Prophecy-Based Automated Verification of Message-Passing Programs

Takashi Nagatomi
The University of Tokyo, Japan

Musashi Katsura
The University of Tokyo, Japan

Naoki Kobayashi
The University of Tokyo, Japan

Yusuke Matsushita
Kyoto University, Japan
MPI-SWS, Kaiserslautern, Germany

Ken Sakayori
The University of Tokyo, Japan

Abstract

We propose a fully automated method for verifying functional correctness of message-passing concurrent programs by reducing verification problems to constrained Horn clause (CHC) solving. Inspired by RustHorn's prophecy-based technique, we represent each sender channel by a list of values to be sent over the channel in the future, which enables modular encoding of sender and receiver threads in CHCs. To capture causal dependencies between different channels, we further attach timestamps to messages. We prove that the resulting reduction is sound and complete: a program is free from assertion failures if and only if the corresponding system of CHCs is satisfiable. We have also implemented a prototype verifier for Rust-like programs and experimentally confirmed the effectiveness of the approach.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Program verification, message-passing concurrent programs, constrained Horn clauses, prophecies

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2026.43

Related Version *Full Version*: https://doi.org/10.48550/arXiv.2606.28066 [31]

Funding *Naoki Kobayashi*: JSPS KAKENHI Grant Numbers JP20H05703, JP26H02486.
Yusuke Matsushita: Hakubi Project at Kyoto University, JSPS KAKENHI Grant Number JP24KJ0133.
Ken Sakayori: JSPS KAKENHI Grant Numbers JP24K20731, JP26H02486.

1 Introduction

We propose a fully automated method for verifying functional correctness of message-passing concurrent programs. Fully automated verification of such programs is challenging because their behavior is non-deterministic and depends on complex interactions through message exchanges. Although type-based approaches to automated analysis and verification of message-passing programs have been proposed [20, 16], they are usually conservative and mainly focus on abstract properties such as deadlock freedom, information-flow security, and protocol conformance. In contrast, automatically verifying functional correctness properties such as "the number of messages received is n" and "the result of the whole computation is $2n$" (where n may be a program parameter) remains challenging, despite recent progress such as the refinement-session-type-based approach of Ueno and Das [33].

```
fn msg_count(n: usize) {
    let (s, r) = channel();
    for i in 0..n { // spawn n senders
        let s_clone = s.clone();
        thread::spawn(move || {
            s_clone.send(1).unwrap(); // each thread sends 1 to the channel
        });
    }
    let mut c = 0; // number of messages received
    loop {  // repeatedly receive and count messages
        let _v = r.recv().unwrap();
        c += 1;  // increments the message counter
        assert!(c <= n); // error if more than n messages are received
    }
}
```

Figure 1 A Rust-style message-passing concurrent program.

Inspired by RustHorn's prophecy-based technique [30] for reducing verification of sequential programs with mutable references to CHC solving, that is, satisfiability checking of constrained Horn clauses, we propose a prophecy-based reduction from verification of message-passing concurrent programs to CHC solving [5, 10].

To illustrate the idea, consider the Rust-style concurrent program in Figure 1. The function `msg_count` takes a natural number n and creates a sender channel `s` and a corresponding receiver channel `r`. It then spawns n threads, each of which sends 1 through the channel. It next repeatedly receives messages, counts them, and asserts that the number of messages received so far is at most n. Suppose we wish to verify that the assertion never fails. We convert this problem to the satisfiability problem for the following constrained Horn clauses (CHCs).

$$\bot \Leftarrow \mathit{Entry}(\texttt{true}, n) \wedge n \geq 0.$$
$$\mathit{Entry}(b, n) \Leftarrow \mathit{ForLoop}(b, 0, n, l, l).$$
$$\mathit{ForLoop}(b, n, n, [\,], r) \Leftarrow \mathit{Loop}(b, 0, n, r).$$
$$\mathit{ForLoop}(b_1 \vee b_2, i, n, s, r) \Leftarrow i < n \wedge \texttt{Merge}(s_1, s_2, s) \wedge \mathit{Send}(b_2, s_2) \wedge \mathit{ForLoop}(b_1, i+1, n, s_1, r).$$
$$\mathit{Send}(\texttt{false}, s) \Leftarrow s = [1].$$
$$\mathit{Loop}(b, c, n, r) \Leftarrow r = v :: r' \wedge c + 1 \leq n \wedge \mathit{Loop}(b, c+1, n, r').$$
$$\mathit{Loop}(\texttt{true}, c, n, r) \Leftarrow r = v :: r' \wedge c + 1 > n.$$

The system of CHCs above is constructed so that it is satisfiable, i.e., there exists an interpretation of the predicates that makes all the clauses valid, if and only if, for every non-negative integer n, `msg_count`(n) never causes an assertion failure. Thus, the verification problem can be reduced to CHC satisfiability and handled by a CHC solver [25, 15, 7, 19].

Intuitively, the predicate $\mathit{Entry}(b, n)$ means that a call to `msg_count`(n) may terminate, with $b = \texttt{true}$ indicating that the call may terminate with an assertion failure. Thus, the first clause states that a call to `msg_count`(n) never terminates with an assertion failure. The predicate $\mathit{ForLoop}(b, i, n, s, r)$ means that, from the program state at the beginning of the `for`-loop with loop index i, the remaining execution may terminate, with b indicating

the error status. Similarly, the predicate $\mathit{Loop}(b, c, n, r)$ means that, from the program state at the beginning of the second loop with counter value c, the remaining execution may terminate, with b indicating the error status.

The main issue is how to represent channels in CHCs. Inspired by RustHorn's prophecy-based technique, we represent a sender channel s as a list of values to be sent *in the future*, and a receiver channel r as a list of values that may be received *in the future*. Thus, we model the behavior of the program as follows. When creating a channel, the program *guesses* a list l of values to be sent through the channel. The second clause reflects this behavior: it says that if the execution from the beginning of the `for`-loop may terminate with error flag b by exchanging a list l of values through the channel, then $\texttt{msg\_count}(n)$ may terminate with error flag b. Cloning of the sender channel is represented by the list-merge relation $\texttt{Merge}(s_1, s_2, s)$ in the fourth clause, which means that the list s is obtained by merging s_1 and s_2 in some order. This models the creation of a clone of channel s: the two resulting sender channels are represented by s_1 and s_2, and the total list of values to be sent is obtained by merging them. The fifth clause, $\mathit{Send}(\texttt{false}, s) \Leftarrow s = [1]$, models the send operation. The equation $s = [1]$ ensures that the prophecy s, representing the list of values to be sent, matches the actual list of messages being sent, namely $[1]$. The last two clauses, in contrast, model the receive operation in the program. Receiving a value v from a receiver channel r is modeled by the equation $r = v :: r'$, which assumes that v is at the head of the receiver queue. Note that there is no clause for the case where r is an empty list; this means that no message is available, and hence the execution is blocked.

The crux of the prophecy-based representation of channels is that it enables modular modeling of sender and receiver threads: it suffices that the sender and the receiver agree, when a channel is created (as in the second clause above), on the values to be sent, and afterwards, the sender locally checks at each send operation that the prophecy is correct, while the receiver locally assumes that messages arrive as described by the prophecy. One may be tempted to model a channel by the current contents of its message queue, rather than by the list of values to be sent in the future. In that case, however, the message queue would have to be treated as global state: since it is updated by both send and receive operations, sender and receiver threads could no longer be modeled as separate predicates in the resulting CHCs.

The idea above is sufficient to ensure soundness: if the corresponding system of CHCs is satisfiable, then the program does not cause any assertion failure. However, it is not sufficient for completeness. This is because the prophecy-based representation of channels does not capture the causal dependency between different channels. To address this issue, we attach a *timestamp* to each value to be sent and represent a channel as a list of pairs consisting of a timestamp and a value: for example, $[(1, v_1); (3, v_2)]$ means that the value v_1 is to be sent at time 1, and v_2 is to be sent at time 3. We show that the reduction to CHCs extended with timestamps is sound and complete: a program does not cause any assertion failure if and only if the corresponding system of CHCs is satisfiable.

The contributions of this paper are summarized as follows.

- Prophecy- and timestamp-based reduction from the safety verification of message-passing concurrent programs to CHC solving. We prove that the reduction is sound and complete. We also discuss a few variations of the reduction, which capture subtle variations of message-passing semantics.
- Implementation and experiments. We have implemented a prototype verification tool for Rust-like programs based on the above idea, and conducted experiments to evaluate its effectiveness. While the experiments confirm the effectiveness of our method, they also suggest the need for further improvement of the backend CHC solvers, which would also benefit the community of CHC-based verification.

The rest of this paper is structured as follows. Section 2 introduces a language with message-passing concurrency. Section 3 formalizes our reduction from the safety verification of message-passing concurrent programs to CHC solving, and Section 4 reports experimental results. Section 5.1 discusses the flexibility of our approach under variations of asynchronous message-passing semantics. Section 6 discusses related work, and Section 7 concludes.

2 Target Language

This section defines the target language, a message-passing concurrent language augmented with first-order recursive functions. The communication channels obey the multi-producer, single-consumer (MPSC) principle as in Rust.

Notation. We write $\widetilde{a}$ to denote a sequence $a_1, \ldots, a_n$. This notation is also extended to binary relations in a pointwise manner. For example, we write $\widetilde{x} : \widetilde{\tau}$ for a sequence of type bindings $x_1 : \tau_1, \ldots, x_n : \tau_n$.

2.1 Syntax and Types

Syntax. In our target language, a program consists of function definitions, each of which only performs a single "instruction". Control flow is encoded as function calls, and every function ends with a call to a continuation function. This design is chosen to simplify the translation to CHCs.

The syntax is given as follows:

$$
\begin{aligned}
\text{(statements)}\ M &::= \texttt{fail} \mid () \mid f(\widetilde{a}) \mid \texttt{if}\ a\ \texttt{then}\ f_1(\widetilde{a})\ \texttt{else}\ f_2(\widetilde{a}) \\
&\quad \mid \texttt{send}\ a\ \texttt{to}\ x; f(\widetilde{a}) \mid\ \texttt{let}\ x = \texttt{recv}\ y\ \texttt{in}\ f(\widetilde{a}) \mid \texttt{new}\ x, y\ \texttt{in}\ f(\widetilde{a}) \\
&\quad \mid \texttt{let}\ x = \texttt{clone}\ y\ \texttt{in}\ f(\widetilde{a}) \mid \texttt{spawn}(f_1(\widetilde{a_1})); f_2(\widetilde{a_2}) \\
\text{(arithmetic exp.)}\ a &::= n \mid x \mid a_1\ \texttt{op}\ a_2 \\
\text{(function defs.)}\ D &::= \{\, f_1(\widetilde{x_1} : \widetilde{b_1}) \ = \ M_1, \ldots, f_n(\widetilde{x_n} : \widetilde{b_n}) \ = \ M_n \,\} \\
\text{(program)}\ P &::= (D, f(\widetilde{a})) \\
\text{(argument types)}\ b &::= \texttt{int} \mid \texttt{ch}^- \mid \texttt{ch}^+ \\
\text{(types)}\ \tau &::= b \mid \star \mid (b, \ldots, b) \to \star
\end{aligned}
$$

The definition of *arithmetic expressions* and the constructs in the first line should be mostly self-explanatory. We briefly note a few points. Here `op` is a meta-variable for binary integer operators. Comparison and logical operators, such as $=$ or $\wedge$, are also regarded as integer operators by regarding $\{0, 1\}$ as Boolean values, and we may write `true` and `false` in place of 1 and 0, respectively. The statement `fail` represents abnormal program termination. The second line shows basic channel-related operations: sending the value of a via the channel x, receiving an integer from the channel y and channel creation. Channel creation $\texttt{new}\ x, y\ \texttt{in}\ f(\widetilde{a})$ creates the two endpoints of the channel, where x is the sender channel and y is the corresponding receiver channel, and proceeds by calling f. The operator `clone` duplicates (i.e. creates an alias of) a sender channel. The statement $\texttt{spawn}(f_1(\widetilde{a_1})); f_2(\widetilde{a_2})$ creates a new thread to execute $f_1(\widetilde{a_1})$, while the current thread continues with $f_2(\widetilde{a_2})$. A *program* is a pair of function definitions and the main function. The type $\star$ denotes the unit type; note that the return type of a function is restricted to the unit type. The types $\texttt{ch}^-$ and $\texttt{ch}^+$ represent types of receiver and sender channels, respectively.

$$\frac{\mathtt{nonlin}(\Gamma)}{\Gamma, x : b \vdash x : b} \qquad \frac{\mathtt{nonlin}(\Gamma)}{\Gamma \vdash n : \mathtt{int}}$$

$$\frac{\Gamma \vdash a_1 : \mathtt{int} \qquad \Gamma \vdash a_2 : \mathtt{int}}{\Gamma \vdash a_1 \ \mathtt{op} \ a_2 : \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{fail} : \star} \qquad \frac{}{\Gamma \vdash () : \star}$$

$$\frac{\Gamma_0(f) = (b_1, \ldots, b_n) \to \star \qquad \mathtt{nonlin}(\Gamma_0) \qquad \Gamma_i \vdash a_i : b_i \ (\text{for each i} \in \{1, \ldots, n\})}{\Gamma_0 + \Gamma_1 + \ldots + \Gamma_n \vdash f(a_1, \ldots, a_n) : \star}$$

$$\frac{\Gamma_0 \vdash a : \mathtt{int} \qquad \Gamma \vdash f_i(\widetilde{a}_i) : \star \ (\text{for } i = 1, 2)}{\Gamma_0 + \Gamma \vdash \mathtt{if} \ a \ \mathtt{then} \ f_1(\widetilde{a}_1) \ \mathtt{else} \ f_2(\widetilde{a}_2) : \star}$$

$$\frac{\Gamma_i \vdash f_i(\widetilde{a}_i) : \star \ (\text{for } i = 1, 2)}{\Gamma_1 + \Gamma_2 \vdash \mathtt{spawn}(f_1(\widetilde{a}_1)); f_2(\widetilde{a}_2) : \star} \qquad \frac{\Gamma + x : \mathtt{ch}^+ + y : \mathtt{ch}^- \vdash f(\widetilde{a}) : \star}{\Gamma \vdash \mathtt{new} \ x, y \ \mathtt{in} \ f(\widetilde{a}) : \star}$$

$$\frac{\Gamma_0 \vdash a : \mathtt{int} \qquad \Gamma(x) = \mathtt{ch}^+ \qquad \Gamma \vdash f(\widetilde{a}) : \star}{\Gamma_0 + \Gamma \vdash \mathtt{send} \ a \ \mathtt{to} \ x; f(\widetilde{a}) : \star} \qquad \frac{\Gamma(y) = \mathtt{ch}^- \qquad \Gamma, x : \mathtt{int} \vdash f(\widetilde{a}) : \star}{\Gamma \vdash \mathtt{let} \ x = \mathtt{recv} \ y \ \mathtt{in} \ f(\widetilde{a}) : \star}$$

$$\frac{\Gamma(y) = \mathtt{ch}^+ \qquad \Gamma + x : \mathtt{ch}^+ \vdash f(\widetilde{a}) : \star}{\Gamma \vdash \mathtt{let} \ x = \mathtt{clone} \ y \ \mathtt{in} \ f(\widetilde{a}) : \star}$$

$$\frac{\Gamma = \{ f_1 : (\widetilde{b}_1) \to \star, \ldots, f_n : (\widetilde{b}_n) \to \star \} \qquad \Gamma, \widetilde{x}_i : \widetilde{b}_i \vdash M_i : \star \ (\text{for each } i \in \{1, \ldots, n\})}{\vdash \{ f_i(\widetilde{x}_i : \widetilde{b}_i) \mid i = 1, \ldots, n \} : \Gamma}$$

$$\frac{\vdash D : \Gamma \qquad \Gamma \vdash f(\widetilde{a}) : \star}{\vdash (D, f(\widetilde{a})) : \star}$$

Figure 2 Typing rules.

▶ **Example 1.** The program in the introduction can be written in our target language as $(D, \mathit{msg\_count}(n))$, where the function definitions in D are given as follows:

$$\mathit{msg\_count}(n : \mathtt{int}) = \mathtt{new} \ s, r \ \mathtt{in} \ \mathit{for}(0, n, s, r)$$
$$\mathit{for}(i : \mathtt{int}, n : \mathtt{int}, s : \mathtt{ch}^+, r : \mathtt{ch}^-) = \mathtt{if} \ i < n \ \mathtt{then} \ f_4(i, n, s, r) \ \mathtt{else} \ \mathit{loop}(0, n, r)$$
$$f_4(i : \mathtt{int}, n : \mathtt{int}, s : \mathtt{ch}^+, r : \mathtt{ch}^-) = \mathtt{let} \ s' = \mathtt{clone} \ s \ \mathtt{in} \ f_5(i, n, s, r, s')$$
$$f_5(i : \mathtt{int}, n : \mathtt{int}, s : \mathtt{ch}^+, r : \mathtt{ch}^-, s' : \mathtt{ch}^+) = \mathtt{spawn}(f_6(s')); \mathit{for}(i + 1, n, s, r)$$
$$\mathit{loop}(c : \mathtt{int}, n : \mathtt{int}, r : \mathtt{ch}^-) = \mathtt{let} \ \_v = \mathtt{recv} \ r \ \mathtt{in} \ \mathit{assert}(c + 1, n, r)$$
$$\mathit{assert}(c : \mathtt{int}, n : \mathtt{int}, r : \mathtt{ch}^-) = \ \mathtt{if} \ c \leq n \ \mathtt{then} \ \mathit{loop}(c, n, r) \ \mathtt{else} \ \mathit{fail}(c, n, r)$$
$$\mathit{fail}(c : \mathtt{int}, n : \mathtt{int}, r : \mathtt{ch}^-) = \mathtt{fail}$$
$$f_6(s : \mathtt{ch}^+) = \mathtt{send} \ 1 \ \mathtt{to} \ s; ()$$

For each program point, we introduce a function parameterized by the variables live at that point. Roughly speaking, f_i corresponds to line i of Figure 1. The function $\mathit{assert}()$ encodes the assertion by branching to `fail` when the assertion condition is violated. ⌟

Typing. We consider only well-typed programs, as defined by the typing rules in Figure 2. A *type environment*, written $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, is a finite map from variables to types. We treat channel types as linear types, and say that Γ is non-linear if it contains no bindings of the form $x : \mathtt{ch}^-$ or $x : \mathtt{ch}^+$. We write $\Gamma_1 + \Gamma_2$ for the union of the two environments, which is only defined if the linear parts of the domains of Γ_1 and Γ_2 are disjoint. The typing rules are fairly standard, except for the linear (or affine, more precisely) treatment of channel types: for example, the statement $\mathtt{spawn}(f_1(x)); f_2(x)$ is not allowed if x has a channel type. Weakening of a variable with a channel type is only allowed in the rules for $()$ and `fail`.

▶ Remark 2. Example 1 slightly abuses the syntax: some functions drop channels even though their bodies are neither $()$ nor `fail`. To be more precise, we could spawn a function $\mathit{drop}(\widetilde{x}) = ()$ to discard channels; however, we omit such explicit drop functions throughout the paper for readability. ⌟

$$\frac{f(\widetilde{x}) = \texttt{fail} \in D}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D \texttt{fail}} \quad \text{(R-Fail)} \qquad \frac{f(\widetilde{x}) = () \in D}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T}, \mathcal{Q})} \quad \text{(R-End)}$$

$$\frac{f(\widetilde{x}) = g(\widetilde{a}') \in D}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g([\widetilde{a}/\widetilde{x}]\widetilde{a}')\}, \mathcal{Q})} \quad \text{(R-Func)}$$

$$\frac{f(\widetilde{x}) = \texttt{if } a_0 \texttt{ then } g_1(\widetilde{a}_1) \texttt{ else } g_2(\widetilde{a}_2) \in D \qquad [\widetilde{a}/\widetilde{x}]a_0 \Downarrow n \qquad n \neq 0}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g_1([\widetilde{a}/\widetilde{x}]\widetilde{a}_1)\}, \mathcal{Q})} \quad \text{(R-IfT)}$$

$$\frac{f(\widetilde{x}) = \texttt{if } a_0 \texttt{ then } g_1(\widetilde{a}_1) \texttt{ else } g_2(\widetilde{a}_2) \in D \qquad [\widetilde{a}/\widetilde{x}]a_0 \Downarrow 0}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g_2([\widetilde{a}/\widetilde{x}]\widetilde{a}_2)\}, \mathcal{Q})} \quad \text{(R-IfF)}$$

$$\frac{f(\widetilde{x}) = \texttt{send } a_0 \texttt{ to } y; g(\widetilde{a}') \in D \qquad \mathcal{Q}(r) = (S, \ell) \qquad [\widetilde{a}/\widetilde{x}]y \in S \qquad [\widetilde{a}/\widetilde{x}]a_0 \Downarrow n}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g([\widetilde{a}/\widetilde{x}]\widetilde{a}')\}, \mathcal{Q}[r \mapsto (S, \ell \cdot n)])} \quad \text{(R-Send)}$$

$$\frac{f(\widetilde{x}) = \texttt{let } y = \texttt{recv } z \texttt{ in } g(\widetilde{a}') \in D \qquad r = [\widetilde{a}/\widetilde{x}]z \qquad \mathcal{Q}(r) = (S, n \cdot \ell)}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g([n/y, \widetilde{a}/\widetilde{x}]\widetilde{a}')\}, \mathcal{Q}[r \mapsto (S, \ell)])} \quad \text{(R-Recv)}$$

$$\frac{f(\widetilde{x}) = \texttt{new } y, z \texttt{ in } g(\widetilde{a}') \in D \qquad s, r \text{ fresh}}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g([s/y, r/z, \widetilde{a}/\widetilde{x}]\widetilde{a}')\}, \mathcal{Q}[r \mapsto (\{\, s \,\}, \varepsilon)])} \quad \text{(R-New)}$$

$$\frac{f(\widetilde{x}) = \texttt{let } y = \texttt{clone } z \texttt{ in } g(\widetilde{a}') \in D \quad \mathcal{Q}(r) = (S, \ell) \quad [\widetilde{a}/\widetilde{x}]z \in S \quad s \text{ fresh}}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g([s/y, \widetilde{a}/\widetilde{x}]\widetilde{a}')\}, \mathcal{Q}[r \mapsto (S \cup \{\, s \,\}, \ell)]} \quad \text{(R-Clone)}$$

$$\frac{f(\widetilde{x}) = \texttt{spawn}(g_1(\widetilde{a}_1)); g_2(\widetilde{a}_2) \in D \qquad j \text{ fresh}}{(\mathcal{T} \uplus \{f(\widetilde{a})\}, \mathcal{Q}) \Longrightarrow_D (\mathcal{T} \uplus \{g_1([\widetilde{a}/\widetilde{x}]\widetilde{a}_1), g_2([\widetilde{a}/\widetilde{x}]\widetilde{a}_2)\}, \mathcal{Q})} \quad \text{(R-Spawn)}$$

Figure 3 Operational semantics.

2.2 Operational Semantics

The operational semantics is defined by the transition relation $C \Longrightarrow_D C'$ on configurations, where a configuration C is either `fail`, representing failure, or a pair $(\mathcal{T}, \mathcal{Q})$ consisting of a *thread pool* and a *channel context*. A thread pool $\mathcal{T}$ is a finite multiset of function calls of the form $f(\widetilde{a})$. A channel context $\mathcal{Q} \in \mathit{Var} \rightharpoonup 2^{\mathit{Var}} \times \mathbb{Z}^*$ maps each receiver channel name (which is represented as a variable) to a pair consisting of the set of corresponding sender channel names and a sequence of integers representing the message queue.

The transition relation $C \Longrightarrow_D C'$ is defined by the rules shown in Figure 3. In the figure, $[\widetilde{a}/\widetilde{x}]a'$ denotes the expression obtained from a' by substituting $\widetilde{a}$ for $\widetilde{x}$, and $a \Downarrow v$ denotes the big-step evaluation relation for simple expressions; its definition is standard and thus omitted. We briefly explain some important rules. The rule R-Send appends the sent value to the end of the corresponding message queue in the channel context. This reflects the non-blocking nature of asynchronous communication, where a sender does not wait for the receiver. The rule R-Recv dequeues a value from the corresponding message queue in the channel context and passes it to the continuation call. Note that the message queue must be non-empty for R-Recv to apply. The rule R-Clone creates a fresh variable s as an alias for the sender channel $[\widetilde{a}/\widetilde{x}]z$ and relates it to the receiver channel associated with $[\widetilde{a}/\widetilde{x}]z$.

3 Reduction to CHC Solving

As outlined in the introduction, we reduce the problem of verifying that a program never reaches `fail` to CHC satisfiability. This section first defines the class of CHCs used in the reduction and then describes the translation from the target language to CHCs.

3.1 Preliminary: CHCs

We briefly review constrained Horn clauses (CHCs) with lists and pairs. The class of CHCs used in this paper is a subclass of CHCs over the combined theory of integer arithmetic and algebraic data types (ADTs).

The sets of *sorts*, *terms*, and *constrained formulas* are defined as follows:

$$\begin{aligned}
\text{(sorts)}\ \beta &::= \mathtt{int} \mid \mathtt{int} \times \mathtt{int} \mid \mathtt{list\ int} \mid \mathtt{list}\ (\mathtt{int} \times \mathtt{int}) \\
\text{(terms)}\ s &::= x \mid n \mid s_1\ \mathtt{op}\ s_2 \mid [\,] \mid s_1 :: s_2 \mid (s_1, s_2) \\
\text{(constraints)}\ \theta &::= \top \mid \bot \mid s_1 = s_2 \mid s_1 < s_2 \mid \theta \vee \theta \mid \theta \wedge \theta \mid \neg\theta
\end{aligned}$$

Here, the metavariable $\mathtt{op}$ ranges over the same binary arithmetic operations as those used in the definition of the target language. As usual, $[\,]$ is the nil list, $s_1 :: s_2$ is the cons operation, and (s_1, s_2) is the pairing. The predicate $<$ is used for integer comparison, and the predicate $=$ is used for equality between terms of the same sort.

A *constrained Horn clause* (CHC) Φ is a formula of the form

$$\forall \widetilde{x} : \widetilde{\beta}.\ \mathcal{H} \Leftarrow P_1(\widetilde{s_1}) \wedge \ldots \wedge P_n(\widetilde{s_n}) \wedge \theta$$

where $\mathcal{H}$, called the *head*, is either $\bot$ or $P_0(\widetilde{s_0})$. Each P_i is a *predicate variable* that has an associated type; we write $P : (\beta_1, \ldots, \beta_n)$ if P is an n-ary predicate whose i-th argument is of sort β_i. We stipulate that all the CHCs are well-sorted and closed, i.e. $\widetilde{x}$ covers all the free variables appearing in $\widetilde{s_i}$ and θ. For readability, we may omit the universal quantifiers of a CHC.

A *system of CHCs* is a pair $(\mathbf{\Phi}, \Delta)$ where $\mathbf{\Phi} = \{\Phi_1, \ldots, \Phi_m\}$ is a finite set of CHCs, regarded as their conjunction, and $\Delta = \{P_1, \ldots, P_n\}$ is the set of predicate variables appearing in $\mathbf{\Phi}$. We often write $\mathbf{\Phi}$ for $(\mathbf{\Phi}, \Delta)$ when Δ is clear from the context. A *solution* of $(\mathbf{\Phi}, \{P_1, \ldots, P_n\})$ is an interpretation for $\{P_1, \ldots, P_n\}$ under which $\mathbf{\Phi}$ becomes true. If such a solution exists, we say that $\mathbf{\Phi}$ is *satisfiable*.

▶ **Example 3.** Consider a system of CHCs $(\mathbf{\Phi}, \{\mathit{ForLoop} : (\mathtt{int}, \mathtt{int}, \mathtt{int}, \mathtt{list\ int}, \mathtt{list\ int}),\ \mathit{Loop} : (\mathtt{int}, \mathtt{int}, \mathtt{int}, \mathtt{list\ int})\})$, where $\mathbf{\Phi}$ consists of the following clauses.

$$\bot \Leftarrow \mathit{ForLoop}(\mathtt{true}, 0, n, l, l) \wedge n \geq 0. \qquad \mathit{ForLoop}(b, n, n, [\,], r) \Leftarrow \mathit{Loop}(b, 0, n, r).$$
$$\mathit{ForLoop}(b, i, n, 1 :: s_1, r) \Leftarrow \mathit{ForLoop}(b, i+1, n, s_1, r) \wedge i < n.$$
$$\mathit{Loop}(b, c, n, r) \Leftarrow \mathit{Loop}(b, c+1, n, r') \wedge r = v :: r' \wedge c + 1 \leq n.$$
$$\mathit{Loop}(\mathtt{true}, c, n, r) \Leftarrow r = v :: r' \wedge c + 1 > n.$$

This is a simplified version of the CHCs in Section 1. It is satisfiable, with the following interpretation as a witness:

$$\{\mathit{ForLoop} \mapsto \{(\mathtt{true}, i, n, s, r) \mid (0 \leq i \leq n \wedge |r| > n \wedge |s| = n - i) \vee n < 0\},$$
$$\mathit{Loop} \mapsto \{(\mathtt{true}, c, n, r) \mid c + |r| > n\}\}.$$

Here, $|r|$ denotes the length of list r. To see that the interpretation above satisfies the first clause, notice that $\mathit{ForLoop}(\mathtt{true}, 0, n, l, l)$ is equivalent to $(0 \leq 0 \leq n \wedge |l| > n \wedge |l| = n - 0) \vee n < 0$ under the interpretation, which can be simplified to $n < 0$.

3.2 Translation from Target Language to CHCs

Now we formalize the translation into CHCs. We first define the translation of types, then describe how programs are translated into systems of CHCs, and finally give the translation rules for individual statements.

Translation of types. The encoding $(\!|-|\!)$ from argument types to sorts is defined by

$$(\!|\mathtt{int}|\!) \stackrel{\text{def}}{=} \mathtt{int} \qquad (\!|\mathtt{ch}^-|\!) \stackrel{\text{def}}{=} \mathtt{list}\ (\mathtt{int} \times \mathtt{int}) \qquad (\!|\mathtt{ch}^+|\!) \stackrel{\text{def}}{=} \mathtt{list}\ (\mathtt{int} \times \mathtt{int}).$$

Channel types are translated into lists of pairs because each channel is represented as a list of timestamped future values to be received or sent on the channel, as explained in the introduction. Functions are translated into predicates. Accordingly, function types are translated as

$$(\!|(b_1, \ldots, b_n) \to \star|\!) \stackrel{\text{def}}{=} (\mathtt{int}, \mathtt{int}, (\!|b_1|\!), \ldots, (\!|b_n|\!)).$$

Note that two additional integer parameters are inserted during the translation. The first integer argument will be used to track whether a program failure has occurred, and the second will be used to pass timing information, as we shall see below.

Translation of programs. Now we explain how a well-typed program $(D, f(\widetilde{a}))$ is translated into a system of CHCs. The core idea is to introduce a predicate $F_i : (\!|\widetilde{b_i} \to \star|\!)$ for every function $f_i : \widetilde{b_i} \to \star$ in D, that roughly expresses the following property:

> $F_i(b, t, \widetilde{x}_{\mathtt{int}}, \widetilde{x}_{\mathtt{c}})$ holds if a call to the function f_i at time t with integer values $\widetilde{x}_{\mathtt{int}}$ and channels $\widetilde{x}_{\mathtt{c}}$[1] satisfies the following conditions.
>
> - it behaves as prophesied by the lists $\widetilde{x}_{\mathtt{c}}$, that is, the channels send (or receive) the values in the list in the order they appear, following their declared timestamps; and
> - the flag b records whether the execution eventually reaches `fail`.

Here, $b = \mathtt{true}$ means that `fail` is reachable, whereas $b = \mathtt{false}$ covers all other cases, including normal termination, divergence, and blocking while waiting for a message.

Based on this idea, we translate each function definition $f_i(\widetilde{x}_i) = M_i$ into a clause $F_i(b, t, \widetilde{x}_i) \Leftarrow (\!|M_i|\!)^b_t$.[2] Here we first discuss the two simplest cases, $M = \mathtt{fail}$ and $M = ()$, deferring the full translation rules to later in this subsection. If $f_i(\widetilde{x}_i) = \mathtt{fail} \in D$, then we add the clause

$$F_i(b, t, \widetilde{x}_i) \Leftarrow b = \mathtt{true} \wedge \widetilde{s}_i = \widetilde{[]}$$

where $\widetilde{s}_i$ are variables of type $\mathtt{ch}^+$ in $\widetilde{x}_i$. (We shall write $\widetilde{s}_i = \mathit{OChan}(\widetilde{x}_i)$ to denote this condition in the sequel.) Since the execution of f_i can reach `fail`, the failure flag must be `true`. Moreover, since no further send operations will be executed, the prophecies for output channels must be empty lists. In contrast, we impose no such restriction on input channels, because their prophecies represent messages that are or will become available on the channels, rather than messages that are actually received by this execution. Similarly, if $f_i(\widetilde{x}_i) = () \in D$, then we add the clause

$$F_i(b, t, \widetilde{x}_i) \Leftarrow b = \mathtt{false} \wedge \widetilde{s}_i = \widetilde{[]} \qquad \text{where } \widetilde{s}_i = \mathit{OChan}(\widetilde{x}_i).$$

The difference is that the failure flag b is set to `false` because () means that a thread terminated without any error.

---

[1] Here, we assume that integer arguments precede list arguments, although this is not the case in general.

[2] Strictly speaking, $(\!|M_i|\!)^b_t$ may be a disjunction $\varphi_{i,1} \vee \ldots \vee \varphi_{i,m_i}$. In that case, $F_i(b, t, \widetilde{x}_i) \Leftarrow (\!|M_i|\!)^b_t$ is split into clauses $F_i(b, t, \widetilde{x}_i) \Leftarrow \varphi_{i,j}$.

$$(\!|\texttt{send}\ a\ \texttt{to}\ x; f(\widetilde{a})|\!)^b_t \stackrel{\text{def}}{=} \exists t_s\, x'.F(b,t_s,[x'/x]\widetilde{a}) \wedge x = (t_s, a) :: x' \wedge t \leq t_s$$

$$(\!|\texttt{let}\ x = \texttt{recv}\ y\ \texttt{in}\ f(\widetilde{a})|\!)^b_t \stackrel{\text{def}}{=} \exists t_r\, x\, y'.F(b,t_r,[y'/y]\widetilde{a}) \wedge y = (t_s, x) :: y' \wedge t_s < t_r \wedge t \leq t_r$$

$$(\!|\texttt{if}\ a\ \texttt{then}\ f_1(\widetilde{a}_1)\ \texttt{else}\ f_2(\widetilde{a}_2)|\!)^b_t \stackrel{\text{def}}{=} (a \neq 0 \wedge F_1(b,t,\widetilde{a}_1)) \vee (a = 0 \wedge F_2(b,t,\widetilde{a}_2))$$

$$(\!|\texttt{spawn}(f_1(\widetilde{a}_1)); f_2(\widetilde{a}_2)|\!)^b_t \stackrel{\text{def}}{=} \exists b_1\, b_2.F_1(b_1,t,\widetilde{a}_1) \wedge F_2(b_2,t,\widetilde{a}_2) \wedge b = (b_1 \vee b_2)$$

$$(\!|\texttt{new}\ x, y\ \texttt{in}\ f(\widetilde{a})|\!)^b_t \stackrel{\text{def}}{=} \exists x\, y.F(b,t,\widetilde{a}) \wedge x = y \wedge \texttt{Sorted}(x)$$

$$(\!|\texttt{let}\ x = \texttt{clone}\ y\ \texttt{in}\ f(\widetilde{a})|\!)^b_t \stackrel{\text{def}}{=} \exists x\, y'.F(b,t,[y'/y]\widetilde{a}) \wedge \texttt{Merge}(x, y', y)$$

Figure 4 Translation of statements. We assume that existentially quantified variables are chosen so as to avoid clashes with free variables in the translated statement.

In addition to the clauses $F_i(b,t,\widetilde{x}_i) \Leftarrow (\!|M_i|\!)^b_t$, we add auxiliary clauses for discarding threads that are irrelevant to the safety property. For each function f_i in D, we add

$$F_i(\texttt{false}, t, \widetilde{x}_i) \Leftarrow \widetilde{s}_i = \widetilde{[\,]} \qquad \text{where } \widetilde{s}_i = \mathit{OChan}(\widetilde{x}_i).$$

Informally, this clause allows us to ignore the future behavior of a thread from any program point f_i at any time t, provided that the thread is assumed not to perform any further sends and not to reach `fail`. The reason is that the safety property of interest asks only whether some thread can reach `fail`; once such a failure occurs, the whole computation aborts, and the subsequent behavior of the other threads becomes irrelevant. The condition $\widetilde{s}_i = \widetilde{[\,]}$ is needed to ensure that the discarded thread contributes no further messages.

Besides the clauses corresponding to function definitions, we add a few more clauses. We add the clause $\bot \Leftarrow F(\texttt{true}, t, \widetilde{a})$, which ensures that the resulting CHC system becomes unsatisfiable whenever executing the main function $f(\widetilde{a})$ can reach `fail`. Finally, we also add the following clauses for auxiliary predicates `Merge` and `Sorted`.

$$\texttt{Sorted}([\,]) \Leftarrow \top. \qquad \texttt{Sorted}([(t,v)]) \Leftarrow \top.$$
$$\texttt{Sorted}((t_1,v_1) :: (t_2,v_2) :: xs) \Leftarrow t_1 < t_2 \wedge \texttt{Sorted}((t_2,v_2) :: xs).$$
$$\texttt{Merge}([\,],[\,],[\,]) \Leftarrow \texttt{true}. \qquad \texttt{Merge}(p :: xs, ys, p :: zs) \Leftarrow \texttt{Merge}(xs, ys, zs).$$
$$\texttt{Merge}(xs, p :: ys, p :: zs) \Leftarrow \texttt{Merge}(xs, ys, zs).$$

We write $(\!|(D, f(\widetilde{a}))|\!)$ for the system of CHCs translated from $(D, f(\widetilde{a}))$ according to the above procedure.

Translation of statements. We complete the definition of $F_i(b,t,\widetilde{x}_i) \Leftarrow (\!|M_i|\!)^b_t$ by defining translation $(\!|-|\!)^b_t$ from statements (except for () and `fail`) in Figure 4. Note that $(\!|-|\!)^b_t$ is parameterized by a failure flag and timestamp. The timestamp t is not a global clock: it records only the most recent send or receive, so as to capture causal dependencies between messages. It is also worth noting that the existential quantifiers in $(\!|-|\!)^b_t$ are introduced solely for readability; in the clause $F_i(b,t,\widetilde{x}_i) \Leftarrow (\!|M_i|\!)^b_t$ the existentially quantified variables can equivalently be universally quantified at the top level.

We explain several key cases. In the encoding of a send operation, we check that the prophecy is consistent with the send operation: the first element of the list must be (t_s, a), where a denotes the value being sent and t_s is the timestamp of that send. The timestamp

t_s is passed to the predicate F, since it represents the time at which the send operation is performed. We require $t \leq t_s$, because t records the most recent send or receive in the current thread. (Note that only send and receive operations update the timestamp.) Since the subsequent execution of $f(\widetilde{a})$ must follow the remaining prophecy, we require $F(b, t_s, [x'/x]\widetilde{a})$. Similarly, the rule for receiving checks that the prophecy for the receiver side is consistent with the receive operation. The timestamp t_r represents the time at which the message is received. We require $t_r > t_s$, where t_s is the time at which the message was sent. This ensures that a message is not received before it is sent, as discussed in Example 5. In the rule for channel creation, x and y represent the sender-side and receiver-side prophecies for future messages, respectively. The condition $x = y$ ensures that they coincide, and $\texttt{Sorted}(x)$ ensures that the timestamps in x are in ascending order. The rule for `clone` expresses that the prophecy y is split into two parts: one for the cloned channel x, and the other for the remaining use of y. This split is represented by the predicate $\texttt{Merge}(x, y', y)$, which holds if and only if y is an interleaving of x and y'; for example $\texttt{Merge}([(1,2);(5,4)],[(2,3)],[(1,2);(2,3);(5,4)])$ holds. The predicate `Merge` is definable using CHCs.

▶ **Example 4.** Recall the program in Example 1. We have already seen in the introduction how it is translated into CHCs without a timestamp. Below we show the CHCs that are obtained by using the translation rules; we only show the important clauses and omit some clauses such as $F_i(\texttt{false}, t, \widetilde{x}_i) \Leftarrow \widetilde{c}_i = \widetilde{[\,]}$ or the clause for `Sorted`.

$$\bot \Leftarrow \mathit{MsgCount}(\texttt{true}, t, n)$$
$$\mathit{MsgCount}(b, t, n) \Leftarrow \mathit{For}(b, t, 0, sr) \wedge r = s \wedge \texttt{Sorted}(s)$$
$$\mathit{For}(b, t, i, n, s, r) \Leftarrow i < n \wedge F_4(b, t, i, n, s, r)$$
$$\mathit{For}(b, t, i, n, s, r) \Leftarrow i \geq n \wedge \mathit{Loop}(b, t, 0, n, r) \wedge s = [\,]$$
$$F_4(b, t, i, n, s, r) \Leftarrow \texttt{Merge}(s_1, s_2, s) \wedge F_5(b, t, i, n, s_1, r, s_2)$$
$$F_5(b_1 \vee b_2, t, i, n, s, r, s') \Leftarrow F_6(b_1, t, s') \wedge \mathit{For}(b_2, t, i+1, n, s, r)$$
$$\mathit{Loop}(b, t, c, n, r) \Leftarrow r = (t_s, \_v) :: r' \wedge t_s < t_r \wedge t \leq t_r \wedge \mathit{Assert}(b, t_r, c+1, n, r')$$
$$\mathit{Assert}(b, t, c, n, r) \Leftarrow c \leq n \wedge \mathit{Loop}(b, t, c, n, r) \qquad \mathit{Assert}(b, t, c, n, r) \Leftarrow c > n \wedge \mathit{Fail}(b, t, c, n, r) \qquad \mathit{Fail}(b, t, c, n, r) \Leftarrow b = \texttt{true}$$
$$F_6(\texttt{false}, t, s) \Leftarrow s = [(t_s, 1)] \wedge t \leq t_s. \qquad \lrcorner$$

▶ **Example 5.** To illustrate the need for timestamps we consider the following simple program (on the left-hand side) and the corresponding CHCs (on the right-hand side).

$f_0() = \texttt{new}\ s, r\ \texttt{in}\ f_1(s, r)$	$F_0(b, t) \Leftarrow F_1(b, t, s, r) \wedge r = s \wedge \texttt{Sorted}(s)$
$f_1(s, r) = \texttt{let}\ x = \texttt{recv}\ r\ \texttt{in}\ f_2(s)$	$F_1(b, t, s, r) \Leftarrow r = [(t_s, x)] \wedge F_2(b, t_r, s)$
	$\wedge\, t_s < t_r \wedge t \leq t_r$
$f_2(s) = \texttt{send}\ 0\ \texttt{to}\ s; f_3()$	$F_2(b, t, s) \Leftarrow s = [(t_s, 0)] \wedge F_3(b, t_s) \wedge t \leq t_s$
$f_3() = \texttt{fail}$	$F_3(b, t) \Leftarrow b = \texttt{true}$
	$\bot \Leftarrow F_0(\texttt{true}, t).$

The program on the left-hand side never fails because the execution is blocked by the receive from r. However, if we ignore the timestamps, that is, omit the red parts of the CHCs, then the resulting system of CHCs on the right-hand side becomes unsatisfiable, which means that such a translation is incomplete. Indeed, by repeatedly applying the clauses backwards, we obtain

$$\bot \Leftarrow F_0(\texttt{true}) \Leftarrow F_1(\texttt{true}, [0], [0]) \Leftarrow F_2(\texttt{true}, [0]) \Leftarrow F_3(\texttt{true}) \Leftarrow \top.$$

Thus, $\bot$ is derivable. The prophecy [0] above records that the message 0 would be sent on s, but does not capture the causal dependency between the receive and send operations. On the other hand, with timestamps, a similar derivation would contain the step

$$F_1(\texttt{true}, t_0, [(t_s, 0)], [(t_s, 0)]) \Leftarrow F_2(\texttt{true}, t_r, [(t_s, 0)]) \text{ where } t_s < t_r \wedge t_r < t_s \wedge t_0 \leq t_r.$$

Here, the constraint $t_r < t_s$ expresses the sequential causality between f_1 and f_2, whereas $t_s < t_r$ comes from the causal dependency induced by the channel. Since these constraints are contradictory, we cannot derive $\bot$. ⌟

3.3 Properties of the Translation

Our translation is sound and complete in the sense that a program is free from failure if and only if the corresponding system of CHCs is satisfiable. We write $C \not\Longrightarrow_D^* C'$ if there is no reduction sequence from C to C'.

▶ **Theorem 6** (Soundness and Completeness). *Let* $\vdash (D, f(\widetilde{a})) : \star$. *Then* $(\!|(D, f(\widetilde{a}))|\!)$ *is a well-sorted system of CHCs that is satisfiable if and only if* $(\{ f(\widetilde{a}) \}, \emptyset) \not\Longrightarrow_D^* \texttt{fail}$. ⌟

The proof is given in the longer version of this paper [31]. Here we provide a brief overview of the argument. The core argument is a correspondence between (i) an execution leading to `fail` and (ii) a derivation of contradiction from the translated CHCs. A subtlety is that such a derivation generally has a tree structure rather than a sequence, because the clause for `spawn` is non-linear. Recovering an execution therefore requires linearizing this tree into a thread schedule while preserving causality. The timestamp information is precisely what makes this possible.

4 Implementation and Experiments

We implemented a verification tool based on our reduction and conducted preliminary experiments on small benchmarks to examine the effectiveness of our approach. Our tool takes as input a program written in the language defined in Section 2. Then it generates CHCs, either with or without timestamps, by applying our reduction described in Section 3. The tool first passes the CHCs with timestamps to a portfolio CHC solver that runs Spacer [25], Eldarica [15], and Catalia [19] in parallel.[3] If one of the solvers returns `sat` (resp. `unsat`), the tool reports `safe` (resp. `unsafe`). If all solvers time out, the tool instead passes the CHCs without timestamps, as an approximate translation, to the same portfolio solver. If any solver reports that the approximate CHCs are satisfiable, the tool returns `safe`; otherwise, it returns `unknown`.

We now describe the experiments. We ran the tool on our benchmark suite, which consists of Example 1, Example 5, and the programs in Appendix B. Example 5 and `ack` are the examples for which the results differ depending on whether timestamps are included. Instances whose names have the suffix "-e" are obtained by modifying the corresponding instances without the suffix so that an assertion fails. All the benchmark instances were run on a commodity laptop (2.8 GHz Intel Core i7 with 16 GB RAM). The timeout for the portfolio solver was set to 120 seconds.

[3] The versions of the solvers used in our experiments are as follows: Spacer 4.15.5, Eldarica 2.2.1, and Catalia with commit hash 227e6b4ae6870cd795a7dbc4a863c45dcb900945.

Table 1 Results for safe cases.

Instance	*Solving Time*
Example 1	19.5
Example 5	< 0.1
ack	0.2
multi-sends	50.4(+120)
client-server	timeout
calc-server	0.8

Table 2 Results for unsafe cases.

Instance	*Solving Time*
Example 1-e	< 0.1
ack-e	0.1
multi-sends-e	< 0.1
client-server-e	< 0.1
calc-server-e	0.1

```
fn main(n: usize) {
  let (s1, r1) = channel();
  let (s2, r2) = channel();
  spawn(move || {
    loop {// #[invariant(r2.recv = 1)]
      let v = r2.recv().unwrap();
      s1.send(v + 1).unwrap();
    }
  });
  let mut sum = 0;
  for i in 0..n {
    s2.send(1).unwrap();
    let v = r1.recv().unwrap();
    sum += v;
  }
  assert!(sum == n * 2);
} // end of main
```

Figure 5 Client-server program. (A single program is shown over two columns.)

Table 1 and Table 2 show the experimental results. Table 1 lists the safe-case benchmarks, and Table 2 lists the unsafe-case benchmarks. In both tables, the "Solving time" column shows the time taken by the portfolio solver to return a result. Here, `timeout` means that the portfolio solver timed out on both the CHCs with timestamps and those without timestamps after 120 seconds, and "(+120)" indicates that it timed out on the CHCs with timestamps. Table 1 shows that, except for `client-server`, all benchmarks in the safe set were verified to be safe within the time limit. Table 2 shows that unsafety was detected in all buggy benchmark programs.

The `client-server` instance illustrates a limitation of the current backend CHC solvers rather than a fundamental limitation of our reduction. The program `client-server` is shown in Figure 5. In the function `main`, a spawned thread acts as a server that adds 1 to the received value and sends back the result. Another thread repeatedly sends 1, receives the response, and repeats this n times, asserting that the sum of the received values is equal to $2n$. The reason this instance is not verified is that the backend CHC solvers fail to discover the invariants stating that the values received through `r1` and `r2` are always 1 and 2, respectively. More precisely, Eldarica and Spacer cannot synthesize predicates involving recursively defined predicates such as "the list only contains 1". On the other hand, in principle, Catalia can synthesize such predicates as long as they are expressible as arithmetic formulas over integer values obtained by applying a certain class of catamorphisms to lists. In practice, however, Catalia fails to synthesize a suitable predicate because it relies on heuristics to infer the catamorphism to use.

The example above suggests that stronger backend CHC solvers for ADTs are needed to make our approach more practical. We are addressing this issue in a separate line of work on CHC solving for ADTs [19, 22]. An alternative way to address the limitation above is to make use of user-provided annotations. We have confirmed that if we manually provide an appropriate catamorphism – namely, one that counts the numbers of 1s and 2s appearing in a list – then Catalia can verify that `client-server` is safe. Directly providing such a catamorphism as a hint would impose a burden on users who are not familiar with the

backend CHC solvers. A more practical approach would be to extend our tool so that annotations in the source program, such as the comment in Figure 5, are translated into hints that help Catalia synthesize the required catamorphism.

5 Discussion

5.1 Variations of Message-Passing Semantics

Our prophecy-based approach is not tied to the particular message-passing semantics considered so far: it can also be adapted to other message-passing semantics with only modest changes to the translation. The semantics of the target language in Figure 3 is based on Rust's standard mpsc library. In this semantics, messages are enqueued in the FIFO buffer owned by the corresponding receiver in the order in which they are sent, and hence the send order is preserved. In settings such as distributed computing, however, it is also natural to consider the following two variants:

1. a semantics in which the order of messages sent by each individual sender is preserved but no ordering is imposed between messages sent by different senders associated with the same receiver; and
2. a semantics in which the order of messages sent is not preserved at all, as in the unordered buffering modeled by the asynchronous π-calculus [4].

The two variants can be captured by modifying how message buffers are represented. Preserving the order of messages only within each sender channel suggests maintaining a separate FIFO queue for each sender channel. In our setting, however, sender channels can be cloned, and this cloning structure must also be reflected in the buffer. This leads to a tree-structured buffer: each leaf corresponds to a sender channel, and when a sender is cloned, the corresponding leaf is replaced by an internal node with two children for the original sender and the newly created clone. The internal node stores the queue of messages sent before the cloning, while the leaves store the messages sent after the cloning.

To model unordered buffering, we instead use multisets: send operations add elements to the multiset, and receive operations remove elements from it nondeterministically.

We illustrate the differences among these three semantics using the example in Figure 6. With the original semantics in Figure 3, after all send operations, the buffer state is the FIFO queue $[0, 1, 2]$. Therefore, the values are received in the order $0, 1, 2$. If we instead use the tree-structured buffer, the buffer state is $\mathsf{Node}([0], s_1 \mapsto [1], s_2 \mapsto [2])$, where the root queue $[0]$ stores the messages sent before cloning, and each leaf $s_1 \mapsto [1]$ and $s_2 \mapsto [2]$ represents the sender and the messages sent via the sender after cloning. Thus, the value 0 is received first, after which either 1 or 2 may be received next. Hence the possible receive orders are $0, 1, 2$ and $0, 2, 1$. Finally, with multiset buffers, the buffer state after the sends is the multiset $\{0, 1, 2\}$. Since a multiset does not preserve order, the values may be received in any order.

We next describe how the translation changes for these two variants. The tree structure is already reflected by the use of `Merge` at `clone` operations: each clone splits a sender prophecy into two branches. Thus, the only change needed is to omit the constraint $\texttt{Sorted}(x)$, which otherwise imposes a global FIFO order on all messages sent to the receiver. For the multiset semantics, we need to represent prophecies using multisets rather than lists. Accordingly, the auxiliary predicates must be modified. The predicate $\texttt{Merge}(x, y', y)$ is replaced by $\texttt{Union}(x, y', y)$, expressing that the union of x and y' is y. Since $\texttt{Sorted}(x)$ has no meaning for multisets, it is omitted. The soundness and completeness proofs require only local changes.

As another variant of the semantics, we can also handle bounded buffers by adding receive-time information to prophecies. Assume that the buffer size is bounded by k. In this case, the sender and receiver prophecies are extended to contain receive-time timestamps

```
fn main() {
    let (s1, r) = channel();
    s1.send(0).unwrap();
    let s2 = s1.clone();
    s1.send(1).unwrap();
    s2.send(2).unwrap();
    let v1 = r.recv().unwrap();
    let v2 = r.recv().unwrap();
}
```

Figure 6 Example program illustrating the differences among the three semantics.

in addition to send-time timestamps, and the `Sorted` predicate additionally imposes the constraint that, for each i, the send-time timestamp of the $(i + k)$-th message is later than the receive-time timestamp of the i-th message. This constraint states that no further value is sent when there are already k unreceived values accumulated in the buffer, which is exactly the buffer-size constraint.

5.2 Deadlock Freedom

Deadlock freedom is another important property of concurrent programs. Our approach can also be adapted to this property by adjusting the translation to CHCs, as sketched below. A full formalization of this extension is left for future work.

In the translation described in Section 3, the flag b is Boolean and indicates whether the program can reach `fail`. To check deadlock freedom, we extend this flag to a three-valued status flag: `F` ("fail"), `T` ("terminated"), and `B` ("blocked"). The statuses `F`, `T`, and `B` indicate that the remaining computation can reach `fail`, terminate normally, and reach a state blocked at a `recv` statement, respectively. In the last case, the computation is blocked because no value will ever be sent on the corresponding channel.

We illustrate the translation for checking deadlock freedom using the example in Figure 7. The program in Figure 7 illustrates a typical deadlock: each of the two threads waits to receive a value from the other.

We make the following changes to the translation from programs to CHCs. First, the translation rule for `recv` statements is extended with a case in which an empty receiver prophecy causes the status flag to be set to `B`.[4] In the translation shown in Figure 7, the following CHCs correspond to this extension.

$$G_0(\mathtt{B}, t, [\,], r_2) \Leftarrow r_2 = [\,] \quad F_3(\mathtt{B}, t, r_1, [\,]) \Leftarrow r_1 = [\,].$$

Next, at a `spawn` statement, the statuses of the two spawned computations are combined so that the overall computation is reported as `B` when both computations either terminate or become blocked and at least one of them becomes blocked. In Figure 7, this combination is performed in the clause for F_2, where the status b is constrained by $b = b_1 * b_2$. Here, the operator $*$ is defined as follows.

$$\mathtt{T} * \mathtt{T} = \mathtt{T} \quad \mathtt{B} * \mathtt{B} = \mathtt{B} * \mathtt{T} = \mathtt{T} * \mathtt{B} = \mathtt{B} \quad \mathtt{F} * \_ = \_ * \mathtt{F} = \mathtt{F}$$

[4] For receive operations whose blocking is intended to be acceptable, such as those in a server thread waiting for requests, the empty-prophecy case can instead be treated as `T`, so that such waiting is not reported as a deadlock.

$$
\begin{aligned}
&f_0() = \texttt{new}\ s_1, r_1\ \texttt{in}\ f_1(s_1, r_1)\\
&f_1(s_1, r_1) = \texttt{new}\ s_2, r_2\ \texttt{in}\ f_2(\widetilde{s}, \widetilde{r})\\
&f_2(\widetilde{s}, \widetilde{r}) = \texttt{spawn}(g_0(s_1, r_2)); f_3(r_1, s_2)\\
&g_0(s_1, r_2) = \texttt{let}\ v = \texttt{recv}\ r_2\ \texttt{in}\ g_1(s_1)\\
&g_1(s_1) = \texttt{send}\ 0\ \texttt{to}\ s_1; ()\\
&f_3(r_1, s_2) = \texttt{let}\ v = \texttt{recv}\ r_1\ \texttt{in}\ f_4(s_2)\\
&f_4(s_2) = \texttt{send}\ 0\ \texttt{to}\ s_2; ()
\end{aligned}
$$

$$
\begin{aligned}
&\bot \Leftarrow F_0(\texttt{B}, t)\\
&F_0(b, t) \Leftarrow F_1(b, t, s_1, s_1) \wedge \texttt{Sorted}(s_1)\\
&F_1(b, t, s_1, r_1) \Leftarrow F_2(b, t, \widetilde{s}, r_1, s_2) \wedge \texttt{Sorted}(s_2)\\
&F_2(b, t, r, \widetilde{s}, \widetilde{r}) \Leftarrow G_0(b_1, t, s_1, r_2) \wedge F_3(b_2, t, r_1, s_2)\\
&\qquad\qquad \wedge b = b_1 * b_2\\
&G_0(b, t, s_1, r_2) \Leftarrow G_1(b, t_r, s_1) \wedge r_2 = (t_s, v) :: r_2'\\
&\qquad\qquad \wedge t_s < t_r \wedge t \leq t_r\\
&G_0(\texttt{B}, t, [\,], r_2) \Leftarrow r_2 = [\,]\\
&G_1(\texttt{T}, t, s_1) \Leftarrow s_1 = [(t_s, 0)] \wedge t \leq t_s\\
&F_3(b, t, r_1, s_2) \Leftarrow F_4(b, t_r, s_2) \wedge r_1 = (t_s, v) :: r_1'\\
&\qquad\qquad \wedge t_s < t_r \wedge t \leq t_r\\
&F_3(\texttt{B}, t, r_1, [\,]) \Leftarrow r_1 = [\,]\\
&F_4(\texttt{T}, t, s_2) \Leftarrow s_2 = [(t_s, 0)] \wedge t \leq t_s
\end{aligned}
$$

Figure 7 Example program that causes a deadlock and its translation.

We also modify the goal clause so that it derives $\bot$ whenever the status of the whole program is $\texttt{B}$. Consequently, the resulting CHC system is unsatisfiable if the program can deadlock. In the example in Figure 7, both G_0 and F_3 can have status $\texttt{B}$, and hence the clause for F_2 gives the combined status $\texttt{B} * \texttt{B} = \texttt{B}$. The status of the entire program is therefore $\texttt{B}$, signaling that a deadlock can occur.

6 Related Work

Type-based approaches have been popular for the automated verification and analysis of message-passing concurrent programs: see [20, 16] for surveys. While such approaches are often lightweight, they are generally not complete and may yield false alarms. Ueno and Das [33] recently proposed a type inference algorithm for refinement binary session types, which can automatically verify the absence of assertion failures. However, binary session types are designed for one-to-one communication and do not directly support multiple senders. Moreover, their inference method is template-based and does not enjoy relative completeness. By contrast, our translation is relatively complete assuming an oracle for CHC solving; this assumption is, of course, idealized, since CHC solving is undecidable in general.

Model checking [8, 9, 3] has also been widely used for the automated verification of concurrent systems. However, standard finite-state or pushdown model checking is not directly applicable to systems with unbounded numbers of threads and unbounded message queues. There are decidable classes of message-passing concurrent systems, such as lossy channel systems [2], but our target language is Turing-complete and therefore cannot be captured by such models. Bounded model checking is often used for infinite state systems, but it is unsound in general.

Our use of prophecies [1] to enable modular modeling of sender and receiver processes is related in spirit to thread-modular verification for shared-memory multi-threaded programs [12, 14, 27]. In thread-modular verification, suitable assumptions about inter-thread interactions are inferred and then used to verify each thread modularly. Although shared-memory and message-passing concurrency differ significantly, our prophecies about future messages play a role analogous to such assumptions.

Prophecies were also incorporated into the Iris separation logic [18] by Jung et al. [17] for modular verification of logical atomicity. Their mechanism supports various types of concurrency, but unlike ours, it is not designed for automation and does not support partial resolution of list prophecies, which our work uses to partially determine the contents of the prophesied list. We also note that prophecy variables were originally introduced by Abadi and Lamport to prove refinement between state machines [1], rather than as a mechanism for automated verification.

Our verification technique was inspired by the prophecy-based technique of RustHorn by Matsushita et al. [30], which has also been applied to semi-automated Rust verification tools such as Creusot [11], Verus [26] and Thrust [32], and has been justified semantically in the Iris separation logic [28, 29, 13]. They considered *sequential* programs with mutable references, representing a borrowed mutable reference as a pair of its current value and the future value at the point where the borrow is released. Our work and RustHorn both prophesy future interactions between entities (channel senders and receivers, or borrowers and lenders) for automated modular functional reasoning. However, our work is novel in that it supports multi-shot, multi-sender channels, where a list of multiple sent values is prophesied, and the send/receive events can interleave in a complex way depending on thread scheduling. Roughly speaking, Rust-style borrows are like a one-shot, single-sender channel.

Automated verification based on CHCs [5] or, more generally, fixpoint logic [24, 6] has attracted considerable attention in recent years. Various verification problems can be encoded and solved using common backend solvers for CHCs [25, 15, 19, 7] or fixpoint logics [21, 23, 34]. Most such verification techniques have primarily been developed for sequential programs, and, to the best of our knowledge, the CHC-based verification of message-passing concurrent programs has not been studied before.

7 Conclusion

We proposed a novel method for verifying the functional correctness of message-passing concurrent programs. Inspired by RustHorn's prophecy-based technique, we represented each communication channel as a list of future messages to be sent on that channel, thereby reducing the verification problem to CHC solving while preserving soundness and completeness. We also implemented a prototype tool to demonstrate the usefulness of our method.

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.

2 Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pages 160–170. IEEE Computer Society, 1993. doi:10.1109/LICS.1993.287591.

3 Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

4 Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. On the asynchronous nature of the asynchronous pi-calculus. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science, pages 473–492. Springer, 2008. doi:10.1007/978-3-540-68679-8_29.

5 Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *LNCS*, pages 24–51. Springer, 2015. doi:10.1007/978-3-319-23534-9_2.

6 Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. Higher-order constrained Horn clauses for verification. *Proc. ACM Program. Lang.*, 2(POPL):11:1–11:28, 2018. doi:10.1145/3158099.

7 Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. HoIce: An ICE-based non-linear Horn clause solver. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 146–156. Springer, 2018. doi:10.1007/978-3-030-02768-1_8.

8 Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

9 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

10 Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi, and Maurizio Proietti. Analysis and transformation of constrained Horn clauses for program verification. *Theory and Practice of Logic Programming*, 22(6):974–1042, 2022. doi:10.1017/S1471068421000211.

11 Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of Rust programs. In *International Conference on Formal Engineering Methods*, pages 90–105. Springer, 2022. doi:10.1007/978-3-031-17244-1_6.

12 Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, Lecture Notes in Computer Science, pages 262–277. Springer, 2002. doi:10.1007/3-540-45927-8_19.

13 Travis Hance, Laila Elbeheiry, Yusuke Matsushita, and Derek Dreyer. VerusBelt: A semantic foundation for Verus's proof-oriented extensions to the Rust type system. *Proc. ACM Program. Lang.*, 10(PLDI), June 2026. doi:10.1145/3808325.

14 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, Lecture Notes in Computer Science, pages 262–274. Springer, 2003. doi:10.1007/978-3-540-45069-6_27.

15 Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7. IEEE, 2018. doi:10.23919/FMCAD.2018.8603013.

16 Hans Hüttel, Ivan Lanese, Vasco T Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, et al. Foundations of session types and behavioural contracts. *ACM Computing Surveys (CSUR)*, 49(1):1–36, 2016. doi:10.1145/2873052.

17 Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019. doi:10.1145/3371113.

18 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

19 Hiroyuki Katsura, Naoki Kobayashi, Ken Sakayori, and Ryosuke Sato. Automated catamorphism synthesis for solving constrained Horn clauses over algebraic data types. In *International Static Analysis Symposium*, pages 305–327. Springer, 2025. doi:10.1007/978-3-032-07106-4_13.

20 Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers*, pages 439–453. Springer, 2003. doi:10.1007/978-3-540-40007-3_26.

21 Naoki Kobayashi, Takeshi Nishikawa, Atsushi Igarashi, and Hiroshi Unno. Temporal verification of programs via first-order fixpoint logic. In Bor-Yuh Evan Chang, editor, *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11822 of *Lecture Notes in Computer Science*, pages 413–436. Springer, 2019. doi:10.1007/978-3-030-32304-2_20.

22 Naoki Kobayashi, Ryosuke Sato, Ayumi Shinohara, and Ryo Yoshinaka. Solvable tuple patterns and their applications to program verification. *Proc. ACM Program. Lang.*, 10(PLDI), June 2026. doi:10.1145/3808258.

23 Naoki Kobayashi, Kento Tanahashi, Ryosuke Sato, and Takeshi Tsukada. HFL(Z) validity checking for automated program verification. *Proc. ACM Program. Lang.*, 7(POPL):154–184, 2023. doi:10.1145/3571199.

24 Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. Higher-order program verification via HFL model checking. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, Lecture Notes in Computer Science, pages 711–738. Springer, 2018. doi:10.1007/978-3-319-89884-1_25.

25 Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.*, 48(3):175–205, 2016. doi:10.1007/s10703-016-0249-4.

26 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.

27 Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 218–232, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74061-2_14.

28 Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 841–856. ACM, 2022. doi:10.1145/3519939.3523704.

29 Yusuke Matsushita and Takeshi Tsukada. Nola: Later-free ghost state for verifying termination in Iris. *Proc. ACM Program. Lang.*, 9(PLDI):98–124, 2025. doi:10.1145/3729250.

30 Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based verification for Rust programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(4):1–54, 2021. doi:10.1145/3462205.

31 Takashi Nagatomi, Musashi Katsura, Naoki Kobayashi, Yusuke Matsushita, and Ken Sakayori. Prophecy-based automated verification of message-passing programs (full version). *CoRR*, 2026. doi:10.48550/arXiv.2606.28066.

32 Hiromi Ogawa, Taro Sekiyama, and Hiroshi Unno. Thrust: A prophecy-based refinement type system for Rust. *Proc. ACM Program. Lang.*, 9(PLDI):2056–2080, 2025. doi:10.1145/3729333.

33 Toby Ueno and Ankush Das. Practical refinement session type inference. In Robbert Krebbers, editor, *Programming Languages and Systems*, pages 298–330, Cham, 2026. Springer Nature Switzerland. doi:10.1007/978-3-032-22723-2_11.

34 Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. Modular primal-dual fixpoint logic solving for temporal verification. *Proc. ACM Program. Lang.*, 7(POPL):2111–2140, 2023. doi:10.1145/3571265.

A An Additional Example of the Translation

▶ **Example 7.** The following is a program with causal dependencies between channels that are used across different threads.[5] It sends messages across three threads f, g and h as illustrated in the left-hand side of Figure 8 and checks whether the values received by r are 10 and then 20. The corresponding CHCs are given side-by-side.

$$
\begin{array}{ll}
 & \bot \Leftarrow F_0(\texttt{true}, t) \\
f_0() = \texttt{new } s_1, r \texttt{ in } f_1(s_1, r) & F_0(b, t) \Leftarrow F_1(b, t, s, r) \wedge r = s \wedge \texttt{Sorted}(s) \\
f_1(s_1, r) = \texttt{new } ack_s, ack_r \texttt{ in } f_2(s_1, r, \widetilde{ack}) & F_1(b, t, s, r) \Leftarrow F_2(b, t, s, r, \widetilde{ack}) \wedge ack_r = ack_s \\
 & \qquad \wedge \texttt{Sorted}(ack_s) \\
f_2(s_1, r, \widetilde{ack}) = & F_2(b, t, s, r, \widetilde{ack}) \Leftarrow \texttt{Merge}(s_1, s_2, s) \\
\quad \texttt{let } s_2 = \texttt{clone } s_1 \texttt{ in } f_3(\widetilde{s}, r, \widetilde{ack}) & \qquad \wedge F_3(b, t, \widetilde{s}, r, \widetilde{ack}) \\
f_3(\widetilde{s}, r, \widetilde{ack}) = \texttt{spawn}(g(s_1)); f_4(s_2, r, \widetilde{ack}) & F_3(b_1 \vee b_2, t, \widetilde{s}, r, \widetilde{ack}) \Leftarrow G(b_1, t, s_1) \\
 & \qquad \wedge F_4(b_2, t, s_2, r, \widetilde{ack}) \\
g(s_1) = \texttt{send } 10 \texttt{ to } s_1; () & G(\texttt{false}, t, s_1) \Leftarrow s_1 = [(t_{s_1}, 10)] \wedge t \leq t_s \\
f_4(s_2, r, \widetilde{ack}) = & F_4(b_1 \vee b_2, t, s_2, r, \widetilde{ack}) \Leftarrow H_1(b_1, t, s_2, ack_r) \\
\quad \texttt{spawn}(h_1(s_2, ack_r)); f_5(r, ack_s) & \qquad \wedge F_5(b_2, t, r, ack_s) \\
h_1(s_2, ack_r) = & H_1(b, t, s_2, ack_r) \Leftarrow ack_r = [(t_{ack_s}, \_)] \\
\quad \texttt{let } \_ = \texttt{recv } ack_r \texttt{ in } h_2(s_2) & \quad \wedge H_2(b, t_{ack_r}, s_2, ack_r') \wedge t_{ack_s} < t_{ack_r} \wedge t \leq t_{ack_r} \\
h_2(s_2) = \texttt{send } 20 \texttt{ to } s_2; () & H_2(\texttt{false}, t, s_2) \Leftarrow s_2 = [(t_{s_2}, 20)] \wedge t \leq t_{s_2} \\
f_5(r, ack_s) = & F_5(b, t, r, ack_s) \Leftarrow r = (t_{s_1}, v_1) :: r' \\
\quad \texttt{let } v_1 = \texttt{recv } r \texttt{ in } f_6(r, ack_s, v_1) & \quad \wedge F_6(b, t_r, r', ack_s, v_1) \wedge t_{s_1} < t_r \wedge t \leq t_r \\
f_6(r, ack_s, v_1) = \texttt{send } 0 \texttt{ to } ack_s; f_7(r, v_1) & F_6(b, t, r, ack_s, v_1) \Leftarrow ack_s = [(t_{ack_s}, 0)] \wedge F_7(r, v_1) \\
 & \qquad \wedge t \leq t_{ack_s} \\
f_7(r, v_1) = \texttt{let } v_2 = \texttt{recv } r \texttt{ in } f_8(v_1, v_2) & F_7(b, t, r, v_1) \Leftarrow r = [(t_{s_2}, v_2)] \wedge F_8(b, t_r, v_1, v_2) \\
 & \qquad \wedge t_{s_2} < t_r \wedge t \leq t_r \\
f_8(v_1, v_2) = \texttt{assert}(v_1 = 10 \;\&\&\; v_2 = 20) & F_8(b, t, v_1, v_2) \Leftarrow b \neq (v_1 = 10 \wedge v_2 = 20)
\end{array}
$$

The assertion never fails because in order for h to send the value 20, (1) the value 10 must be sent before a message is sent using ack_s; (2) ack_r must wait for ack_s; and (3) a message must be received via ack_r for h to send 20. The red part of the CHCs imposes an order $t_{ack_r} \leq t_{s_2}$ which captures the last causal dependency; the other two dependencies $t_{s_1} < t_{ack_s}$ and $t_{ack_s} < t_{ack_r}$ can similarly be read from the CHCs.

If there were no timestamps, then $s \mapsto [20, 10]$ would become a valid prophecy. This prophecy does not capture the correct behavior as it corresponds to a situation where the send using s_2 happened before the send using s_1 as, for example, in the right-hand side of Figure 8. ⌟

[5] The example relaxes the syntax and typing rules of the language for readability; for example, we drop variables before reaching ().

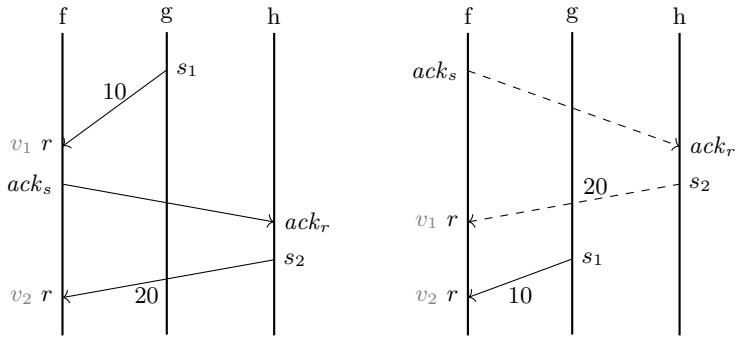

Figure 8 Message sequence diagrams for Example 7: the left shows a correct sequence, the right an incorrect one where the dotted lines are communications that cannot happen.

B Benchmark Programs for the Experiments in Section 4

▶ **Example 8.** The program of the benchmark `running-ex-e` is shown below. In this program, the loop that spawns sender threads iterates one more time than in Example 1, which causes the assertion to fail when the number of messages received exceeds n.

```
fn msg_count(n: usize) {
    let (s, r) = channel();
    for i in 0..=n { // spawn n senders
        let s_clone = s.clone();
        thread::spawn(move || {
            s_clone.send(1).unwrap(); // each thread sends 1 to the channel
        });
    }
    let mut c = 0; // number of messages received
    loop {  // repeatedly receive and count messages
        let _v = r.recv().unwrap();
        c += 1;  // increments the message counter
        assert!(c <= n); // error if more than n messages are received
    }
}
```

▶ **Example 9.** The programs of the benchmarks `multi-sends` and `multi-sends-e` are shown below. In the function `main`, one thread sends the value 1 n times, while another thread receives it n times and asserts that their sum equals n. In the function `main_with_bugs`, although it should send and receive values n times, it mistakenly does so only $n-1$ times.

```
fn main(n: usize) {
    let (s, r) = channel();
    spawn(move || {
        for i in 0..n {
            s.send(1).unwrap();
        }
    });
    let mut sum = 0;
    for i in 0..n {
        let v = r.recv().unwrap();
        sum += v;
    }
    assert!(sum == n);
} // end of main
```

```
fn main_with_bugs(n: usize) {
    let (s, r) = channel();
    spawn(move || {
        for i in 1..n {
            s.send(1).unwrap();
        }
    });
    let mut sum = 0;
    for i in 1..n {
        let v = r.recv().unwrap();
        sum += v;
    }
    assert!(sum == n);
} // end of main_with_bugs
```

The programs and explanations of the benchmarks `ack-e`, `client-server-e`, `calc-server` and `calc-server-e` are given in the longer version of this paper [31].