optimize.* The existing Java implementation of Reo has optimizations that have not been implemented yet in our Rust implementation (e.g., the Java implementation parallelizes execution of the generated code).
- *Relaxations.* In practice, it may be desirable to relax the model to allow senders to keep a read-only reference to sent data (e.g., to improve performance). We are interested to investigate how to balance this relaxed setting with freedom of data races.

References

[1] Farhad Arbab. 2004. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14, 3 (2004), 329–366.
[2] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. 2006. Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.* 61, 2 (2006), 75–113.
[3] Gartner. 2017. Hype Cycle for Embedded Software and Systems. (2017).
[4] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.
[5] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (Lecture Notes in Computer Science)*, Vol. 10202. Springer, 116–133.
[6] Sung-Shik T. Q. Jongmans and Farhad Arbab. 2016. PrDK: Protocol Programming with Automata. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9636. Springer, 547–552.
[7] Sung-Shik T. Q. Jongmans, Tobias Kappé, and Farhad Arbab. 2017. Constraint automata with memory cells and their composition. *Sci. Comput.*

Program. 146 (2017), 50–86.

[8] Natallia Kokash, Christian Krause, and Erik P. de Vink. 2012. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *Formal Asp. Comput.* 24, 2 (2012), 187–216.

[9] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *TGC (Lecture Notes in Computer Science)*, Vol. 8358. Springer, 22–41.