Encoding Network PlusCal into the Join Calculus

Ghilain Bergeron
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Abstract

We present a compilation scheme from Network PlusCal, a dialect of PlusCal equipped with explicit network communication primitives, to the Join Calculus. This scheme maps each Network PlusCal process to a Join Calculus program that encodes local variables as *local state atoms* and control flow as a family of guarded reactions. Mutual exclusion among non-deterministic branches of atomic blocks is enforced by a per-block nullary atom that is consumed once a branch succeeds. We illustrate this scheme on a Ping-Pong example involving a server and multiple client processes.

1 Introduction

Distributed algorithms are notoriously hard to design, implement, and verify, for example due to the difficulty of reproducing timing-dependent behaviors across large sets of interleavings. Several formalisms have been proposed over the years for specifying such systems: TLA$^+$ [12] based on the Temporal Logic of Actions (TLA), Communicating Sequential Processes (CSP) [10], the π-calculus [16], or Event-B [1]. Lamport's PlusCal [13, 14] improves upon TLA$^+$ by providing a pseudocode-like syntax reminiscent of actual (C or Pascal) code, unifying both specification and implementation under the same syntactical constructs. Distributed PlusCal [5] further extends PlusCal with built-in communication and concurrency primitives. Yet, Distributed PlusCal only compiles to a TLA$^+$ specification, leaving the user with the task of writing the implementation code. Our approach follows an alternative pipeline: rather than specifying and implementing distributed algorithms independently and connecting them through informal arguments, we propose to generate implementations from Distributed PlusCal specifications—more precisely from Network PlusCal [3] algorithms, a fragment of Distributed PlusCal where message buffering is made explicit.

In this paper we propose a compilation scheme $\mathcal{C}$ from Network PlusCal to the Join Calculus [8]. The central idea is to represent the mutable local state of a Network PlusCal process as *local state atoms* (single-token channels), and to encode each atomic block as a set of guarded reactions that atomically read, update, and re-emit those atoms. The resulting Join Calculus process can then serve as the basis for an executable backend for Network PlusCal.

Outline. Section 2 discusses work related to the Join Calculus and PlusCal. Section 3 and Section 4 respectively recall Network PlusCal and the Join Calculus. Section 5 then presents the compilation scheme from Network PlusCal to the Join Calculus. Section 6 finally illustrates the compilation of a simple Ping-Pong example.

2 Related Work

The Join Calculus was introduced by Fournet and Gonthier [8] as another foundation for distributed systems programming; expressiveness equivalence with the π-calculus is well-known [7]. Join patterns, the crux of the dynamic behavior of the Join Calculus, appear in several implementations, ranging from full-blown compilers such as JoCaml [6] and Cω [2] to standalone libraries such as JErlang [17].

PlusCal [13] is a standard tool in the TLA$^+$ ecosystem, and Distributed PlusCal [5] extends it with built-in communication and threading primitives. To the best of our knowledge, the connection between Distributed PlusCal and process calculi has not been previously studied. There exist several compilers for different variants of PlusCal: Erla$^+$ [11] translates a subset of PlusCal into Erlang; PGo [9] translates a superset of PlusCal, Modular PlusCal, into the Go programming language; the standard PlusCal compiler [13] produces a TLA$^+$ specification; and recent work by Bergeron et al. [3] introduces first steps towards a mechanically verified compiler for Distributed PlusCal targeting Go. That compiler includes a pre-processing pass for translating Distributed PlusCal specifications into Network PlusCal specifications, which we also use as a preliminary step in the compilation to the Join Calculus.

3 Network PlusCal

In this section, we recall the structure of Network PlusCal algorithms, departing slightly from its true syntax—which can be found in the PlusCal manual [14]—for clarity of presentation.[1] A Network PlusCal *algorithm* is a finite set of named processes running in parallel, communicating via FIFO queues. Each process is assigned a unique FIFO from which it can receive, which we call its *mailbox*.

Processes. A process P is written

$$\text{self} \ \star \ x_1 = e_1, \ldots, x_n = e_n \ \star \ \{T_1\} \ldots \{T_m\}$$

where self is the (unique) "runtime address" of the process (assigned to its self local variable), $x_i = e_i$ are local variable declarations with initial values, and $T_1, \ldots, T_m$ are *threads* that run in parallel within P. One may think of Network PlusCal processes as distant machines communicating over the network, and threads as processes of the machines that synchronize across some locally-shared memory.

[1] The true syntax of Network PlusCal contains many more keywords, such as algorithm or process.

Threads and atomic blocks. Each thread of the process is a list of labelled *atomic blocks*. An atomic block has the form

$$l\colon \text{ either } \{G_1^*; S_1^* \text{ ; goto } l_1\} \text{ or } \ldots \text{ or } \{G_n^*; S_n^* \text{ ; goto } l_n\}$$

where l is the label of the block, each G_i^* is a (potentially empty) sequence of guards, each S_i^* is a sequence of (effectful) statements, and l_i are labels of the blocks scheduled for execution next. An atomic block is a non-deterministic choice among its branches; at most one branch may execute at any given scheduling step. Branches are allowed to execute only if they are *enabled*, that is if all their guards are satisfied. The selected branch runs to completion before another atomic block can be scheduled. Processes are executed on a *per-block basis*: the first block of each thread is initially scheduled for execution, and at each execution step, a single enabled block in any thread (if there is one) is executed at a time. If no such block is found, the process waits for a block to be enabled: it may be waiting for messages to arrive.

Each process is also equipped with a virtual thread $T_{rx}(\mathit{mailbox} \rightarrow \mathit{inbox})$ that relays incoming messages to the process-local variable *inbox* (one of the x_i), buffering them for later use.

Guards and statements. Guards are special kinds of (side-effect-free) statements that control whether a block is *enabled*. In Network PlusCal, they must appear at the start of the branches of atomic blocks, and come in two flavors: await e restricts execution until expression e holds true, with $x = e$ creates an immutable block-local variable named x bound to the value of e. (Effectful) statements come in several other flavors: $x := e$ assigns e to the process-local variable x; send(c, e) sends a message (the value of e) through the channel c, the FIFO mailbox of a process; skip simply does nothing; and print e outputs the value of e on some designated standard output.

4 The Join Calculus

The Join Calculus [8] is first a process calculus, but also a programming language [15, 6], based on the chemical reaction concurrency model [4, 7], featuring concurrent processes running on distant machines, transparent remote communication, as well as some mechanisms to handle failure. Throughout this section, we will use the usual chemical metaphor of the Join Calculus to describe our extended dialect.

Syntax. We give the full syntax of the extended dialect of the (Distributed) Join Calculus that we target in Figure 1. It corresponds to the Join Calculus with guarded reactions and name server operations (JoCaml's [6] `NS.register` and `NS.lookup` functions) baked in. *Atoms* $x\langle e_1, \ldots, e_n\rangle$ form the building blocks of the Join Calculus: they are messages to be consumed on the channel x, and the set $\{e_1, \ldots, e_n\}$

$$
\begin{array}{lll}
P ::= & x\langle e_1, ..., e_n\rangle & \text{Message} \\
| & P \mid P & \text{Composition} \\
| & \mathsf{def}\ D\ \mathsf{in}\ P & \text{Definition} \\
| & \mathbf{0} & \text{Inert process} \\
| & \mathsf{register}\ a\ \mathsf{as}\ e; P & \text{Registration} \\
| & \mathsf{let}\ a := \mathsf{lookup}\ e; P & \text{Atom lookup}
\end{array}
\qquad
\begin{array}{lll}
D ::= & J\ \mathsf{if}\ e \rhd P & \text{Local rule} \\
| & D\ \mathsf{or}\ D & \text{Co-definition} \\
J ::= & x\langle x_1, ..., x_n\rangle & \text{Message pattern} \\
| & J \mid J & \text{Join pattern}
\end{array}
$$

Figure 1: Formal syntax of the Join Calculus

forms the message's payload—expressions, in this case. *Molecules—processes* in the concurrent nomenclature—are composed of atoms, the inert process $\mathbf{0}$, reaction *definitions* $\mathsf{def}\ D\ \mathsf{in}\ P$, or *name server operations* for registration and lookup of atoms. In the chemical metaphor, molecules float around in some global solution—*the ether*—and may merge to form complex molecules $P_1 \mid ... \mid P_m$. Definitions $\mathsf{def}\ D\ \mathsf{in}\ P$ define ways to interact with the solution, basically the dynamic behavior of the program: when D contains a local rule $J\ \mathsf{if}\ e \rhd P'$, a molecule that matches the pattern J and that satisfies the guard e may be immediately consumed (albeit not necessarily) to form a new molecule P'. In contrast with other process calculi such as the π-calculus [16], channels are implicitly defined together with their reaction rules, rather than using a standalone ν construct. The set of variables present in each atom of a pattern J is called the set of *parameters*, or $rv(J)$ for *reaction variables*, of the reaction. The set of channel names defined by some D is called $dv(D)$ for *defined variables*.

Operational semantics. We then recall the operational semantics—the Reflexive Chemical Abstract Machine (RCHAM) [7]—of the Join Calculus. The semantics is defined by two types of computation steps: *reaction rules* $\rightarrow$ are the actual reduction rules of the Join Calculus; *heating rules* $\rightharpoonup$ are meta-computation rules that dictate how molecules interact with the solution, and are always reversible. We denote $\rightharpoondown$ for *cooling rules*, the converse of heating rules. In this presentation, since $\rightharpoonup$ and $\rightharpoondown$ are converse, we will define both $\rightharpoonup$ and $\rightharpoondown$ simultaneously using the $\rightleftharpoons$ symbol: heating rules are to be read from left to right, and cooling rules in the other direction.

In the distributed model of the Join Calculus, global solutions, denoted $\Gamma \Vdash \mathcal{S}$, are sets $\mathcal{S}$ of *local* solutions $\mathcal{D} \vdash_\alpha \mathcal{P}$ indexed by the name server Γ, where α is the *location* of the sub-solution, $\mathcal{D}$ is a multiset $\{\!|D_1, ..., D_n|\!\}$ of reaction definitions and $\mathcal{P}$ is a multiset $\{\!|P_1, ..., P_m|\!\}$ of processes. The location α of a local solution is a value that uniquely identifies it in the global solution: it may be a PID, an IPv4 address, a file path, *etc*. Computation rules are given in Figure 2 for local solutions and Figure 3 for global solutions. In this presentation, the context ($\mathcal{D}$, $\mathcal{P}$, $\mathcal{S}$ and/or Γ), if it does not change by effect of the rules, is left out to reduce unnecessary visual clutter. Multiset union is also replaced by a comma. For example, the verbose Str-Null rule is $\mathcal{D} \vdash_\alpha \mathcal{P} \cup \{\mathbf{0}\} \rightleftharpoons \mathcal{D} \vdash_\alpha \mathcal{P}$.

- Rules Str-Par and Str-Null represent spawning and ending processes.

$$
\begin{array}{rl}
\text{Str-Null} & \vdash_\alpha \mathbf{0} \rightleftharpoons \vdash_\alpha \\
\text{Str-Par} & \vdash_\alpha P_1 \mid P_2 \rightleftharpoons \vdash_\alpha P_1, P_2 \\
\text{Str-And} & D_1 \text{ or } D_2 \vdash_\alpha \rightleftharpoons D_1, D_2 \vdash_\alpha \\
\text{Loc-React} & J \text{ if } e \rhd P \vdash_\alpha J\sigma_{rv} \to J \text{ if } e \rhd P \vdash_\alpha P\sigma_{rv} \quad (\vdash e\sigma_{rv} \Downarrow \text{true})
\end{array}
$$

Figure 2: Operational semantics of local solutions.

$$
\begin{array}{ll}
\text{Str-Def} & \text{(if range}(\sigma_{dv}) \text{ fresh names)} \\
\quad \Vdash (\vdash_\alpha \text{def } J \text{ if } e \rhd P \text{ in } P') & \rightleftharpoons \quad \Vdash (J\sigma_{dv} \text{ if } e \rhd P\sigma_{dv} \vdash_\alpha P'\sigma_{dv}) \\
\text{Register} & \text{(if} \vdash e \Downarrow v \text{ and } v \notin \text{dom}(\Gamma)) \\
\quad \Gamma \Vdash (\vdash_\alpha \text{register } a \text{ as } e; P) & \to \quad \Gamma(v \leftarrow a) \Vdash (\vdash_\alpha P) \\
\text{Lookup} & \text{(if} \vdash e \Downarrow v \text{ and } \Gamma(v) = v') \\
\quad \Gamma \Vdash (\vdash_\alpha \text{let } x := \text{lookup } e; P) & \to \quad \Gamma \Vdash (\vdash_\alpha P[x/v']) \\
\text{Str-Comm} & \\
\quad \Vdash (\vdash_\alpha a\langle e_1, ..., e_n\rangle, \vdash_\beta) & \rightleftharpoons \quad \Vdash (\vdash_\alpha, \vdash_\beta a\langle e_1, ..., e_n\rangle)
\end{array}
$$

Figure 3: Operational semantics of global (distributed) solutions.

- Rule Str-Def "allocates" new fresh channels (using σ_{dv}, which substitutes $dv(J)$ for fresh names) and registers new definitions for these channels, while forking a new process.
- Loc-React consumes molecules that match some reaction's pattern, and spawns a new process with the parameters substituted, if the guard of the reaction is satisfied. σ_{rv} substitutes $rv(J)$ for concrete values.
- Register and Lookup respectively share an atom (by associating it to a given name, which should be known to other locations, *e.g.* a common token) and retrieve the shared distant atom.
- In the Str-Comm rule, any message is allowed to migrate between different sites. Since atoms cannot be matched upon in the definition of local reactions after being looked up, they are *unreactive* in all sub-solutions except the one they originate from (*i.e.* the site the atom is imported from). This is a generalization of the Join Calculus' comm rule, which only allows atoms to migrate to the sub-solution where their valences have been defined.

5 Compiling Network PlusCal into the Join Calculus

We define the compilation function $\mathcal{C}$ mapping each Network PlusCal process P to a Join Calculus process. Throughout this section, numbered boxes indicate subterms of the compiled output whose definition requires interleaving Join Calculus

syntax with mathematical notation. They are described in the surrounding text. For the sake of the presentation, we will write consecutive reaction definitions rather than a single or-separated one.

5.1 Compiling processes

Let $P := \text{self} \star x_1 = e_1, ..., x_n = e_n \star \{T_1\} \ ... \ \{T_m\}$ be a Network PlusCal process. In order to avoid dealing with substitutions in the expressions appearing throughout the Network PlusCal process, we first generate a mapping $\mathbb{V} : \{x_1, ..., x_n\} \to \mathcal{V}$ for some subset of variables $\mathcal{V}$ disjoint from $\{x_1, ..., x_n\}$. We will use these mappings, which we will denote as $\overline{x}$ for the variable x, as the name of the channels in charge of carrying the values of local variables in the processes.

State as atoms. A mutable Network PlusCal variable x holding a value v is modelled as a single *state atom* $\overline{x}\langle v \rangle$ floating in the solution. Every reaction that reads x must include $\overline{x}\langle v \rangle$ in its pattern, and re-emits $\overline{x}\langle v' \rangle$ in its body (possibly with an updated value v'). Not re-emitting such atoms would permanently consume variables, which is no equivalent in Network PlusCal. This is a standard approach in process calculi, sometimes referred to as a "box" or "reference cell". Atomicity of Network PlusCal blocks is enforced by the fact that exactly one atom $\overline{x}\langle v \rangle$ floats in the local solution for every variable x of P, and atoms cannot be consumed by more than one reaction simultaneously.

Process skeleton. P is compiled by $\mathcal{C}$ to the following Join Calculus process:

$$
\begin{array}{l}
\textsf{def } p\langle \text{self} \rangle \triangleright \\
\quad \textsf{def } \mathit{recv}\langle v \rangle \mid \overline{\mathit{inbox}}\langle vs \rangle \triangleright \overline{\mathit{inbox}}\langle vs :: v \rangle \ \textsf{in} \\
\quad \boxed{\quad \textcircled{1} \quad} \\
\quad \textsf{register } \mathit{recv} \textsf{ as } \text{"\{self\}"}; \\
\quad \overline{x_1}\langle e_1 \rangle \mid ... \mid \overline{x_n}\langle e_n \rangle \mid l_i\langle\rangle \mid ... \mid l_j\langle\rangle
\end{array}
$$

where *recv* is a channel dedicated to receiving messages from other actors. $inbox$ is a special variable introduced in Network PlusCal that serves as a buffer for incoming messages. Notice that each process defines its own *recv* atom. Although their names are the same, they are different in the chemical model: messages that are meant for other Network PlusCal processes to receive will only be received by them, as these atoms are unreactive in other processes' solutions.

Intuitively, state atoms are introduced to define the reactions corresponding to every atomic block of every thread of the process (in box ①, explained below), then initialized to their initial values by spawning them in the solution. An additional channel *recv* together with its reaction rule are also created to buffer incoming messages, which corresponds to the $T_{rx}(\mathit{mailbox} \to \mathit{inbox})$ thread in the Network PlusCal process. Each label in $\{l_i, ..., l_j\}$ is the first syntactic label of each thread of the process.

Processes are then expected to be run, possibly on different distant machines, by emitting the atom $p\langle\alpha\rangle$ with α the location of the process.

5.2 Compiling atomic blocks

Let $B \coloneqq l\colon \{G_1; S_1 \text{ ; goto } l_1\}$ or $\cdots$ or $\{G_n; S_n \text{ ; goto } l_n\}$ be an atomic block. Each branch $\{G_i; S_i \text{ ; goto } l_i\}$ is compiled to:

$$\text{def } l\langle\rangle \mid \overline{x_a}\langle x_a\rangle \mid \ldots \mid \overline{x_g}\langle x_g\rangle \text{ if } \boxed{②} \triangleright \boxed{③} \mid l_i\langle\rangle$$

The box ② basically contains the conjunction of all await guards in G_i, and the box ③ contains the molecule composed of:

- all state atoms consumed by the reaction, possibly with their carried value changed—the set of atoms $\{\overline{x_a}, \ldots, \overline{x_g}\}$ present in the reaction's pattern—, as dictated by the assignments in S_i;
- atoms $out\langle v\rangle$ for each print statement in S_i, where *out* is some reserved channel name meant to denote standard console output;
- let $send \coloneqq$ lookup $\alpha; send\langle e\rangle$ for every send$(c[\alpha], e)$ statement in S_i. Recall that each Network PlusCal process has a single unique mailbox, which is therefore able to uniquely identify each process. Rather than tracking the mailbox explicitly, we track the address of the process instead, which also uniquely identifies it.

The last $l_i\langle\rangle$ atom simply schedules the follow-up block, labelled l_i, for execution.

Notice that the atom $l\langle\rangle$ is consumed by all branches and never re-emitted in the solution, unless goto l is present in a branch. This ensures that only a single reaction may fire at once, thereby restricting execution to a single branch of the block at once. Jumping back to the block B, by spawning a new atom $l\langle\rangle$, simply allows all the branches to again be candidates for future reactions.

6 Example: Ping-pong

We illustrate the compiler on the two process definitions from a simple Ping-Pong protocol given in Figure 4. Given their obvious similarity, we only discuss the compilation of the process named Ping, and leave the translation of Pong as an exercise to the reader. We begin by generating reactions for each atomic block of the process. For *rcvPi*, since it only contains a single atomic branch, the following single reaction is introduced:

```
def rcvPi⟨⟩ | tmp1⟨tmp1⟩ | inbox_Ping⟨inbox_Ping⟩
     if Len(inbox_Ping) > 0 ∧ Head(inbox_Ping).mes = "Ping" ▷
  tmp1⟨Head(inbox_Ping)⟩ | inbox_Ping⟨Tail(inbox_Ping)⟩ | sndPo⟨⟩
```

```
--algorithm PingPong {
    fifos ping, pong[Pongs];

    (* @mailbox: ping; *)
    process (Ping = 0)
      variables
        tmp1 = [from ↦ self, mes ↦ ""],
        inbox_Ping = ⟨⟩;
    {
    rcvPi: await Len(inbox_Ping) > 0;
           await Head(inbox_Ping).mes = "Ping";
           tmp1 := Head(inbox_Ping);
           inbox_Ping := Tail(inbox_Ping);
           goto sndPo;
    sndPo: send(pong[tmp1.from], "Pong");
           goto rcvPi;
    }

    (* @rx: ping → inbox_Ping; *) {}

    (* @mailbox: pong[self]; *)
    process (Pong ∈ Pongs)
      variables
        tmp2 = "",
        inbox_Pong = ⟨⟩;
    {
    sndPi: send(ping, [from ↦ self,
                       mes ↦ "Ping"]);
           goto rcvPo;
    rcvPo: await Len(inbox_Pong) > 0;
           await Head(inbox_Pong) = "Pong";
           tmp2 := Head(inbox_Pong);
           inbox_Pong := Tail(inbox_Pong);
           goto sndPi;
    }
    (* @rx: pong[self] → inbox_Pong; *) {}
}
```

Figure 4: Ping-Pong algorithm in Network PlusCal syntax. T_{rx} threads are indicated by `@rx` annotations.

One can easily see that this reaction bears the same semantics as the original block: if `inbox_Ping` is not empty, and the projection `.mes` of its head is `"Ping"`, then `tmp1` becomes the head of the inbox, and the inbox becomes its own tail. Then, `sndPo` is scheduled for execution next.

A similarly structured reaction definition is generated for *sndPo*, although with fewer variables captured.

```
def sndPo⟨⟩ | tmp1⟨tmp1⟩ ▷
  let send := lookup "{tmp1.from}";
  send⟨"Pong"⟩ | tmp1⟨tmp1⟩ | rcvPi⟨⟩
```

Finally, the reaction for the entire process `Ping` is generated by combining the reactions generated above with the initialization of local variables and the T_{rx}-specific *send* channel.

```
def Ping⟨self⟩ ▷
  def recv⟨v⟩ | inbox_Ping⟨vs⟩ ▷ inbox_Ping⟨vs :: v⟩ in
  // def rcvPi & def sndPo --- see above
  register recv as "{self}";
  tmp1⟨[from ↦ self, mes ↦ ""]⟩ | inbox_Ping⟨[]⟩ | rcvPi⟨⟩
```

At the end of the definition of *Ping*, only *rcvPi* is scheduled for execution, as it is the very first label of the thread in `Ping`.

7 Conclusion and Future Work

We have presented a compilation scheme $\mathcal{C}$ from Network PlusCal to the Join Calculus. The scheme encodes local state as private atoms, and non-deterministic choice within blocks using a private channel (named after the block itself) and guarded reactions. The Ping-Pong example shows that the compiled terms are still readable and closely mirror the structure and semantics of the source Network PlusCal processes. Furthermore, atomicity of the Network PlusCal blocks is trivially handled by reaction definitions, making the Join Calculus a suitable target for a verified compiler of Distributed PlusCal algorithms.

Several directions remain open.

- An implementation of this compilation scheme, in Lean to follow the existing pipeline [3] from Distributed PlusCal to Network PlusCal, is left for future work.
- The correctness of $\mathcal{C}$ remains to be established. Intuitively, the values of all Network PlusCal variables are stored in state atoms at all points of the execution, and reactions are also fired atomically, which directly fits Distributed PlusCal's execution model. A formal proof, however, remains to be given.
- The Join Calculus term that is generated by $\mathcal{C}$ cannot be compiled by existing implementations, because of the use of guarded reactions. def J if $e \triangleright P$ can be encoded as def $J \triangleright$ if e then P else J (with if conditionals suitably encoded, as in JoCaml [6]). Although such encoding is correct, it is not satisfactory in terms of performance: reactions can fire prematurely even if the condition is false, which just reintroduces the consumed atoms, unchanged, back in the solution. A better implementation would rely on passive waiting instead, thereby reducing the performance cost of consuming and re-emitting atoms unchanged when guards reduce to false.

References

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[3] G. Bergeron, H. Cirstea, and S. Merz. Towards a verified compiler for Distributed PlusCal. In *11th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, 2025.

[4] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[5] H. Cirstea and S. Merz. Extending PlusCal for modeling distributed algorithms. In P. Herber and A. Wijs, editors, *18th International Conference on Integrated Formal Methods (iFM 2023)*, volume 14300. Springer, 2023.

[6] S. Conchon and F. Le Fessant. JoCaml: mobile agents for Objective-Caml. In *Proceedings. First and Third International Symposium on Agent Systems Applications, and Mobile Agents*, 1999.

[7] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1996.

[8] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*. Springer, 1996.

[9] F. Hackett, S. Hosseini, R. Costa, M. Do, and I. Beschastnikh. Compiling distributed system models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, 2023.

[10] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[11] M. Hristov and A. Bieniusa. Erla$^+$: Translating TLA$^+$ models into executable actor-based implementations. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang*. ACM, 2024.

[12] L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] L. Lamport. The PlusCal algorithm language. In M. Leucker and C. Morgan, editors, *6th Intl. Coll. Theor. Asp. Comp. (ICTAC 2009)*, volume 5684. Springer, 2009.

[14] L. Lamport. A PlusCal user's manual. `https://lamport.azurewebsites.net/tla/p-manual.pdf`, 2024.

[15] F. Le Fessant and L. Maranget. Compiling join-patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205–224, 1998. HLCL '98, 3rd International Workshop on High-Level Concurrent Languages (Satellite Workshop of CONCUR '98).

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[17] H. Plociniczak and S. Eisenbach. JErlang: Erlang with joins. In D. Clarke and G. Agha, editors, *Coordination Models and Languages*. Springer, 2010.