Erla$^+$: Translating TLA$^+$ Models into Executable Actor-Based Implementations

Marian Hristov
University of Kaiserslautern-Landau
Kaiserslautern, Germany
m_hristov17@cs.uni-kl.de

Annette Bieniusa
University of Kaiserslautern-Landau
Kaiserslautern, Germany
bieniusa@cs.uni-kl.de

Abstract

Distributed systems are notoriously difficult to design and implement correctly. Although formal methods provide rigorous approaches to verifying the adherence of a program to its specification, there still exists a gap between a formal model and implementation if the model and its implementation are only loosely coupled. Developers usually overcome this gap through manual effort, which may result in the introduction of unexpected bugs.

In this paper, we present *Erla$^+$*, a translator which automatically translates models written in a subset of the *PlusCal* language to *TLA$^+$* for formal reasoning and produces executable *Erlang* programs in one run. *Erla$^+$* additionally provides new PlusCal primitives for actor-style modeling, thus clearly separating the modeled system from its environment.

We show the expressivity and strengths of *Erla$^+$* by applying it to representative use cases such as a Raft-based key-value store. Our evaluation shows that the implementations generated by *Erla$^+$* offer competitive performance against manually written and optimized state-of-the-art libraries.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Source code generation**; • **Computer systems organization → Distributed architectures**; • **Theory of computation → Verification by model checking**.

Keywords: Verification, Distributed Systems, Model checking

ACM Reference Format:
Marian Hristov and Annette Bieniusa. 2024. Erla$^+$: Translating TLA$^+$ Models into Executable Actor-Based Implementations. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang (Erlang '24), September 2, 2024, Milan, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3677995.3678190

Erlang '24, September 2, 2024, Milan, Italy

ACM ISBN 979-8-4007-1098-8/24/09
https://doi.org/10.1145/3677995.3678190

1 Introduction

In the software industry, testing remains the standard method for ensuring software correctness. Both manual and automated testing techniques are widely used to detect and fix bugs before deployment. However, this approach has proven to be insufficient for complex, high-scaling distributed systems due to the infeasibility of covering all possible states a given program can reach. Therefore, system designers need to find alternative solutions for proving the correctness of their algorithms. In the Erlang ecosystem, a number of tools have been developed that explore how formal verification techniques such as model checking or theorem proving can be applied to support developers in implementing correct distributed algorithms [9, 10, 12]. Using these techniques to verify formal specifications of distributed systems enables developers to identify bugs early, preventing costly bug-fixing in later stages of development, saving effort and finances.

However, languages used for formal verification differ from languages used for implementing software due to their respective purposes and the types of properties they are designed to express. Formal models are typically more general and abstract. They describe *what* the system is supposed to do without going into specifics about *how* it should be done. Traditional programming languages, on the other hand, while powerful for implementation, are not equipped to express certain desirable properties. These languages lack built-in support for predicate logic and temporal logic, which are critical for specifying and reasoning about the correctness of systems, particularly in the context of distributed and concurrent systems. Developers bridge this gap between formal model and code typically through manual effort, which is error-prone. Ensuring that the implementation of a system corresponds to its formal model can be achieved by various means such as:

1. Test case generation from the formal model;
2. Comparison of the implementation's traces with the formal model's set of possible behaviors; or
3. Code generation of an executable implementation from the formal model.

In this work, we focus on the latter approach, also known as *model-driven development.*

Since the complexity of the translation is strongly dependent on the distance between the source and the target languages [27], choosing a specification language that is close to a software development language simplifies the translation and provides traceability. Due to their restricted use of mutable state, functional programming languages are a popular translation target [14, 26]. For this reason, we have selected *PlusCal* [24] and *Erlang* as the source and target languages, respectively. This decision was motivated by the ability to specify actor-style server processes in PlusCal [44]. Additionally, PlusCal can be transpiled to *TLA*$^+$ [23], which enables formal reasoning via the *TLC* model checker [43] [24]. Erlang is widely used in industry for building scalable distributed systems with high-reliability guarantees [16, 35, 40].

Our goal is to automatically translate verified formal distributed system models written in a subset of the PlusCal language into Erlang programs, which adhere to the formal model's behavior. To achieve this, we created a translator called *Erla*$^+$, which is built upon the original PlusCal translator [7], adding an Erlang translation functionality and extending the PlusCal language with additional PlusCal primitives for actor-based modeling. This extension enables a clear separation between the modeled system and its environment. It enables developers to express distributed algorithms close to their implementation but on a higher level of abstraction.

To summarize, this paper offers the following key contributions:

- ➔ An extension of PlusCal for actor-based modeling, which includes additional primitives for message-passing and error simulation (see Section 5).
- ➔ A source-to-source compiler called *Erla*$^+$ which automatically translates PlusCal specifications into TLA$^+$ models for formal reasoning and produces executable actor-based system implementations in Erlang.
- ➔ We show that implementations generated by *Erla*$^+$ offer competitive performance compared to production-level systems.

2 Background

Model Checking. System specifications can be written as a set of properties in temporal logic that the system model is expected to satisfy, covering both safety and liveness properties. Model checking is an automated formal verification technique used to verify that a model adheres to its specification. To this end, a model checker exhaustively explores the state space of the model. If a behavior is discovered that violates the specification, the model checker can generate a counterexample as output, illustrating the execution steps leading to the issue.

However, model checking has its limitations. As the complexity of the system increases, so does the number of possible states. This is especially the case for distributed systems, where concurrent process execution leads to an exponential growth in the interleavings of actions. Checking all possible interleavings makes it difficult to exhaustively explore every possible state of the system and can even turn out to be computationally infeasible.

To manage the large (and potentially infinite) state space, developers often need to impose certain restrictions or create an approximation of the model. These strategies make the verification process more manageable and feasible by reducing the computational complexity and focusing on the most critical parts of the system. These constraints, however, sometimes lead to weaker guarantees compared to theorem provers, which can handle infinite state spaces but require additional developer input during the verification process.

***TLA*$^+$.** TLA$^+$ [22] is a formal specification language based on the Temporal Logic of Actions (*TLA*) [23] for modeling concurrent systems. Using the *TLC* model checker [43], one can verify liveness and safety properties of TLA$^+$ models. TLA$^+$ models specify the system's behavior through multiple predicates that define the initial state and the transition relation. Importantly, transitions are interpreted as atomic steps of the system.

PlusCal. PlusCal is a formal specification language that can be transpiled to TLA$^+$, enabling model checking via the TLC model checker. While TLA$^+$ primarily focuses on action-based modeling for state machines, PlusCal resembles an imperative programming language.

A PlusCal model consists of multiple processes that are defined by a sequence of statements, including familiar control flow constructs like if-statements and while-loops. Processes run concurrently and can interact with each other through shared global variables.

An integral part of every PlusCal model are labels. Blocks of statements annotated with a label are treated as an atomic step, akin to a transition in TLA$^+$. The main purpose of labels is to indicate the granularity of concurrency: The utilization of more labels in a model allows for more interleaved behavior, which results in exponential growth in the state space explored by the model checker. Furthermore, labels can be used as control flow targets for goto-statements.

Macros are a construct for encapsulating reusable code segments. They offer a concise means of representing frequently used sequences of statements, allowing for more modular and concise specifications. After their definition, macros can be called anywhere within the PlusCal specification, dynamically expanding into the associated sequence of statements during compile time, similar to macros in the *C* language.

3 Related Work

Automated Verification. *PSync* [8] is a tool used for semi-automated verification, in which algorithms are divided into communication-closed rounds, enabling developers to

design asynchronous faulty system models as synchronous ones, without worrying about concurrency. An important feature of PSync is that its verification engine is able to check liveness properties in addition to safety properties. A similar approach is *pretend synchrony* [39], which transforms an asynchronous model to a semantically equivalent synchronous model so that verification in the *Floyd-Hoare* logic is possible. *Verdi* [42] provides a framework for verifying and extracting runnable distributed systems via *Coq* [3] libraries, in which various fault models that developers may need are included. In *Ironfleet*'s methodology [15] developers can prove that a formal *Dafny* [25] model corresponds to a practical distributed implementation. *Ivy* [32] is a system that can interactively verify safety of infinite-state systems. In particular, Ivy can be used to interactively generate inductive invariants for distributed protocols. *I4* [28] almost fully automates Ivy's methodology, minimizing the developer's involvement during the verification and invariant generation process.

While most of these approaches do not focus on generating implementations but rather on automating the verification process, they have nonetheless inspired the development of *Erla⁺*.

Model-Checking Erlang Implementations. *McErlang* [10] is a model checker specifically tailored for verifying the properties of Erlang programs. While McErlang can verify both liveness and safety properties, it can only do so when the abstract model's state space is finite. Another model-checking tool designed for detecting concurrency errors in Erlang programs is *Concuerror*. Concuerror systematically tests Erlang programs to identify concurrency-related issues such as race conditions and deadlocks. *Soter* [9] is a static analyzer and verifier that extracts abstract models called *Actor Communicating Systems* from Erlang programs and then uses a combination of static analysis and infinite-state model-checking to verify safety properties. With *Erla⁺*, we take the orthogonal approach of model checking an abstract model instead of an implementation.

Translation of Formal Models To Implementations. *PGo* [13] is a compiler tool-chain that proposes a domain-specific language, *MPCal*, which is a superset of the PlusCal language, and a compiler that translates MPCal into a formal TLA⁺ model and an implementation in Go. In contrast to PGo, *Erla⁺* allows for developers that are familiar with PlusCal to directly start writing formal models and generating implementations by only complying with a handful of restrictions, without needing to learn a new formal language. *TLA⁺ Transmutation* [29] presents an automatic translator of TLA⁺ models to executable Elixir programs. Whereas *Erla⁺* strives towards practical code, TLA⁺ *Transmutation*'s generated code is more abstract, staying true to TLA⁺'s transition-system-like structure.

The paper 'Teaching practical realistic verification of distributed algorithms in Erlang with TLA+' [44] by Peter Zeller, Annette Bieniusa, and Carla Ferreira reports the authors' experiences of teaching verification of Erlang programs using the TLA⁺ language in a distributed systems course. By translating various broadcast algorithms by hand, the authors found a close resemblance between the structure of PlusCal models and Erlang programs do exist, which served as an inspiration to further explore this path.

What distinguishes *Erla⁺* from the other tools discussed so far is its capability to automatically translate formal specifications into actor-based implementations. Actor-based systems, characterized by their inherent concurrency, fault tolerance, and scalability, present unique challenges that *Erla⁺* is designed to address. This approach differentiates *Erla⁺* from other tools like PGo, which do not specifically focus on the actor model.

Timed Rebeca [36] is an actor-based modeling language that integrates model-checking tools and provides formal semantics, as well as tools for modular verification and partial-order reduction. It extends the *Rebeca* language [37] by incorporating timing primitives to address aspects such as computation time and message delivery time and introduces checkpoint functions. These checkpoints serve as synchronization points in simulations and model checking, marking important events and initiating verification. Additionally, Timed Rebeca supports the automated translation of its models to Erlang, enabling the generated code to be used with the McErlang model checker for running simulations [20]. The checkpoint functions are utilized in defining McErlang's monitors to verify safety properties, thus enabling the verification process. While Timed Rebeca is specifically focused on timed event-driven systems, *Erla⁺* does not impose such constraints. Another key difference is that Timed Rebeca's translation to Erlang is driven by the goal of enabling model checking through the McErlang model checker, whereas *Erla⁺* has a broader focus on generating practical executable implementations.

4 Workflow

4.1 Overview

Figure 1 gives an overview of a typical workflow for *Erla⁺*. The developer starts by specifying a formal model of the distributed system in a subset of the PlusCal language (see Section 5). The resulting *.tla* file is then passed to the *Erla⁺* translator. The translator parses the PlusCal model and produces two artifacts: a TLA⁺ model and an Erlang implementation.

Developers can prove the correctness of their model by using the TLA⁺ model and a system specification (safety and liveness properties) as input for the TLC model checker [43].

The generated implementation utilizes a custom Erlang library, which handles process registration and implements

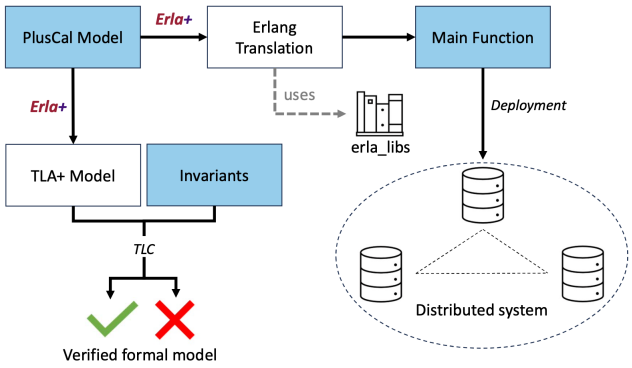

Figure 1. A visualization of *Erla*⁺'s workflow. Blue-colored items indicate where developer input is needed.

the behavior of *Erla*⁺'s primitives according to Erlang's semantics. The Erlang code does not require any manual adjustments and can be directly executed or integrated into an existing system.

4.2 Trusted Computing Base

In the domain of automated verification, every tool incorporates a Trusted Computing Base (TCB) as a fundamental component. The TCB comprises critical elements such as hardware, software, and firmware, which the tool relies on to ensure the accuracy and reliability of its verification processes.

In our approach, developers must place trust in several components. First, they must trust the correctness of the *Erla*⁺ compiler, specifically that the translation steps from PlusCal to Erlang and PlusCal to TLA⁺ are executed correctly. Secondly, developers must have confidence in the accuracy of the model checker (TLC) used in the verification process. Furthermore, the manually written custom Erlang library used in the generated implementations also falls within the TCB. Each of these components is integral to the trustworthiness of the automated verification process.

5 Modeling Language

Let us consider a simple semaphore protocol where a central server manages a lock, while several clients can concurrently request to acquire and release the lock. Figure 2 depicts the model of the semaphore server, written in traditional PlusCal.

Upon closer examination, particularly focusing on lines 9-11, 18-23, and 34-39, we can observe that a significant portion of the process' body is concerned with modeling the network, while only a small portion actually describes the server's behavior. Such mixing of a system model with its environment presents a challenge for a compiler, as they must decide what to translate and what to ignore during the translation process. What would make this decision easier, however, is having primitives that clearly express the intention of the programmer regarding interaction with the

```
1  variables queues = [id \in NodeSet |-> <<>>];
2
3  fair process Server = 0
4      variables msg = <<>>; q = <<>>; dest = 0;
5    begin
6      serverLoop:
7      while (TRUE) do
8          (* Receive message *)
9          await Len(queues[self]) # 0;
10         msg := Head(queues[self]);
11         queues[self] := Tail(queues[self]);
12         if (msg.type = "lock") then
13             if (q = <<>>) then
14                 (* q is empty *)
15             respondLock:
16                 (* Send response confirming lock
17                    to sender *)
18                 either
19                     dest := msg.sender;
20                     queues[dest] := Append(queues[dest],
                                        ↪ "grant");
21                 or
22                     skip;
23                 end either;
24             end if;
25         addToQ:
26             q := Append(q, msg.sender);
27         elsif (msg.type = "unlock") then
28             q := Tail(q);
29             if (q # <<>>) then
30                 (* q is not empty *)
31             respond_Unlock:
32                 (* Send response confirming lock
33                    to next in queue *)
34                 either
35                     dest := Head(q);
36                     queues[dest] := Append(queues[dest],
                                        ↪ "grant");
37                 or
38                     skip;
39                 end either;
40             end if;
41         end if;
42     end while;
43 end process;
```

Figure 2. Semaphore server model in PlusCal, inspired by PGo's locksvc model [13].

system environment. This would not only enable developers to concentrate on the logic and behavior of their distributed algorithms without becoming entangled in intricate network details, but, more importantly, provide clarity to the compiler regarding the semantics of the code, thus enabling a translation. Figure 3 shows the program written in *Erla*⁺'s modeling language.

```
1   variables queues = [id \in NodeSet |-> <<>>];
2
3   fair process Server = 0
4     variables msg = <<>>; q = <<>>;
5   begin
6     serverLoop:
7     while (TRUE) do
8         receive(msg);
9         if (msg.type = "lock") then
10            if (q = <<>>) then
11            respondLock:
12                send("grant", msg.sender);
13            end if;
14        addToQ:
15            q := Append(q, msg.sender);
16        elsif (msg.type = "unlock") then
17            q := Tail(q);
18            if (q # <<>>) then
19            respond_Unlock:
20                send("grant", Head(q));
21            end if;
22        end if;
23    end while;
24  end process;
```

Figure 3. Semaphore server model in *Erla⁺*, with message-passing primitives highlighted in red.

As explained in Section 4, *Erla⁺* translates models written in a subset of the PlusCal language. This subset includes familiar control-flow constructs (e.g. while loops and if-statements), complex data types (e.g. sets, sequences, and tuples), assertions, and more. Traditional PlusCal models, however, mix the behavior of a system with its environment, as previously discussed in this section. Therefore, to enable a translation from PlusCal models, it was necessary both to restrict and extend the PlusCal language that is accepted by *Erla⁺*. To this end, we extended *Erla⁺*'s language with additional *message-passing* primitives, i.e. send and receive operations, which provide developers with an intuitive means to model the communication between processes. Furthermore, the addition of these primitives alleviates the need for global variables. Global variables are not supported by *Erla⁺*, as our tool translates to an actor-based language, which implies that processes must not share memory.

Moreover, *Erla⁺* supports neither non-deterministic constructs (e.g. *either-or* statements) nor *goto*-statements. Non-deterministic constructs are typically used to describe a system's response to changes in the environment. In our aim to separate the system's behavior from that of its environment, we constrain developers by removing non-deterministic constructs from *Erla⁺*'s language. This urges developers to explicitly model the interaction between entities of the system, enabling the translation to Erlang. Goto-statements are most commonly used to model process crashes, which *Erla⁺* solves by providing a specific primitive, called *maybeFail*, expressing the idea that a process might crash at a particular point of the execution.

6 From Model To Code

To obtain an executable implementation from the PlusCal language, *Erla⁺* needs to handle a blend of TLA⁺ and PlusCal semantics in addition to the PlusCal language extension we discussed in Section 5. Next, we demonstrate our approach to addressing the challenges associated with the translation of TLA⁺'s set-theoretic data types, non-determinism, inter-process communication, and failures. To illustrate these translation strategies, we provide the Erlang translation of the client from the semaphore protocol in Section 5, shown in Figure 5. This example will serve as a practical demonstration of *Erla⁺*'s code generation techniques.

6.1 PlusCal Statements

While some PlusCal statements can be easily represented in Erlang, others demand a more complex strategy. We will now discuss selective aspects to justify some of the design decisions.

Processes, Variable Declarations and Assignments. Similar to PlusCal, *Erla⁺* incorporates variable declarations as an integral component of the process' state. In Erlang, process state is manifested through a record that encompasses all PlusCal variables along with their respective initial values. An example of this can be seen in Lines 1-4 of Figure 5 where the state of the semaphore client is depicted, comprising its id, its request, the response, and a boolean flag indicating whether it holds the lock.

Since Erlang adheres to *static single-assignment*, meaning each variable can only be assigned once, while PlusCal does not, *Erla⁺* handles reassignment by generating a new state record reflecting the updated value of the variable.

In addition, *Erla⁺* also supports *with*-statements, which are represented in Erlang as unique temporary-variable assignments. Consequently, variables created using this statement are, akin to PlusCal, not part of the processes' state.

Control-flow Statements, Macros, and Asserts. An integral part of most PlusCal models is *while*-loops. Due to the absence of while-loops in the Erlang language, *Erla⁺* produces a function utilizing recursion to replicate the same functionality.

If-statements are translated into Erlang's case-statements.

For the handling of macros, *Erla⁺* utilizes a strategy similar to that employed in the standard TLA⁺ code generation, which involves expanding the statements of macros within the Erlang code.

Assert-statements are idiomatically translated to equivalent statements within Erlang's own *assert* library (see Line 73 in Figure 5).

6.2 TLA⁺ Data Types and Expressions

We categorize TLA⁺ data types into two groups: primitive types and built-in types. Primitive types comprise booleans, integers, and strings, while built-in types include key-value mappings (records), sets, and sequences. Primitive TLA⁺ values and their operations are translated idiomatically using Erlang's arithmetic, boolean, and list expressions. Built-in types, on the other hand, require more work.

Key-value mappings are translated as *maps* along with their corresponding *get* and *put* operations. However, while PlusCal allows for nested map access, this can not be expressed using Erlang's built-in functions (*BIFs*). To address this, *Erla⁺* employs a function from our custom library that implements this behavior.

Sets and their operations are idiomatically translated, making use of the standard library's functions for union, intersection, subtraction, and other set operations. Set operations lacking counterparts (e.g. power-set construction) are translated in Erlang into calls to our custom library dedicated to sets.

We represent PlusCal's sequences as Erlang lists, preserving their heterogeneous nature. As for operations, these are matched with their equivalent counterpart in Erlang's standard library module for lists.

6.3 Model Values and Definitions

In TLA⁺, model values are special constants used to abstractly represent entities without specifying their internal structure. They help simplify models, keep the state space finite, and parameterize modules for reusability. For example, model values can represent different processes in a distributed system, allowing the specification and verification of their behaviors and interactions.

Model values are defined out of the PlusCal algorithm's scope. This means that these are not visible to our translator when compiling the PlusCal code. However, as we operate under the assumption that the TLA⁺ model has been model-checked and is thus correct, *Erla⁺* interprets unknown identifiers as constants, which we represent in Erlang as macros. The values of these constants are to be provided after translation in the corresponding Erlang macro declarations.

Definitions, on the other hand, can be defined in-scope, which enables translation of their values. Currently, only constant definitions but no function definitions are supported.

6.4 Message Passing

Next, we discuss how *Erla⁺* handles the message-passing communication primitives in the translations. The entire functionality is encapsulated within a module in our custom library named *erla_libs_comm*.

Enabling communication among nodes within a distributed Erlang system necessitates the adoption of a registration strategy. For this, we utilize Erlang's built-in *global* module

```
1   variables queues = [id \in NodeSet |-> <<>>];
2   define
3     ServerID == 0
4   end define;
                          ⋮
41  fair process Client \in ClientSet
42   variables
43    req = <<>>, resp = "", hasLock = FALSE;
44  begin
45  acquireLock:
46    req := [sender |-> self, type |-> "lock"];
47    (* send lock request to server *)
48    send(req, ServerID);
49  criticalSection:
50    (* receive response *)
51    receive(resp);
52    assert(resp = "grant");
53    hasLock := TRUE;
54  unlock:
55    req := [sender |-> self, type |-> "unlock"];
56    (* send unlock request *)
57    send(req, ServerID);
58    hasLock := FALSE;
59  end process;
```

Figure 4. Semaphore client model in *Erla⁺*.

to handle process registration and inter-process communication. The *global* module enables the registration of processes by unique names, which can be used by other processes in the distributed system for message addressing. Leveraging this functionality, *Erla⁺* can register processes by the same integer identifiers used in the PlusCal model. Hence, the generated processes can address each other using the corresponding PlusCal integer process identifiers.

Apart from process registration, our module *erla_libs_comm* provides the implementation of our PlusCal message-passing primitive, *send*. Like in the process registration, we utilize Erlang's standard library module *global* for the sending of messages. Regarding the reception of messages, these are intuitively translated into Erlang's signature *receive*-statement.

6.5 Crashes

As mentioned in Section 5, our translator provides the user with the primitive *maybeFail* to mark critical sections in which a process crash becomes observable. *Erla⁺* does not translate crashes in its generated implementations since developers typically anticipate and handle potential crashes within the formal model, rather than simulating them in the implementation itself.

7 TLA⁺ Code Generation

Since we have enhanced the PlusCal language with message-passing primitives and a primitive for failure simulation,

```
-record(state_Client, {
    id = -1,
    req = [], resp = "", hasLock = false
}).

-define(const_ServerID, 0).
                    ⋮
process_Client(State0) ->
    % Register process globally
    erla_libs_comm:register_proc(State0#state_Client.ID),
    % Body
    State1 = State0#state_Client{req =
      #{key_sender => State0#state_Client.id,
        key_type => "lock"}},
    erla_libs_comm:send(?const_ServerID,
      State1#state_Client.req),
    receive
      Message1 ->
        State2 = State0#state_Client{resp = Message1},
        ?assert(State2#state_Client.resp =:= "grant"),
        State3 = State2#state_Client{hasLock = true},
        State4 = State3#state_Client{req =
          #{key_sender => State3#state_Client.id,
            key_type => "unlock"}},
        erla_libs_comm:send(?const_ServerID,
          State4#state_Client.req),
        State5 = State4#state_Client{hasLock = false}
    end.

start_process_Client(Id) ->
    spawn_link(?MODULE, process_Client, [#state_Client{id = Id}]).
```

Figure 5. Semaphore client model translation in Erlang.

Erla+ must also translate these new features into TLA+ to enable model checking. All new additions to the PlusCal language are implemented as *macros*, as explored by the *Distributed PlusCal* project [1]. This approach allows *Erla*+ to leverage the already existing *PlusCal translator* [7] to produce correct TLA+ representations of our language extension. Figures 6 and 7 show the equivalent macro representations for *Erla*+'s PlusCal extension.

Despite Erlang providing various network guarantees, such as FIFO order and at most once delivery, there still remains a risk of messages being lost in transit within a distributed system [19, 38]. Even though Erlang's built-in features aim to mitigate these risks to a great extent, they do not entirely eliminate the possibility of message loss. Therefore, developers must be aware of this inherent risk and design their system with appropriate error handling and recovery mechanisms to ensure robustness in the face of message loss. To check these fault tolerance mechanisms, we introduce a lossy behavior in our *send* primitive, which employs an *either-or* statement, reflecting the unreliable nature of the network: either the message successfully reaches its destination (Line 6), or it is lost (Line 8).

The *receive* primitive exhibits a blocking behavior, similar to Erlang: when no message is available in its incoming message-queue, also known as mailbox, it waits until one arrives, assigns it to the parameter *Msg*, and removes it from the message queue.

As shown in Figure 6, both message-passing primitives assume the existence of a record named queues which models the process' mailboxes. While such a variable is mandatory, it is up to the user to define which processes should be part of the communication by initializing the record as a mapping of process identifiers (integers) to mailboxes (sequences).

Erla+'s failure generation primitive, *maybeFail()*, uses the *either* statement to model an unrecoverable non-deterministic crash, represented by the `await FALSE` expression in Line 4 of Figure 7. This behavior aligns with the *crash-stop* failure model[4], which we have settled on as our failure model. This model follows Erlang's "let it crash" philosophy [2], where the concept is that if a process crashes, it does not propagate the failure to other processes. However, unlike Erlang, where processes will be notified whenever a process they depend on crashes, our network model does not currently notify PlusCal processes in case of a crash. This aspect is left for future work.

8 Implementation

Erla+ is an open-source project; its implementation is available on GitHub [17]. The compiler is a fork of the original PlusCal translator [7] and is also implemented in Java. We leverage the existing Abstract Syntax Tree construction to produce Erlang code, while extending the TLA+ generation process with the translation of our PlusCal extension (see Section 5). Another advantage of using the existing TLA+ code base is the potential integration with the other TLA+ tools like the TLA+ toolbox [21] and the TLC model checker.

As mentioned in Section 7, we drew inspiration from the Distributed PlusCal project [1] and implemented the added primitives as macros. This approach allows us to take advantage of the existing PlusCal translation infrastructure, obtaining a correct TLA+ translation of our primitives "for free".

However, building upon the original PlusCal translator comes with its limitations. The original PlusCal translator is constrained to the scope of the PlusCal algorithm, meaning that anything defined outside its scope, such as model values and operators, is not visible for translation. As mentioned in Section 6.3, we work around this issue by interpreting unknown identifiers as constant values. This suffices for our approach, as the only out-of-scope constructs we support are model values and do not handle operators.

```
variables queues = [id \in NodeSet |-> <<>>];

macro Send(Msg, Dest)
begin
    either
        queues[Dest] := Append(queues[Dest], Msg)
    or
        skip;
    end either;
end macro;

macro Receive(Msg)
begin
    await queues[self] # 0;
    Msg := Head(queues[self]);
    queues[self] := Tail(queues[self]);
end macro;
```

Figure 6. Message-passing primitives represented as PlusCal macros.

```
macro MaybeFail()
begin
    either
        await FALSE;
    or
        skip;
    end either;
end macro;
```

Figure 7. The failure primitive, *maybeFail()*, represented as a PlusCal macro.

9 Evaluation

To show the expressiveness and applicability of *Erla*$^+$, we have successfully modeled and translated three systems, shown in Table 1. Our Raft model *Erla-Raft* is based on the authors' original TLA$^+$ specification [30] and PGo's MPCal model [13]. Expanding upon Erla-Raft, we developed a key-value store *Erla-RaftKVS* on top of it. During the modeling process, we deliberately favored simplicity and understandability over efficiency and complexity of our PlusCal systems. All experiments in this section, including model checking and performance benchmarking, were performed on an Apple M3 processor with 16 CPU cores and 64 GB of RAM, running macOS Sonoma 14.5.

The model checking results, including correctness properties, duration, and number of explored states, are listed in Table 1. Since TLC only terminates after exhaustively covering all possible states, we had to impose certain restrictions on our models to ensure termination. Our Raft-based models were limited to a cluster of three nodes, with one client sending a maximum of two requests and allowing for at most three leader elections (one per Raft node).

In our evaluation, we aim to assess how *Erla*$^+$'s generated implementations compare to state-of-the-art, efficient, manually written implementations in terms of performance. To this end, we evaluated the Erla-RaftKVS model translation generated by *Erla*$^+$ and a key-value store implementation based on *Ra* [33, 34], a Raft implementation in Erlang used in production by a leading company in the industry [40]. The evaluation was conducted using the YCSB benchmark [6] to measure throughput for five standard YCSB workloads, all of which use a Zipfian distribution:

(A) Update heavy: 50% read and 50% write
(B) Mostly read: 95% read and 5% write
(C) Read-only: 100% read
(D) Read latest: 95% read, 5% write
(F) Read-modify-write: 50% read and 50% read-modify-write

Note that workload E is not included, as the measured key-value store systems do not support range queries.

We extended the original YCSB codebase [5], written in Java, by adding custom *database clients* and custom Erlang clients. The database clients manage the communication and operations between the benchmark and the Erlang clients, which, in turn, facilitate communication with the key-value store systems. The code is available on Github [18].

We performed measurements for each workload by incrementally increasing the number of concurrent client threads generating the system load. Each combination was repeated five times for reliability. On the machine that we used to conduct the experiments, we reached maximum saturation for both systems at a total of 32 threads.

In Figure 8, we present the maximum throughput values we have measured during our experiments for a cluster of 3 Raft nodes across the standard YCSB workloads described above. Throughput is measured in operations per second (ops/sec). Overall, *Erla*$^+$'s Raft-based key-value store implementation demonstrates competitive performance compared to the state-of-the-art Raft implementation Ra-RaftKVS. In particular, for the read-intensive workloads B, C, and D, the gap between both key-value stores' throughput values is relatively minor. Notably, for workload D, Erla-RaftKVS manages to narrow the throughput gap to only 8% less than Ra-RaftKVS's throughput, achieving a maximum of 5 319 ops/sec. For the write-intensive workloads A and F, however, Ra-RaftKVS achieves a significantly higher maximum throughput: in workload F Ra-RaftKVS reaches 4 784 ops/sec at its peak, which is 27% more than Erla-RaftKVS's 3 496 ops/sec.

Table 2 compares the average throughput of Erla-RaftKVS and Ra-RaftKVS for a cluster consisting of 3 Raft nodes across a selection of different numbers of client threads, ranging from 2 to 128, using the standard YCSB workload A. For a low number of client threads, Erla-RaftKVS outperforms Ra-RaftKVS, demonstrating better efficiency in low-concurrency

Table 1. The systems that we successfully translated using *Erla⁺*: the semaphore protocol from Figure 3, the Raft consensus protocol, and a key-value store based on Raft. Note that details about model-checking for the Raft model are not listed, since they are entirely covered by Erla-RaftKVS.

System	PlusCal LOC	TLA⁺ LOC	Properties	Model checking duration (min.)	Num. of checked states	Erlang LOC
Semaphore protocol	46	88	Mutual exclusion and liveness	< 1	$2,21 \times 10^5$	91
Erla-Raft	285	645	-	-	-	447
Erla-RaftKVS	514	810	The 5 key Raft properties [31]	224	$2,9 \times 10^8$	591

Table 2. Average throughput (in ops/sec) comparison: for standard YCSB workload A across various numbers of client threads.

Systems	Num. of client threads						
	2	4	8	16	32	64	128
Erla-RaftKVS	3286	3934	4334	4261	3963	3567	3554
Ra-RaftKVS	1763	3000	4607	5455	5751	5656	4800

scenarios. However, as the number of client threads increases, Ra-RaftKVS starts to excel, showing a steady improvement in throughput and surpassing Erla-RaftKVS at higher thread counts. As the number of client threads increases to 4, 8, and 16, Erla-RaftKVS continues to show a steady increase in throughput, reaching up to 4334 ops/sec. However, beyond 16 client threads, Erla-RaftKVS's throughput starts to decline slightly. In contrast, Ra-RaftKVS shows a significant improvement in throughput with increasing client threads, reaching up to 5 751 ops/sec at 32 threads, and thus outperforming Erla-RaftKVS at higher thread counts. Beyond 32 client threads, both systems show a decline in throughput as the systems have reached their peak saturation.

We attribute Ra-RaftKVS's smaller throughput under low load to the stronger consistency guarantees provided by Ra [33]. Specifically, Ra-RaftKVS uses the function *consistent_query*, which guarantees that the query will return results containing at least all changes committed before the query is issued, and may also include changes committed while the query is running. To provide these strong consistency guarantees, the system must ensure that all changes are visible to the query through additional coordination mechanisms. This synchronization overhead results in increased latency and variability in response times. We suspect that under low load, the relative cost of maintaining these consistency guarantees becomes more significant, leading to reduced throughput as resources are diverted from processing queries to ensuring consistency.

Figure 9 provides a comparative analysis of the scalability of *Erla⁺*'s Raft-based key-value store translation and the Ra-based key-value store implementation across different cluster sizes using the standard YCSB workload A. As anticipated, the throughput of both systems decreases as the cluster size increases. However, Ra-RaftKVS demonstrates better scalability, maintaining higher throughput across all cluster sizes, indicating its superior optimization in multi-threading and coordination mechanisms compared to Erla-RaftKVS. As the cluster size increases from 3 to 9 nodes, Ra-RaftKVS's throughput decreases only by 11%, while Erla-RaftKVS's throughput decreases by 38%.

A potential reason for Erla-RaftKVS's larger regression in scalability is that its Raft nodes are single-threaded. This limitation could be addressed either on the model side or on the implementation side. On the model side, this would involve splitting the PlusCal processes into multiple processes, albeit considerably increasing the complexity of the model and its verification. Alternatively, on the implementation side, optimizing the code to leverage multi-threading could mitigate the scalability issues without adding significant complexity to the model, which, however, would lead to a divergence of the implementation from the model.

Additionally, we speculate that the usage of Erlang's *global* module in *Erla⁺*'s libraries might also influence Erla-RaftKVS's performance, as *global* commands are known to affect the scalability of distributed Erlang applications [11].

10 Conclusion

With our work, we target the gap between formal models and implementations when verifying distributed systems. We presented an extension of PlusCal for actor-based modeling and a compiler named *Erla⁺* that automatically translates models written in a subset of that language extension to TLA⁺ specifications for formal reasoning and produces executable Erlang implementations. As our evaluation shows, *Erla⁺* can be used to verify complex distributed systems, like a Raft-based key-value store, while generating idiomatic Erlang code. The generated implementations provide competitive performance for low to moderate workloads against fine-tuned state-of-the-art implementations.

In future work, we aim to enhance the reliability of the generated code by ensuring its adherence to the expected behaviors defined in the formal specifications. To this end, we plan to explore test-case generation from the model checker's explored state space, enabling more comprehensive verification of the implementations [29, 41]. By integrating such techniques, we hope to further bridge the gap between formal models and practical, robust implementations.

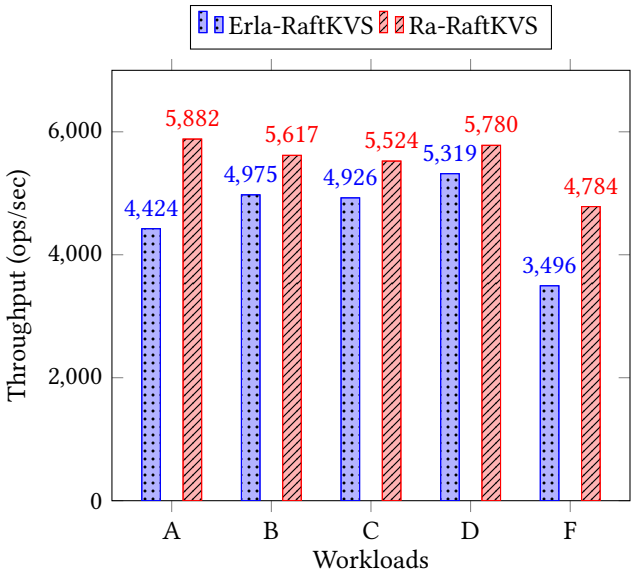

Figure 8. Maximum throughput comparison: *Erla*$^{+}$'s Raft key-value store translation vs. state-of-the art Raft implementation across various standard YCSB workloads.

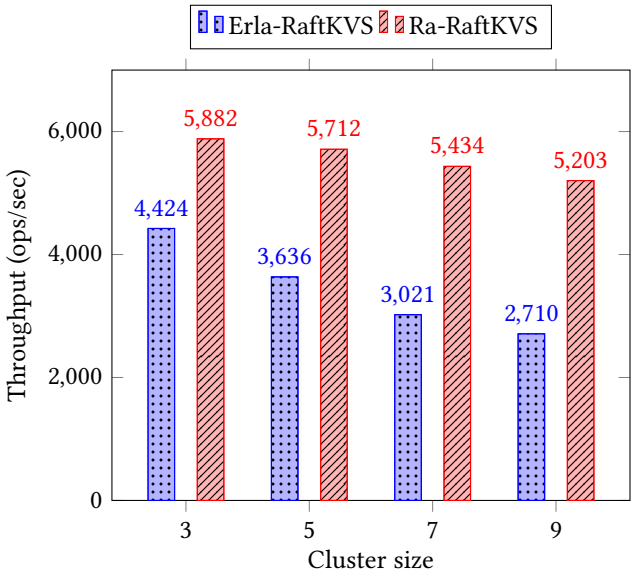

Figure 9. Scalability comparison: *Erla*$^{+}$'s Raft key-value store translation vs. state-of-the art raft implementation across varying cluster sizes for standard YCSB workload A.

References

[1] Heba Alkayed, Horatiu Cirstea, and Stephan Merz. 2020. An Extension of PlusCal for Modeling Distributed Algorithms. In *TLA+ Community Event 2020*.

[2] Joe Armstrong. 2010. erlang. *Commun. ACM* 53, 9 (2010), 68–75.

[3] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[4] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (second edition. ed.). Springer Nature, Berlin, Heidelberg.

[5] Brian F Cooper. 2019. *Yahoo! Cloud Serving Benchmark*. https://github.com/brianfrankcooper/YCSB (last accessed on 2024-05-15).

[6] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[7] Microsoft Corporation. [n. d.]. *tlaplus Repository*. https://github.com/tlaplus/tlaplus (last accessed on 2023-05-15).

[8] Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices* 51, 1 (2016), 400–415.

[9] Emanuele D'Osualdo, Jonathan Kochems, and C H Luke Ong. 2013. Automatic verification of Erlang-style concurrency. In *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings 20*. Springer, 454–476.

[10] Lars-Åke Fredlund and Hans Svensson. 2007. McErlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*. 125–136.

[11] Amir Ghaffari. 2014. Investigating the scalability limits of distributed Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*. 43–49.

[12] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. 2011. Test-driven development of concurrent programs using Concuerror. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. 51–61.

[13] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 159–175. https://doi.org/10.1145/3575693.3575695

[14] Florian Haftmann and Lukas Bulwahn. 2013. Code generation from Isabelle/HOL theories. *Part of the Isabelle documentation: http://isabelle. in. tum. de/dist/Isabelle2017/doc/codegen. pdf* (2013).

[15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 1–17.

[16] Fred Hebert. 2013. *Learn You Some Erlang for Great Good! A Beginner's Guide*. No Starch Press, USA.

[17] Marian Hristov. 2024. *Erla+ Repository*. https://github.com/mmhristov/Erlaplus

[18] Marian Hristov. 2024. *Yahoo! Cloud Serving Benchmark for Raft implementations in Erlang*. https://github.com/mmhristov/ycsb_raft

[19] John Högberg. 2021. *A few notes on message passing*. https://www.erlang.org/blog/message-passing/ (last accessed on 2024-05-15).

[20] Haukur Kristinsson, Ali Jafari, Ehsan Khamespanah, Brynjar Magnusson, and Marjan Sirjani. 2013. Analysing timed rebeca using mcerlang. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*. 25–36.

[21] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. 2019. The TLA+ toolbox. *arXiv preprint arXiv:1912.10633* (2019).

[22] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (may 1994), 872–923. https://doi.org/10.1145/177492.177726

[23] Leslie Lamport. 2002. Specifying systems: the TLA+ language and tools for hardware and software engineers. (2002).

[24] Leslie Lamport. 2009. The PlusCal Algorithm Language. In *Theoretical Aspects of Computing - ICTAC 2009*, Martin Leucker and Carroll Morgan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–60.

[25] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[26] Pierre Letouzey. 2008. Extraction in coq: An overview. In *Logic and Theory of Algorithms: 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008 Proceedings 4*. Springer, 359–369.

[27] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2017. From Clarity to Efficiency for Distributed Algorithms. *ACM Trans. Program. Lang. Syst.* 39, 3, Article 12 (may 2017), 41 pages. https://doi.org/10.1145/2994595

[28] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 370–384.

[29] Gabriela Moreira, Cristiano Vasconcellos, and Janine Kniess. 2022. Fully-Tested Code Generation from TLA+ Specifications. In *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing* (Uberlandia, Brazil) *(SAST '22)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/3559744.3559747

[30] Diego Ongaro. 2014. *TLA+ specification for the Raft consensus algorithm*. https://github.com/ongardie/raft.tla (last accessed on 2024-05-15).

[31] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[32] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 614–630.

[33] RabbitMQ. 2018. *Ra: A Raft Implementation for Erlang and Elixir*. https://github.com/rabbitmq/ra (last accessed on 2024-05-15).

[34] RabbitMQ. 2018. *Raft-based Key/Value Store*. https://github.com/rabbitmq/ra-kv-store (last accessed on 2024-05-15).

[35] Rick Reed. 2024. *Scaling to Millions of Simultaneous Connections*. http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf (last accessed on 2024-05-15).

[36] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. 2014. Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Science of Computer Programming* 89 (2014), 41–68.

[37] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S De Boer. 2004. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae* 63, 4 (2004), 385–410.

[38] Hans Svensson and Lars-Åke Fredlund. 2007. Programming distributed erlang applications: pitfalls and recipes. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop* (Freiburg, Germany) *(ERLANG '07)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/1292520.1292527

[39] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[40] VMWare. [n. d.]. *RabbitMQ*. https://www.rabbitmq.com (last accessed on 2023-05-15).

[41] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 127–143.

[42] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 357–368.

[43] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 54–66.

[44] Peter Zeller, Annette Bieniusa, and Carla Ferreira. 2020. Teaching Practical Realistic Verification of Distributed Algorithms in Erlang with TLA+. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang* (Virtual Event, USA) *(Erlang 2020)*. Association for Computing Machinery, New York, NY, USA, 14–23. https://doi.org/10.1145/3406085.3409009

Received 2024-05-30; accepted 2024-06-27