Compiling Distributed System Specifications into Implementations

by

Renato Mascarenhas Costa

B. Computer Science, University of São Paulo, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Compiling Distributed System Specifications into Implementations

submitted by **Renato Mascarenhas Costa** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Ivan Beschastnikh, Computer Science
Supervisor

Ronald Garcia, Computer Science
Supervisory Committee Member

Abstract

Distributed systems are notoriously difficult to get right: the inherently asynchronous nature of their execution gives rise to fundamentally non-deterministic systems. Previous work has shown that the use of formal methods to reason about distributed systems is a promising area of research. In particular, model-checking allows developers to verify a system model against a correctness specification by performing an exhaustive search over the system's possible executions. However, the transition from a trusted specification to a valid implementation is a manual, error-prone process that could introduce subtle, hard to find bugs.

In this work, we aim to bridge this gap by automatically translating a specification into an implementation that refines it. Specifically, we leverage the clear separation between application-specific logic and abstract components in the Modular PlusCal language to define an interface that concrete implementations must follow in order to implement the abstractions. Our evaluation shows that this approach handles complex specifications and generates systems that preserve the correctness properties verified via model-checking.

Extended version: May 23, 2019

Lay Summary

Distributed systems are at the heart of many online services that people use every day, such as email, file storage, and messaging. Despite this pervasiveness, distributed systems are complex to build and often fail in unexpected ways. These failures may lead to service outages and even data loss, impacting all users of the service. There has been increasing interest in the application of formal methods as an alternative to deal with the complexity of a distributed system. In particular, model checking is a technique that allows system designers to verify the correctness of a model by performing exhaustive testing. However, the transition from the model to a correct implementation is manual and can introduce bugs, even if the model is believed to be correct. In this work, we propose a technique to bridge this gap by automatically translating from a trusted model to a correct implementation of a distributed system.

Preface

All the work presented henceforth was conducted in the Networks, Systems and Security (NSS) lab in the Department of Computer Science at the University of British Columbia, Vancouver campus.

The work presented in this thesis is original and unpublished work by the author, Renato Mascarenhas Costa. It builds on top of the PGo compiler, a research project in which Matthew Do and Finn Hackett are also core contributors. The entirety of this work was designed and written with the assistance of Dr. Ivan Beschastnikh.

Table of Contents

List of Tables

List of Figures

Glossary

API application programming interface

AST abstract syntax tree

BMC bounded model checking

FIFO first-in, first-out

LOC Lines of code

RSM replicated state machine

SAT satisfiability

TLA Temporal Logic of Actions

Acknowledgments

This research was funded by a National Science and Engineering Research Council of Canada (NSERC) Discovery Grant.

I would like to thank Dr. Ivan Beschastnikh for his supervision and guidance during the development of this work. Our weekly meetings were very helpful in the process of defining the direction of the project.

I would also like to thank Matthew Do and Finn Hackett for their contributions to the project and the many hours we spent together discussing design decisions on a whiteboard.

Finally, I would like to thank my family and friends for their support during my time at UBC.

Chapter 1

Introduction

Engineering performant and correct distributed systems is a hard task, despite their ubiquitous presence in systems behind popular software services offered by companies like Google, Amazon, Netflix, and many others. This difficulty stems from the inherently concurrent and asynchronous environment in which they are deployed: processes execute independently from one another and on different machines; networks connecting these processes may fail, drop, or reorder packets; and, individual nodes may crash at any time. This dynamic setting gives rise to a fundamentally *non-deterministic* system, in which it is not possible to predict the exact sequence of events for a given input. As a consequence, distributed systems running in production today often have critical bugs that lead to degraded performance [12], service outage [3, 16] and even data loss [13].

Due to the asynchronous nature of distributed systems, protocol designers face the challenging task of reasoning about concurrent execution. Even when communication protocols are formally specified, they may make certain assumptions about the execution environment that do not hold in practical systems. This gap forces engineers to fill in the missing pieces and potentially introduce bugs in the process [6]. Developers writing production distributed systems resort to adding error handling code in an ad-hoc manner, making it more difficult to relate the implementation to the original protocol description; this has the undesirable consequence of making protocol changes significantly harder and more error-prone.

These difficulties have motivated the research community to look for and de-

vise a number of solutions. In particular, the use of formal verification methods in the context of distributed systems is an area of active research. For instance, *model-checking* is a technique that can be used to exhaustively explore a distributed system's model in order to find *violations* of correctness properties of the system. The main advantage of model checking is its ability to uncover bugs hiding in corner cases of a protocol or implementation, testing scenarios that manually written tests would generally not exercise [21]. In addition, model checkers are also able to provide a *trace* when a property is violated, making it easier for the developer to understand how the system reached an invalid state.

Many model checking languages and tools exist. TLA$^+$ [27] is a declarative specification language that allows developers to formalize the behavior of their systems as a collection of steps and transitions between them. PlusCal [28] is a specification language built on top of TLA$^+$ that makes it easier to specify systems in a procedural style: more specifically, users can specify how different processes in a system interact using familiar control flow constructs such as `if` statements and while-loops. Protocol designers using TLA$^+$ or PlusCal can write properties about their system and verify that they hold for every possible execution interleaving by using the TLC model checker. This gives them more confidence about the correctness of the protocol. Both TLA$^+$ and PlusCal, however, only describe a system *abstractly*: implementation details are intentionally not included in the specification.

Despite being a useful tool to find subtle bugs in distributed systems, model checking suffers from a major drawback: *state-space explosion*. Complex systems generally have very large state spaces that grow exponentially with respect to certain parameters of the model (for example, the number of replicas in a cluster). This is even more pronounced if a concrete implementation, rather than an abstract specification, is being explored since it has more states, many of which are irrelevant to the high level properties of the distributed protocol the developer is interested in. By checking only a model, the protocol designer is able to leave certain implementation details abstract, focusing exploration on protocol-specific behavior and reducing the state space. The obvious downside of this approach is that the process of writing an implementation of the protocol model that was checked is a manual, error-prone process and bugs could be introduced during translation.

In this work, we aim to bridge the gap between the design and verification of a distributed protocol specification and the construction of a correct implementation for the model. More specifically, we use a variant of PlusCal called Modular PlusCal (MPCal for short) to mechanize the translation from a verified model to an implementation that exhibits the same behavior. In other words, the implementation is a **refinement** of the model. This is achieved by exploiting Modular PlusCal's clear separation between the specification of a system's behavior and the implementation details that are modeled abstractly. By providing developers with concrete implementations of abstract components typically found in a distributed system specification (for instance, network models, file systems, time, etc), we are able to generate implementations of model-checked specifications that have performance comparable to handwritten systems. Figure 1.1 depicts the different stages of the process in this workflow.

In summary, this thesis makes the following contributions:

- The definition of a general application programming interface (API) that gives users the ability to create their own abstractions in Modular PlusCal and define how they map to concrete implementations.
- A compiler from Modular PlusCal specifications to implementations in the Go programming language, and a distributed execution environment that includes concrete implementations for common abstractions needed in distributed applications and protocols. This work was developed as part of the PGo [40] compiler[1].
- An evaluation of distributed systems generated from specifications, showing that they have performance comparable to equivalent handwritten systems in most cases (being 2x slower in the worst case observed) while drastically reducing the number of lines of code the developer needs to write.

[1] PGo is open source and is available at https://github.com/ubc-nss/pgo.

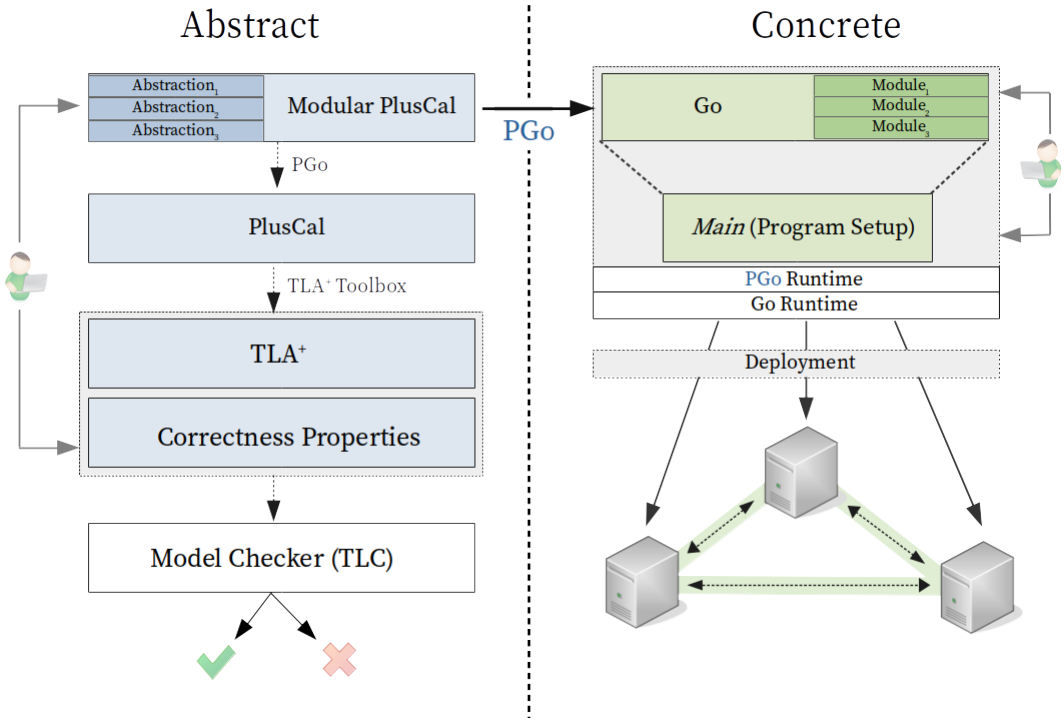

Figure 1.1: PGo workflow proposed in this thesis. Programmers specify their distributed protocols or applications using the Modular PlusCal language. PGo translates that specification into PlusCal, allowing developers to compile it to TLA$^+$ and model-check it against relevant system properties. Once the specification is believed to correctly describe the system, PGo can automatically generate an *implementation* that exhibits the same behavior as the abstractly defined specification. The developer provides concrete implementations for the components that were left abstract in the specification and is able to run the system in a distributed environment.

Chapter 2

Background

This work builds on top of multiple languages, tools and concepts previously designed and built to solve specific problems. In this chapter, we briefly outline the most important ideas this work builds on. Section 2.1 describes *model-checking*, a technique used to verify whether a model satisfies a certain correctness specification; sections 2.2 and 2.3, respectively, present an overview of TLA$^+$ and PlusCal, two specification languages that can be used to model arbitrary systems; section 2.4 describes Modular PlusCal, a variant of PlusCal used in this work. Finally, section 2.5 introduces PGo, a compiler from specifications to implementations. The work described in this thesis is part of the PGo compiler.

2.1 Model Checking

Model checking is a technique to algorithmically determine whether a *model* of a system satisfies a certain correctness *specification*. Models are typically expressed as a transition relation that defines the states that a system can be in, as well as how the transitions between them may occur. Model checking performs an exhaustive search over the state space of a system in order to determine whether the correctness specification is satisfied on every possible execution. When an execution is found to violate a correctness property, model checking has the ability to produce a *counterexample*, in the form of an execution trace, that informs the system designer how the system reached an invalid state.

Correctness specifications are written as properties about the system. In order to prove correctness of the model with respect to these properties, it is common to divide them into two categories: *safety* and *liveness* properties [24]. Informally speaking, safety asserts that bad states are not reachable from the initial state; liveness properties express that the system must eventually do something good. This categorization is useful because different techniques are used to prove each type of property, and previous work has shown that every property can be constructed as a conjunction of safety and liveness properties [2][1]. Figure 2.1 lists some examples of safety and liveness properties in the context of distributed systems.

A note on wording: it is common to refer to a model and correctness properties of a system as its "specification". Unless otherwise stated, this is the interpretation used in this thesis when referring to a specification of a system.

$$\forall j \in JobSet : (j \notin DOMAIN(AssignedJobs)) \vee (\exists n \in NodeSet : AssignedJobs(j) = n)$$

(a) Safety property: every job is either not scheduled yet, or assigned to a valid node in the cluster. This property must hold for every reachable state in the system.

$$\Diamond(\forall j \in JobSet : \exists n \in NodeSet : AssignedJobs(j) = n)$$

(b) Liveness property: eventually ($\Diamond$), all jobs must be assigned to a valid node in the cluster. This property talks about entire execution *traces* of the system, and not particular states. Note that satisfying the safety property above **does not** guarantee this fact, since a scheduler that never schedules any job trivially satisfies the safety requirement.

Figure 2.1: Examples of safety and liveness properties in the context of distributed job scheduling. In this model, `AssignedJobs` is a map from job to node identifiers; and `JobSet` and `NodeSet` represent the set of jobs to be scheduled and nodes in the cluster, respectively.

The biggest issue with model checking as a correctness verification technique is *state-space explosion*. Complex systems have state space that grows exponentially with respect to data-structure sizes and parameters of the model. This is exacerbated when execution is concurrent (such as in distributed systems): model

[1] As long as they are *trace properties*. Hyperproperties [7], able to express more complex behavior, are handled differently.

checking needs to explore every possible execution interleaving, further increasing the state space.

A complementary approach to traditional model checking is *bounded model checking (*BMC*)* [5]. In this approach, the transition relation is unfolded a fixed number of times k, conjoined into a single formula and the search for a violating trace is given to a satisfiability (SAT) solver. While this approach is useful for finding shallow bugs, it is generally incomplete.

Many techniques exist that aim to alleviate the problem of state-space explosion without bounding the search depth. *Partial-order reduction* [11, 14] groups transitions into equivalence classes and avoids exploring interleavings that do not matter to prove correctness. *Abstraction* can also significantly reduce the state space by treating certain aspects of a system abstract: instead of exploring all states associated with the steps required to perform a certain task, abstraction allows the system designer to express the same behavior as an abstract component. TLA$^+$ is one example of a specification language that allows systems to be specified at arbitrary levels of abstraction. This work aims to match the use of abstractions in model checking with concrete implementations that exhibit the same behavior, specifically in the context of distributed systems.

2.2 TLA$^+$

TLA$^+$ [27] is a declarative specification language designed by Leslie Lamport. Its main goal is to allow the specification of practical systems on top of a simple, yet rigorous mathematical foundation. TLA$^+$ is based on Lamport's Temporal Logic of Actions (TLA) [26], a variant of Pnueli's Temporal Logic [36] and uses first-order logic and set theory to formalize mathematics.

Typical specifications in TLA$^+$ consist of a definition of the system's *initial state* as well as the *transition relation* that describes the possible ways that the system can make progress. The transitions between states define *atomic steps* in the system and a sequence of them form a *behavior* of the system. TLA$^+$ was specifically designed to allow the description of what the system does in all its possible behaviors, as well as the verification of its functional properties. Other kinds of properties, such as average runtime performance or probabilistic guarantees, are

not supported by the underlying formalism.

TLA$^+$ has seen greater adoption in an industrial setting than most distributed system verification tools [10, 35]. Two important factors contribute to this success: its effectiveness at describing concurrent and asynchronous systems, very important for the specification of practical distributed systems; and the development of TLC, a model checker for the TLA$^+$ language that is capable of verifying both safety and liveness properties.

Figure 2.2 uses the distributed job scheduler example from Figure 2.1 and illustrates how distributed systems can be modeled using TLA$^+$ and how properties about the system are translated from the mathematical notation used previously.

MODULE *jobscheduler*

EXTENDS *Naturals*, *FiniteSets*

$FiniteNaturalSet(set) \triangleq IsFiniteSet(set) \land \forall\, e \in set : e \in Nat$

CONSTANTS *JobSet*, *NodeSet*
ASSUME $FiniteNaturalSet(JobSet) \land FiniteNaturalSet(NodeSet)$

CONSTANT *Unassigned*
ASSUME $Unassigned \notin NodeSet$

VARIABLES *AssignedJobs*, *LoadedJobs*

$Init \triangleq$ In the initial state, all jobs are unassigned
$\land\, AssignedJobs = [job \in JobSet \mapsto Unassigned]$
No jobs are loaded
$\land\, LoadedJobs = \langle\rangle$

$LoadJob \triangleq$ Load job from some source

$Assign(j) \triangleq$ Assign job j to a node in *NodeSet*

$Next \triangleq$ Load next job or schedule some loaded job
$\lor\, LoadJob$
$\lor\, \exists\, job \in LoadedJobs : Assign(job)$

Safety property
$SafeAssignment \triangleq \forall\, j \in JobSet : \lor\, AssignedJobs[j] = Unassigned$
$\lor\, \exists\, n \in NodeSet : AssignedJobs[j] = n$

Liveness property
$EventuallyAssigned \triangleq \Diamond(\forall\, j \in JobSet : \exists\, n \in NodeSet : AssignedJobs[j] = n)$

Figure 2.2: Definition of the distributed job scheduler example in TLA$^+$ (comments displayed in grey background). To make the example more interesting, the scheduler here is modeled to load jobs from some source and schedule them concurrently (expressed as a disjunction in the definition of `Next`). The safety and liveness properties of Figure 2.1 are redefined here using TLA$^+$ syntax.

One important characteristic of TLA$^+$ is that it allows the description of some behavior both in terms of *what* a component is supposed to do and *how* that task is accomplished. In Figure 2.2, for example, we only model what the job scheduler does, leaving other details abstract (for instance, how jobs are loaded). These powerful notions of **abstraction** and **refinement** enable a TLA$^+$ specification to describe a system at different levels of detail and is at the core of the formalism.

2.3 PlusCal

PlusCal [29] is another specification language designed by Leslie Lamport. It is an algorithm description language that enables correctness verification. PlusCal can be compiled to TLA$^+$, has precise semantics and, due to its integration with TLA$^+$, allows the designer to specify a system at different levels of abstraction.

PlusCal changes the way that specifications are written from a declarative to a procedural style in order to make it easier for engineers without a strong background in mathematics or formal verification to start specifying their systems without a lot of training. Listing 2.1 uses PlusCal to model the previous job scheduler example.

```
1  --algorithm JobScheduler {
2    variables AssignedJobs = [job \in JobSet |-> Unassigned],
3              LoadedJobs = << >>;
4
5    macro Assign(job) {
6        (* assigns a job to a node *)
7        with (node \in NodeSet) { AssignedJobs[job] := node };
8    }
9
10   macro LoadJob() {
11       (* loads a new job; not included for simplicity *)
12   }
13
14   process (Scheduler = 0) {
15       schedule:
16         either { LoadJob() }
17         or     { with (job \in LoadedJobs) { Assign(job) } };
18   }
19 }
```

Listing 2.1: Job scheduler in the PlusCal language.

PlusCal requires the user to structure a system being specified around *labels* (such as `schedule` on Listing 2.1, line 15). The statements within a label are the atomic steps in the model. Just like in TLA$^+$, there is a tradeoff to be made with respect to the use of labels in a model: the more concurrency (in the form of interleavings) is allowed in the model, the closer it will be to a concrete environment; however, this comes at the cost of exponential growth of the state space, substantially increasing the model-checking time. It is up to the system designer to decide how granular these atomic steps should be to meaningfully capture a system's behavior.

2.4 Modular PlusCal

Modular PlusCal is a variant of the PlusCal language developed as part of the PGo project (see section 2.5). The language inherits PlusCal's syntax, but adds several constructs to enable the separation between the description of the system and its abstract components. Modular PlusCal can be compiled to PlusCal, which in turn can be translated to TLA$^+$ and model checked against system-specific properties.

Modular PlusCal introduces three important concepts:

Archetypes define the components of the system being specified. They are isolated from one another and can only interact with the outside environment (for instance, other archetypes) by making use of parameters passed to them. Since these parameters typically represent the execution environment, they are called *resources* in Modular PlusCal.

Mapping macros allow the system designer to specify how the parameters passed to archetypes are abstractly modeled. They are expanded when the specification is compiled to PlusCal.

Instances specify how archetypes are mapped to PlusCal processes for model-checking purposes. Archetypes are given abstract representations of their execution environment for every parameter declared in their definition, as well as which mapping macros should be applied, if any.

In PlusCal, algorithm variables have global scope: every process in the system

```
1  --mpcal JobScheduler {
2    mapping macro FIFOChannel {
3        read {
4            await Len($variable) > 0;
5            with (msg = Head($variable)) {
6                $variable := Tail($variable);
7                yield msg;
8            };
9        }
10
11       write {
12           await Len($variable) < BUFFER_SIZE;
13           yield Append($variable, $value);
14       }
15   }
16
17   archetype JobScheduler(ref connections)
18   variables j, targetNode; (* local variables *)
19   {
20       schedule:
21           (* schedules a job 'j' to a 'targetNode', omitted for simplicity *)
22       sendJob:
23           connections[targetNode] := j;
24   }
25
26   variables AssignedJobs = [job \in JobSet |-> Unassigned],
27             (* incoming messages for scheduler: initially, no messages *)
28             network = [job \in NodeSet |-> << >>];
29
30   (* create an instance of the JobScheduler archetype with ID 0 *)
31   process (Scheduler = 0) == instance JobScheduler(network)
32       mapping network[_] via FIFOChannel;
33 }
```

Listing 2.2: Sending a job to an assigned node in Modular PlusCal.

can read and update their values; this is, in fact, the only way to express communication across two different PlusCal processes. Modular PlusCal, however, takes a different approach: archetypes do not have access to global state and can only manifest externally visible changes by interacting with the parameters they are passed. This separation makes it easier to use different communication abstractions without significantly changing the system's logic.

Listing 2.2 illustrates the concepts introduced in this section by having our job schedulers communicate with one another over a network that is abstractly modeled as a first-in, first-out (FIFO) sequence of messages (`FIFOChannel` mapping macro, lines 2-15). It also uses a number of features of Modular PlusCal:

- Mapping macros can define behavior of parameters passed to archetypes when they are *read* (lines 3-9) and when they are *written* (lines 11-14).
- Mapping macros have access to two special variables: `$variable` and `$value`. The former is expanded to the variable being mapped when the archetype is instantiated (lines 31-32); the latter is the expression being assigned to a resource (for example, `j` in line 23).
- Parameters passed to archetypes can be *mapped as functions* (`network[_]` syntax on line 32). This distinction between archetype resources that are mapped as functions and those that are not becomes relevant in the code generation process, as discussed in section 3.2.1

Together, archetypes, mapping macros and instances enable the complete separation between the description of system behavior and implementation-specific components that are intentionally left abstract in model checking. These primitives also encourage a more modular specification construction by allowing abstractions to be shared and reused across models. In this work, we leverage the clear separation between the system and its execution environment to generate efficient implementations of distributed systems specified in Modular PlusCal.

For a more comprehensive description of Modular PlusCal, consult the language manual[2].

2.5 PGo

PGo [40] is a source-to-source compiler from PlusCal specifications to Go implementations. It was developed with the same goal as this work: automatically generating an implementation for a model-checked specification. However, PGo was initially focused only on concurrent systems and did not support the generation of distributed systems.

While the idea of using PlusCal to generate implementations of distributed systems sounds promising, it presents a number of challenges in practice. First, PlusCal requires that processes communicate by making updates to globally visible

[2] https://github.com/UBC-NSS/pgo/blob/v0.1.4/manual.pdf

data structures. In order to maintain PlusCal semantics, a concrete implementation would need to provide the same globally accessible, distributed data structures, incurring a prohibitive performance penalty. In addition, system designers typically wish to leave certain implementation-specific details abstract. For instance, it is common to model an operation that can fail by using the `either` keyword (introducing a non-deterministic choice), leaving details of what the operation is and when it can fail unspecified. The consequence of this mismatch is that the developer is then required to manually edit the generated source code to provide a concrete implementation for components that were left abstract in the specification. This process is error-prone and is precisely what PGo proposes to reduce (or eliminate).

This work is part of the PGo compiler and adds support to the compilation of specifications written in Modular PlusCal to Go implementations, focused on the challenges of compiling distributed systems.

Chapter 3

Compiling Modular PlusCal Specifications

This chapter describes the challenges faced and techniques used in the process of compiling distributed systems specified in Modular PlusCal into correct implementations of the model in the Go programming language. Section 3.1 introduces what this process means by describing how valid behaviors of a system specified using Modular PlusCal are derived by the model checker; section 3.2 continues the discussion by detailing the strategy used in this work to maintain the same execution semantics in the compiled implementation in Go. Section 3.3 describes how archetypes can be seen as the API that can be used to interact with the system being specified. Section 3.4 discusses the role of resources in the compilation process: how they conform to a well-defined API, and challenges faced when resources are mapped as functions. Section 3.5 introduces a common pattern in modeling long running processes in PlusCal, and how it can be accommodated in the design proposed in this work. Finally, section 3.6 discusses how failures can be dealt with if the specification does not account for them.

3.1 Modular PlusCal Execution Semantics

In the workflow proposed in this work, verifying whether a model described in Modular PlusCal satisfies a set of user-defined correctness properties requires it to

be model-checked. This process involves compiling the specification to PlusCal and translating that output to TLA$^+$, which is the language being model-checked by TLC (see diagram in Figure 1.1). The algorithm used compute all behaviors expressed in a model (documented in [27]) is, therefore, important to this work: in order to consider the generated implementation correct with respect to the specification, we need to ensure that all executions are possible behaviors defined by the specification.

To motivate the discussion, consider the specification in Listing 3.1. It describes how a producer adds jobs to a shared queue and consumers read jobs from the queue and process them somehow. Since archetypes can only make externally visible changes by performing operations on parameters passed to them, the queue is passed to both the `Producer` and `Consumer` archetypes.

Modular PlusCal is an extension of PlusCal and inherits the same execution semantics [28]. In summary, behaviors of a specification are derived as follows:

- *Labels* represent atomic steps (or *actions*) in the specification. The model checker will only perform one atomic step at a time when exploring the state space. Execution of an action consists of the evaluation of all statements from that label to the next. Modular PlusCal enforces the same labeling rules defined by PlusCal, which are documented in the language manual [30].

- For an action to be scheduled (and become an event in the behavior being explored), it needs to be *enabled*. One scenario where a step is not enabled is if it contains an `await` statement with a predicate that evaluates to `FALSE`. For example, in Listing 3.1, the operations that the consumer performs on the queue are defined by the `AbstractQueue` mapping macro (lines 40-41). Since the `doWork` action (line 25) reads a job from the queue (line 26), it will **not** be enabled if the queue is empty (line 4).

- All PlusCal processes run "concurrently". This means that the model checker will explore every possible interleaving of the atomic steps (labels) defined in every process. In the context of Modular PlusCal, this implies that every instance created in the model executes concurrently.

```
--mpcal Queue {
  mapping macro AbstractQueue {
      read {
          await Len($variable) > 0;
          with (job = Head($variable)) {
              $variable := Tail($variable);
              yield job;
          }
      }

      write {
          await Len($variable) < QUEUE_SIZE;
          yield Append($variable, $value);
      }
  }

  mapping macro AbstractJob {
      read  { yield 0; }
      write { yield $value }
  }

  archetype Consumer(queue)
  variable job;
  {
      doWork:
        job := queue;
        (* do some work with 'job' *)
  }

  (* parameters need to be annotated with 'ref' if assigned *)
  archetype Producer(ref queue, jobs) {
      enqueue:
        while (TRUE) {
            queue := jobs;
        }
  }

  variable q = << >>,
            job_stream;

  process (ConsumerP = 0) == instance Consumer(q)
      mapping q via AbstractQueue;

  process (ProducerP = 1) == instance Producer(ref q, job_stream)
      mapping q via AbstractQueue
      mapping job_stream via AbstractJob;
}
```

Listing 3.1: A queue specification in Modular PlusCal

3.2 Preserving Execution Semantics

From the rules outlined in section 3.1, we can conclude that a naïve but correct implementation of the system could rely on a runtime scheduler to decide the order in which actions in the Modular PlusCal specification execute. This, however, comes at a prohibitive performance cost, especially in the context of distributed systems: no concurrency is allowed across different archetypes, even if they are deployed separately and perform local computation that is not externally visible.

To allow concurrent execution of atomic steps defined in the Modular PlusCal specification, we let the **resources** dictate the safety of concurrent execution. This separates the discussion into two kinds of actions:

Labels that do not use any resource. Since resources are the only way an archetype can interact with its environment, an action that does not make use of any resource performs only local computation and is, therefore, safe to run concurrently with other atomic steps.

Labels that use one or more resources. These are labels that potentially interact with the environment. To ensure that execution semantics are preserved, the implementation generated by PGo first *acquires* access to the resources used in the label before the statements in the label are evaluated. An archetype resource that requires exclusive access, then, can make sure that other processes attempting to use the same resource will block until the resource becomes available again. We further refine this case by analyzing whether the resource is being used in *read-only* mode, a property that can be statically determined from the Modular PlusCal specification.

Resource acquisition has semantics that are specific to the environment they represent: for instance, a resource that encapsulates shared state may use locks to ensure that access is exclusive. Resources are acquired at the start of every label. We analyze the specification and determine which resources are used in each action, as well as the permissions necessary (read-only or read-write). The resulting implementation, then, first acquires all resources that are used in the atomic step, performs the operations defined in it, and releases the resources. However, this process is unsafe: suppose two actions, A_1 and A_2, use resources r_1 and r_2, which

require exclusive access. Also, suppose that concurrent execution of A_1 and A_2 leads to the following sequence of events:

- A_1 acquires exclusive access to r_1.
- A_2 acquires exclusive access to r_2.
- A_1 attempts to acquire exclusive access to r_2; it blocks because A_2 already has access to it.
- A_2 attempts to acquire exclusive access to r_1; it blocks because A_1 already has access to it.

In other words, acquiring all resources at the start of an action can lead to *deadlocks*. For this process to be safe, resources need to be acquired in a *consistent order*: if r_1 and r_2 are used in two different actions, they need to be acquired always as either $\langle r_1, r_2 \rangle$ or $\langle r_2, r_1 \rangle$. This is ensured by enforcing that concrete implementations of archetype resources are comparable to one another.

While we do not provide a proof that the concurrent execution model presented here actually preserves the execution semantics of Modular PlusCal and the TLC model checker, we sketch an informal reduction [33] argument, as has been done in previous work [18]. Consider two actions running concurrently in an implementation of a Modular PlusCal specification. There are three cases to consider:

1. *One of the labels does not use resources.* This case is always safe to be executed concurrently. Since local computation is not externally visible, we argue that concurrent execution is equivalent to an execution where all of the statements involved in the local computation step happen before (or after) the other action's statements. Both event orderings should have been explored by TLC and are, therefore, possible behaviors of the system.
2. *The labels use disjoint sets of resources.* This case is always safe to be executed concurrently. Since the actions interact with disjoint parts of the environment (represented by the resources they use), concurrent execution is equivalent to an execution where all statements in one label are evaluated before (or after) the statements in the other. As with the previous scenario, the model checker should have explored both possibilities.

3. *The labels use overlapping sets of resources.* This case is not always safe to be executed concurrently. However, with the strategy described above where resources are first acquired before they are used, execution will run one of the actions to completion first, and only then execute the statements in the other; this sequential execution is similar to the execution model used by TLC.

3.2.1 Resources Mapped as Functions

Archetype resources in Modular PlusCal can be mapped as functions (see section 2.4 for an example). If a resource `r` is mapped as a function, it will only be expanded (according to the abstraction defined in the mapping macro) if it is used as a function (i.e., `r[expr]`). This is useful, for example, if an archetype parameter represents connections to a set of nodes (the number of which is a parameter of the model) and, as a system designer, you intend to express the sending of a message to a particular node in that set of connections.

Resources mapped as functions, however, pose a challenge to the compilation strategy described in section 3.2. In particular, it becomes no longer possible to acquire all resources used in an action before it executes because the argument passed to the function may not be known when control reaches the start of the atomic step. For example, consider the specification in Listing 2.2, where the `network` resource passed to `JobScheduler` is mapped as a function. In the `sendJob` action, the job *j* is sent to a `targetNode`; if the value of `targetNode` changes in the action before it is used as an argument to `connections`, acquiring all resources beforehand would lead to incorrect behavior. Listing 3.2 illustrates the issue.

```
1 sendJob:
2     targetNode := targetNode + 1;
3     connections[targetNode] := j;
```

Listing 3.2: An example where it is not possible for an implementation to acquire access to all resources before an action starts. *targetNode* is mutated before it is used as an argument to *connections*; using its value for resource acquisition before the action starts leads to incorrect behavior.

Algorithm 1: Action Execution

```
actionCompleted = FALSE;
while ¬ actionCompleted do
    Acquire all resources not mapped as functions in consistent order;
    ... action statements ...;
    /* A resource mapped as function is required at
       this point                                          */
    error = Acquire(resourceMappedAsFunction);
    if error ≠ NULL then
        Abort all resources acquired so far;
        if ShouldRetry(err) then
            continue;
        end
        return error
    end
    ... more action statements ...;
    /* End of action                                       */
    Release all acquired resources;
    actionCompleted = TRUE;
end
```

Figure 3.1: Executing an action that uses resources mapped as functions.

Due to the impossibility of acquiring all resources that are mapped as functions before an atomic step begins, we resort to a best-effort approach. Resources mapped as functions are acquired *at the time of use*; this guarantees that the values of the arguments used are accurate. In addition, the implementation we generate keeps track of all resources previously acquired in each step to make sure that no resource is acquired twice in the same action. This approach, however, can lead to **deadlock**: since resources are acquired at the time of use, we can no longer enforce a consistent order as described in 3.2. To circumvent this issue, we allow resource acquisition to fail and, when a failure is detected, we abort any changes made to the environment and try again. Figure 3.1 specifies how an action in Modular PlusCal is translated to a concrete implementation in light of the discussion presented here.

Since an action has no externally visible interactions with the environment before it completes, and the changes proposed here to support resources mapped as

functions never leave the boundaries of a single action, we argue that the the execution model described in Figure 3.1 does not impact the reduction argument made in section 3.2.

3.3 Archetypes as APIs

Modular PlusCal uses archetypes to describe the behavior of components in a system; therefore, they can be seen as defining execution entrypoints of the system being designed. For this reason, the implementations we compile transform each archetype instantiated in the Modular PlusCal specification into a Go function. Listing 3.3 shows the signatures of the resulting functions when the queue specification in 3.1 is compiled.

```
1  package queue
2  import "pgo/distsys"
3  var QUEUE_SIZE int
4  func init() {
5          QUEUE_SIZE = 10
6  }
7  func Consumer(self int, queue distsys.ArchetypeResource) error {
8          // body
9  }
10 func Producer(self int,
11                 queue distsys.ArchetypeResource,
12                 jobs distsys.ArchetypeResource) error {
13         // body
14 }
```

Listing 3.3: Function signatures in Go when the specification in Listing 3.1 is compiled with PGo.

A few points are worth highlighting in this transition:

- The compiled code lives in a separate package (called `queue` in Listing 3.3). The developer wishing to use the the system compiled by PGo can import this package and call the functions defined there, passing a concrete implementation for every resource.
- TLA$^+$ constants (parameters of the model) used in the specification are given a concrete value when the user compiles their specification. All constants

are assigned in the `init` function, which the Go runtime ensures will be executed exactly once if the package is imported.

- Every function compiled for an archetype needs to be passed `self` as the first argument. This is an implicit variable in PlusCal specifications that contains the process identifier.
- For every parameter declared in the archetype definition, a corresponding `ArchetypeResource` is expected in Go. Archetype resources have a well defined interface (see section 3.4). It is up to the caller to ensure that the concrete implementation passed in has the same semantics as the abstraction used in the specification. The PGo runtime provides a number of commonly used components in distributed systems, such as communication channels, access to the file system, time, among others; their implementation is the focus of Chapter 4.
- Functions derived from archetypes return errors. These errors are used to indicate when the specification reached a state that cannot be mapped to a state in the specification. For example, the `AbstractQueue` mapping macro of Listing 3.1 does not specify what happens if the queue cannot be accessed. However, a practical implementation may require that the queue be stored across multiple servers, and a network failure could render the queue unavailable. In these scenarios, the error is returned to the caller, that is then responsible for determining what to do in response. Fault tolerance in this setting is discussed in section 3.6.

Once the specification is compiled to a Go package, the user is required to write the program's *main* function. In general, this will involve writing "glue" code that sets up the connections between processes and creates the concrete implementations for archetype resources. Most of the difficulties of this process, however, are alleviated by the execution runtime provided by PGo.

3.4 Resources as Interfaces

Listing 3.1 defines two archetypes, `Consumer` and `Producer`. Both the queue and the jobs that are submitted to it are abstractly modeled in the specification; the

system designer can still write and verify properties about how jobs are processed without having to specify lower-level details of the queue. An implementation of this specification, however, would need to inject a concrete implementation for a queue and a stream of jobs to be processed. These could come from another server, the file system, or another source: as long as the semantics provided by the concrete implementation match the abstraction used in the specification, the system's behavior is preserved.

In order for the Modular PlusCal execution semantics to be preserved as described in the previous section, however, the implementation of resources must satisfy a set of functional requirements. For that reason, we view archetype resources as implementations of a well-defined interface. Resources and resources mapped as functions defined as Go interface types are provided in Listing 3.4.

```
1  type ArchetypeResource interface {
2          Acquire(access ResourceAccess) error
3          Read() (interface{}, error)
4          Write(value interface{}) error
5          Release() error
6          Abort() error
7          Less(other ArchetypeResource) bool
8  }
9
10 type ArchetypeResourceCollection interface {
11         Get(value interface{}) ArchetypeResource
12 }
```

Listing 3.4: Type definitions for archetype resources

The Go language considers that a type implements a certain interface if it defines the same set of functions with matching signatures. A valid implementation of an archetype resource, however, has to obey stronger rules. The precise meaning and expectations for every function defined in the `ArchetypeResource` interface in Listing 3.4 are described as follows:

Acquire is invoked when access to a resource is required in the execution of an action. In the case of a resource mapped as a function, it is called immediately before the statement that uses it. `Acquire` takes as parameter a `ResourceAccess` struct that defines how the resource is used in the action (for instance, read-only or read and write).

Read is called when the archetype resource is used within an expression. The return type of this function may be relevant depending on the expression in which it is used: if the Modular PlusCal specification uses a resource `r` in the expression `r + 1`, then the return of the `Read` function will be cast to an integer in the code generated by PGo. It is up to the archetype resource implementation to make sure that the types returned by this function are compatible with the expectations of the caller.

Write is invoked when an archetype resource is used on the left side of a Modular PlusCal assignment; the value being assigned corresponds to the `value` argument that this function receives. If `Write` has to perform type-specific operations, it needs to cast the received value to the appropriate type. It is up to the resource implementation to ensure that casting is safe and compatible with how the resource is used in the specification.

Release is called on all resources that are used in an action that is executed to completion. This is the *only* function in an archetype resource implementation that is allowed to perform externally visible changes. After `Release` returns, the caller no longer holds access to the resource.

Abort is used when interactions performed with a resource, in the form of `Read` and `Write` calls, need to be aborted. As defined in the algorithm of Figure 3.1, an action is retried if a resource mapped as function fails to be acquired: in that case, all resources previously acquired in the atomic step up to that point are aborted before the action is restarted. After `Abort` returns, the caller no longer holds access to the resource.

Less defines how the archetype resource can be compared with another resource. This allows PGo to generate code that acquires resources in a consistent order, as discussed in section 3.2. The semantics of `Less` are defined by the `sort` package of the Go language[1].

Resources mapped as functions are seen as a map from a certain domain (specific to the environment they describe) to implementations of concrete resources.

[1] https://golang.org/pkg/sort

They are defined as implementations of the `ArchetypeResourceCollection` interface. The **Get** method is invoked when `resource[expr]` is used in a statement: it is given as argument the result of evaluating the expression and is expected to return an implementation of an archetype resource. In addition, implementations should ensure that if `Get` is called multiple times with the same argument, it should return a reference to the same resource implementation (and not copies of it).

3.4.1 Handling Errors

With the exception of `Less`, all functions that must be implemented by archetype resources may return errors (Listing 3.4). To ensure that PGo generates correct implementations, these functions can return two types of errors: `AbortRetryError` and `ResourceInternalError`.

Archetype resource implementations may choose to request the current action to be aborted: this avoids the deadlock issue described in section 3.2.1. When another failure happens during the processing of an operation, the implementation can also return *internal errors*: these represent unrecoverable situations and are propagated back to the caller. Internal errors can be seen as the implementation reaching states that are not part of any behavior in the specification. Table 3.1 describes some examples of the two different kinds of errors.

Error	Examples
`AbortRetryError`	Locking a resource that is already locked; reading a network message when none was received; sending a message to a node that has no buffer space to process it.
`ResourceInternalError`	I/O error when reading a file or socket; timeout when performing a network operation.

Table 3.1: Possible errors an archetype resource implementation may return.

3.5 Long Running Processes

A common idiom when specifying how different components in a system interact concurrently is to express them as long running PlusCal processes. This is especially important in the context of distributed systems: application designers are interested in verifying that certain correctness properties are maintained when the execution of multiple roles in the system (clients, servers, leaders, replicas) is concurrent. In Listing 3.1, for example, the `Producer` process runs without termination and keeps pushing jobs to the queue.

A more interesting case is illustrated in the definition of the `Consumer` process in the same specification. The consumer reads the next job to be processed in the queue, does some work, and terminates. While this might be the behavior intended by the system designer specifying an API to interact with this system, it causes subsequent jobs pushed by the producer to not be processed if all consumers terminated. This need to model a system that is constantly pushing new jobs and working on them concurrently, coupled with the fact that PlusCal does not support "dynamic" process spawning leads to processes like `Consumer` to also be defined as infinite loops.

This potential mismatch, however, can be unified using the notion of archetype resources presented in this work. If the `Consumer` archetype is changed to be executed in a loop that is conditioned by an argument passed to it, the developer is able to have an infinite loop for model checking purposes, and still be able to control the execution of the resulting function as if it were a regular, one-off call. Listing 3.5 describes the proposed change.

```
1  archetype Consumer(queue, startSignal)
2  variable job;
3  {
4      consumerLoop:
5        while (startSignal) {
6          doWork:
7            job := queue;
8            (* do some work with 'job' *)
9        }
10 }
11
12 instance (ConsumerP = 1) == instance Consumer(queue, TRUE)
```

```
13      mapping queue via AbstractQueue;
```

Listing 3.5: Updating the definition of the *Consumer* archetype to run in a loop: the consumer processes jobs as long as a *signal* sent to it is true.

When using the code generated by PGo for the consumer process as defined in the previous listing, the developer can call the `Consumer` function generated by the compiler in the background, passing as `startSignal` an implementation that blocks when attempted to be acquired. When the user actually wishes to process a job in the queue, the resource unblocks and returns `true`, allowing execution to continue. Fortunately, the Go language makes this straightforward: functions can be executed in the background with Go routines, and channels have the exact semantics described here and can be used as a concrete implementation of the `startSignal` parameter.

3.6 Dealing with Failures

The possibility of failure is one of the reasons that make writing correct distributed systems challenging. In most cases, the system needs to detect when a failure happens and perform some recovery procedure to continue making progress. In addition, different systems can choose to support different *failure models*: for instance, a protocol may be designed to accommodate crash-stop failures, but not Byzantine failures [31].

Systems compiled by PGo are as fault-tolerant as the specification itself: in particular, if the system uses abstractions that model the possibility of failure and the system is designed to tolerate them, the generated implementation should be able to present the same behavior, provided the archetype resource implementations exhibit the same semantics as the abstraction. Consider Listing 3.1 and suppose the user is interested in writing a property that states that every job pushed by the producer is eventually processed by the consumer in the correct way. While this property may hold for the specification in Modular PlusCal, it would not hold in a concrete implementation if the consumer node were to crash, for example; note that if a similar event happened in the specification, the liveness property stated before would be violated.

Providing language-level support for fault-tolerance is currently future work in

this project. However, the proposed design and execution model leaves a few options for users that intend to design fault-tolerant systems. We discuss two possibilities to handle failures in the current design: dealing with failures when interacting with the environment; and recovering from crash-stop failures.

3.6.1 Recovering from Errors in the Environment

As described in section 3.3, the functions used to interact with an archetype can return errors; when these indicate an error situation that is not part of the abstraction in the specification, the error is returned to the program's *main* function, which is manually written by the user.

Consider the definition of the `Producer` archetype in Listing 3.1, and suppose that the implementation of the queue passed to it issues a network call to push jobs to a replicated object store. A user could recover from failures of this remote store by inspecting any errors returned by the corresponding function, choosing a different replica when a failure is detected, and restarting the function. Listing 3.6 illustrates how this could be achieved in Go.

```
1  queueObj := RemoteQueueAt(master)
2  jobStream := NewJobStream()
3
4  // queue.Producer should run indefinitely unless an error occurs
5  for {
6          err = queue.Producer(queueObj, jobStream)
7          switch err.cause.(type) {
8          case *object_store.MasterTimeout:
9                  master = electNewMaster()
10                 queueObj = RemoteQueueAt(master)
11
12         case *job_stream.IOError:
13                 // deal with job stream error
14         }
15 }
```

Listing 3.6: Recovering from a failure by restarting

The pattern illustrated here can be used to detect multiple kinds of errors and recover from them appropriately, as the previous code snippet shows: a failure in the job stream can also be detected and potentially recovered from, depending on the semantics of the implementation passed to the `Producer` function. To

make this process easier, the implementation generated by PGo will make sure that, when an error that is returned to the user occurs, all resources previously acquired are aborted before the function returns; this guarantees that no externally visible behavior is observed if the failure happens while executing a statement halfway in an atomic step. Despite this fact, the system should be prepared to be restarted at arbitrary points: while this may be the case for some specifications, PGo requires that the user design the system for that possibility.

3.6.2 Crash Failures

Many reasons could lead to a system crashing: hardware faults, power failures, bugs in the software stack (operating system, programming language runtime, an archetype resource implementation), among others. While entirely supporting crash recovery in a system requires the specification to be aware of the fact that applications may stop at arbitrary points in time, the user is capable of performing recovery under certain circumstances using the application design proposed in this work.

One way to add fail-stop tolerance to a specification that does not model crashes is to hide their occurrence completely. If the crashed process can be restarted in a different machine where it left off, the failure can become invisible to the other processes in the system. To illustrate this mechanism, consider how a user may specify their system to capture local state (Listing 3.7) and allow it to be recovered.

```
1  archetype Producer(ref queue, jobs, ref context)
2  variable nextJob;
3  {
4      producerLoop:
5        while (TRUE) {
6            readJob:
7              if (context.action = "readJob") {
8                  nextJob := jobs;
9                  context := [action |-> "processJob", job |-> nextJob];
10             }
11           processJob:
12               if (context.action = "processJob") {
13                   (* process job in 'context.job' *)
14               }
15       }
```

```
16  }
```

Listing 3.7: Capturing local state in a context object in order to make it accessible in the event of a crash.

By having execution be dictated by a *context*, the user is able to start computation at arbitrary steps in the definition of the `Producer` archetype. Specifically, if the implementation of the context object passed to the producer updates the state in a persistent store, an external monitor process can detect failures in the producer node and start another instance of the same function. Since the *context* is stored in a different source, it can be read by the new instance, and execution may resume.

Note that while the approach above is sufficient to handle fail-stop failures in most circumstances, it still requires careful, manual error handling from the developer. In particular, if the crash happens during the process of *releasing* resources, the system could be left with a partial state change leading to behavior that is not part of the model-checked specification. In this scenario, the programmer is required to correct these inconsistencies before attempting a restart. While the execution runtime could help in the recovery process by providing stronger transaction semantics with a write-ahead log implementation [34], this is currently not supported.

Chapter 4

Execution Runtime

In this chapter, we discuss the runtime support provided by PGo to applications that were compiled from Modular PlusCal specifications. In section 4.1, we describe the protocol used to make sure all components of the system are connected and start execution at the same time. PGo also provides some implementations of commonly used resources in distributed applications: distributed global state can be easily used as a resource (section 4.2); and direct communication channels are also provided as a convenience to applications based on message-passing (section 4.3). Finally, section 4.4 provides an overview of other resources provided by the PGo runtime and how they can be used by system designers when compiling their specifications into executable implementations.

4.1 Synchronized Start

When a Modular PlusCal specification is compiled to TLA$^+$ and model-checked, all enabled states are scheduled infinitely often if they remain enabled (assuming a weak fairness model; for a longer discussion of fairness in TLA$^+$, see [27]). In an effort to replicate this behavior in the context of a distributed system, we provide tools for the developer to enforce a *distributed barrier*, allowing all components of the system to start at the same time. For example, in the context of the queue specified in Listing 3.1, synchronization could enforce that the producer does not inundate the queue with jobs before consumers are ready to process them.

PGo provides an implementation of a distributed barrier that also makes sure that every component is connected to each other. Figure 4.1 shows one execution of the protocol to illustrate how it works.

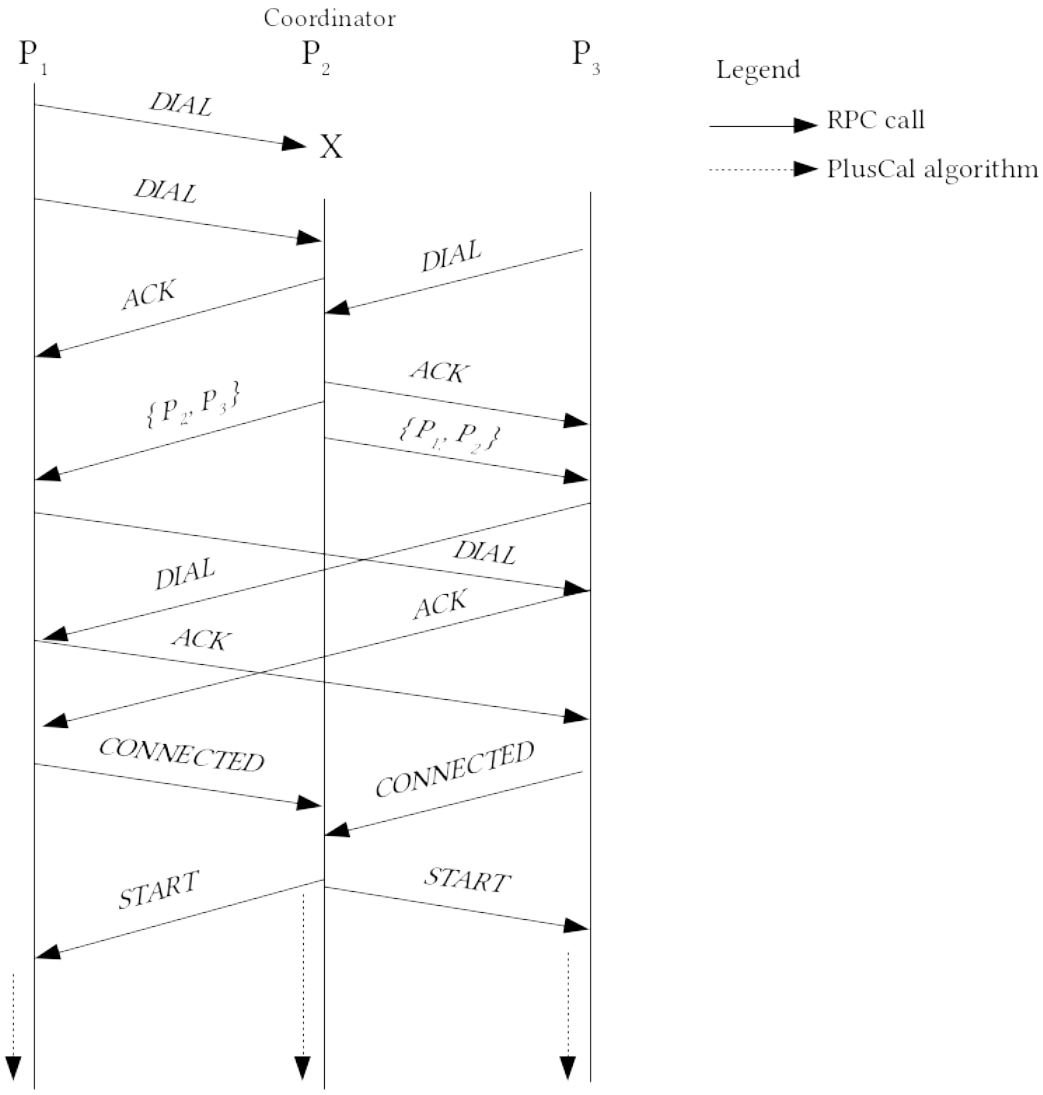

Figure 4.1: Synchronization protocol provided by the PGo runtime to allow applications to optionally coordinate their start.

As used here, a *process* loosely corresponds to a component in the system being designed. Since archetypes define the API that developers use to interact with the system, there is no one-to-one correspondence between their definitions and processes: there could be multiple archetypes associated with the same component, likely executing in the same host when deployed. Processes are, in general, deployed on different hosts, although that is not a requirement of the model. In the context of the queue specification of Listing 3.1, processes P_1, P_2 and P_3 in the diagram of Figure 4.1 could be `Consumer(0)`, `Consumer(1)` and `Producer(2)`, respectively (numbers in parentheses are the process identifiers; the implicit `self` variable in PlusCal specifications).

One arbitrary process in the system is assigned the role of *coordinator* of the synchronization protocol. The coordinator process does not need to be the first process to be executed: in Figure 4.1, for example, process P_2 is chosen to be the coordinator, but P_1 is the one that comes online first. When it tries to reach the coordinator, the call fails; processes keep trying to reach the coordinator until they eventually succeed. Once every process reported that they have connected to every other node in the system, the coordinator is responsible for informing them that they may start execution of their respective functions.

The messages exchanged during the synchronization protocol are briefly described below:

`DIAL`. When processes are created, they attempt to send a `DIAL` message to the coordinator, including the network address where they are deployed. As described previously, a failure in this process causes processes to keep trying to contact the coordinator until they finally succeed. The coordinator acknowledges `DIAL` messages, and processes then wait for a message from the coordinator.

{ P_1, ..., P_n }. As soon as the coordinator receives `DIAL` messages from every process, it sends a list of network addresses where every other process in the system is active. This list is aggregated from the `DIAL` messages that the coordinator received from every process.

`CONNNECTED`. When a process receives the list of other processes in the system

from the coordinator, it establishes a connection with them. When that is completed, a `CONNECTED` message is sent to the coordinator to indicate that the process is ready to start execution any time.

`START`. When the coordinator received `CONNECTED` messages from every process, it is then responsible for informing the processes that they may start execution of their functions in the system. The coordinator sends a `START` message to every process to express that intent.

4.2 Distributed Global State

As described in section 2.4, communication between processes in PlusCal happens via updates to global state; while Modular PlusCal makes it easier to use different communication abstractions, relying on global state can be more convenient for some applications. For this reason, PGo provides an implementation of distributed, shared state that developers can use as archetype resources in implementations. This component can be especially useful if the system has already been specified in PlusCal, where communication is achieved via updates on global state, but the system designer wishes to translate it to a Modular PlusCal specification in order to more clearly separate implementation-specific concerns from protocol definition.

The implementation of global state in PGo assigns an *owner* to every object stored; however, object ownership is a dynamic property. Nodes may become owners of objects they use (if, for example, a series of operations will be performed on them over a short period of time) and ownership can be controlled via application-specific policies. The design and implementation is similar to, and inspired by, previous work in distributed object systems and programming languages [20, 34].

4.2.1 Data Store

Global state in the system is stored in each node's *data store*. A node's data store represents the global state information that each node knows about at an arbitrary point in time. In particular, it includes the *name* associated with a portion of the state; the believed *owner* of that variable: i.e., the node that knows its current value; and the *value* of the variable. Figure 4.2 illustrates a snapshot of the data stores in

Name	Value	Owner
counter	`42`	P_1
queue	`nil`	P_3
history	`nil`	P_2

(a) P_1

Name	Value	Owner
counter	`nil`	P_3
queue	`nil`	P_3
history	`["e1", "e2"]`	P_2

(b) P_2

Name	Value	Owner
counter	`nil`	P_1
queue	`["j1", "j2"]`	P_3
history	`nil`	P_1

(c) P_3

Figure 4.2: State of the data stores in every node in a system at a certain point in time. A value is only associated with a global variable if the node owns the data. Ownership information may be outdated: for instance, P_3 believes that *history* is owned by P_1 when it is, in fact, owned by P_2 at this moment.

a system with three processes and three global variables.

The most important aspects of how the data store is managed are summarized below:

A process is aware of the data it owns. There is always at most one process that believes it owns a variable of the global state. More specifically, for every variable in the global state, there is either exactly one process where the *owner* column of the corresponding data store entry points to itself; or ownership is being transferred to another node. In the period it takes for the ownership transfer message to reach the new owner, no node owns the data.

No data is kept for state that is not owned. If a node does not own a variable in the global state, it keeps no value associated with it (even if it owned the value before), avoiding inconsistency and unnecessary memory use. If an operation on an object that is not owned is required, the caller needs to obtain a reference from the owner before the operation can be performed.

Ownership information may be outdated. When the ownership of an object is *moved* from one process to another, the update is not visible to other processes that were not involved in the exchange. For this reason, ownership information in the data store may be outdated; in particular, processes that receive requests for a reference to variables that they no longer own respond with the node that is believed to own the variable at that moment (that is,

their current *owner* entry for the variable in their data store). The requester then updates its own data store and repeats the process until it finally gets a reference to the object.

Global state can hold data of arbitrary types. The types of the values associated with each variable in the global state are not maintained by the data store. Instead, these values are serialized and need to be converted to their appropriate types by the caller (if they wish to perform operations on them). In practice, the implementation relies on the `gob`[1] serialization format, and supports the same sets of types and values.

Migration can be controlled via application-specific policies. When a request for a reference to a variable reaches the process that owns that state, it can choose to *move* ownership to the caller. Applications can provide their own policies to determine when moving should be allowed by implementing a simple interface (see Listing 4.1). The PGo runtime provides two simple migration policies, allowing processes to always (or never) migrate the data to the requester.

```
1 type MigrationStrategy interface {
2         // ShouldMigrate is called when a process `requester` is attempting
3         // to get a reference to a varible of the given `name`.
4         ShouldMigrate(name, requester string) bool
5 }
```

Listing 4.1: Applications can provide a custom migration policy in the form of a definition of a *ShouldMigrate* function, defined when the user creates the global state resource.

4.2.2 Protocol

This section describes the underlying protocol that implements the semantics of the data store and object ownership described in 4.2.1. One of the key ideas behind the protocol is that references are obtained in lexicographical order of their

[1] https://golang.org/pkg/encoding/gob/

names. This notion is similar to acquiring archetype resources in consistent order as described in section 3.2 and avoids deadlocks under concurrent execution.

When a reference to a variable is obtained by a process, it causes subsequent requests for the same variable (from the same or other processes) to block until the reference is *released*. In other words, only one process in the system can have access to shared state at a time.

We discuss the protocol by overviewing three examples of its execution in increasing levels of complexity. In the diagrams that follow, arrows represent network calls performed from one process to another; straight vertical lines represent process execution over time.

Scenario 1: Borrowing a Single Variable (Figure 4.3) In this scenario, node P_1 wants to grab a reference to (borrow) variable `a`, which is believed to be owned by node P_2. P_2 turns out to actually have the ownership of `a`, so it returns a reference to it back to P_1. The green line in P_1 indicates the time during which it has exclusive access to `a` and runs application specific logic. In the meantime, a third node, P_3, wishes to get access to `a` too. However, since that variable is currently borrowed to P_1 (the entry in the data store is locked), the call is blocked (red line on P_3). When P_1 is done using variable `a`, it sends a `RELEASE` message to P_2. At that point, the entry for variable `a` is unlocked, and P_2 can now borrow variable `a` to P_3, which had been waiting for it.

Scenario 2: Multiple Variables and Outdated Ownership (Figure 4.4) This scenario illustrates how a node may combine requests for references to different variables in the same message. P_1 wants to borrow variables `a` and `b`, which it believes are owned by node P_2. When P_2 receives the request, it notices that it indeed owns variable `a`, but not variable `b` – it was previously moved to node P_3. Therefore, P_2 returns a reference to variable `a` (allowing P_1 to use it) and indicates that `b` has moved to process P_3. When receiving the response, P_1 updates its data store to indicate that `b` is owned by P_3 and moves on to request access to `b` from it. P_3 grants it access, and P_1 is able to use `a` and `b` exclusively (green line in the diagram). Once P_1 has used `a` and `b`, it can release the references that it borrowed – first to P_2 (for variable `a`) and

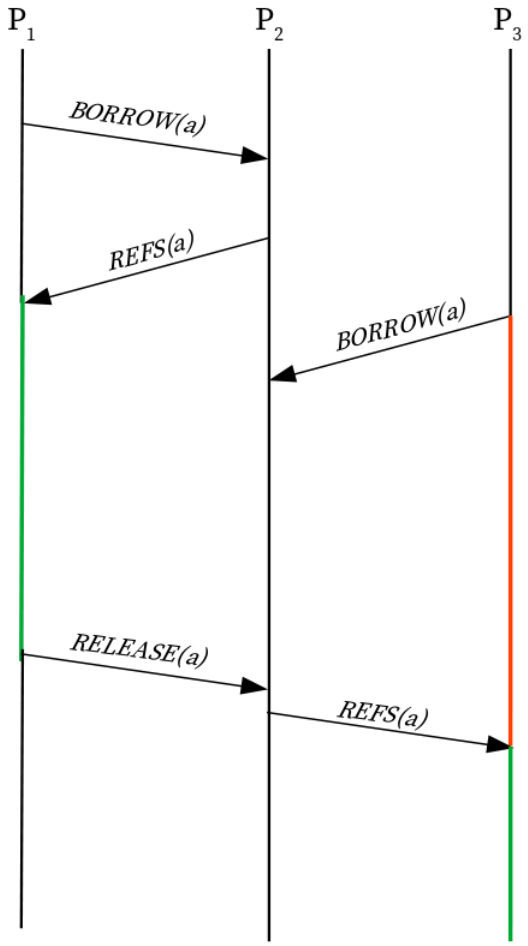

Figure 4.3: Borrowing a single variable from another process.

then to P_3 (for variable `b`).

Scenario 3: Ownership Move (Figure 4.5) This scenario is more complex and involves a number of complicated conditions. The order of events depicted in the diagram are:

- Process P_1 wants to have exclusive access to variables `a`, `b` and `c`, which it believes are owned by node P_2.
- Node P_2, when receiving the request, consults its data store and realizes

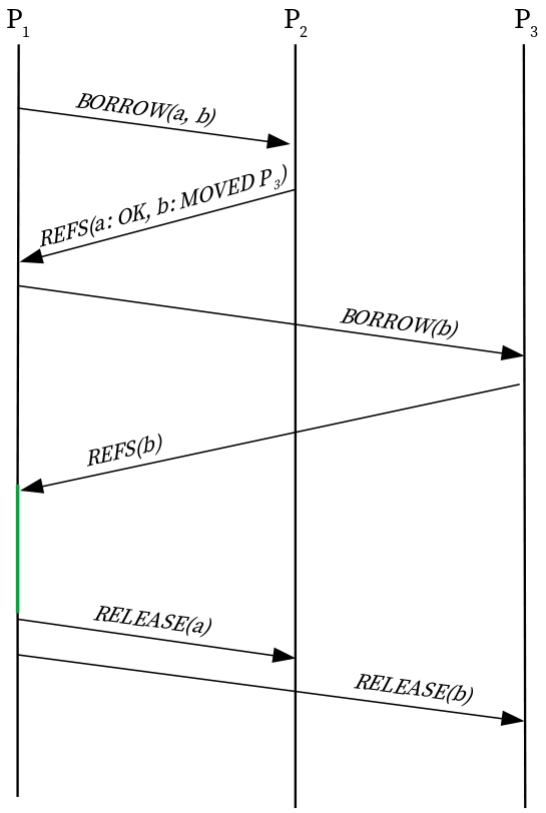

Figure 4.4: Requesting multiple references in the same message. In this example, P_2 no longer owns variable *b*, and it informs P_1 that the new owner is believed to be P_3.

that while it does own variables `a` and `c`, variable `b` was previously moved to P_3. In addition, it decides to move ownership of variable `a` to P_1. In the diagram, `a*: OK` indicates that P_2 returned a reference to `a`, including ownership of it.

- Since `b` has moved to P_3 (according to P_2), a reference to `c` cannot be returned to P_1 at this point. Exclusive access to variables need to be acquired in lexicographical order to avoid deadlocks. That's what `c: SKIP` in the diagram indicates.
- When receiving the reply from P_2, P_1 notices that it received a reference to `a` that includes ownership of the variable. It updates its data

store to indicate that it now owns variable `a` and acknowledges P_2 that it successfully received ownership of `a` and is ready to handle requests related to `a`.

- In the meantime, however, process P_3 wishes to have exclusive access to `a`, which it believes is owned by P_2 (and it indeed was until moments ago, before P_2 decided to move ownership of `a` to P_1). Since P_2 knows that an ownership move of `a` is currently underway, *it blocks until an acknowledgment from P_1 is received*.
- P_1 sends an `ACK(a)` message to P_2, indicating that ownership move is completed. Once that message is received, P_2 is ready to tell P_3, which had been waiting, that `a` moved to P_1.
- P_1 continues its `Acquire` process (this interaction started with P_1 needing exclusive access to `a`, `b` and `c`). Since it already has access to `a` (and owns it), it moves on to request `b` from P_3, and eventually receives a reference from it.
- In the meantime, P_3 itself wants access to variable `a`, which it now knows that it's owned by P_1. Since P_1 itself is currently using `a` exclusively for its own purposes, that request is blocked (red line on P_3 in the diagram).
- Finally, P_1 requests variable `c` to P_2, which then returns a reference to it (without ownership this time). Since P_1 has now exclusive access to `a`, `b` and `c`, it can now run its application logic (green line on P_1 in the diagram).
- Once the P_1's computation is over, it has to release the references it previously acquired. Variable `a` is now owned by it, so a local `RELEASE` call is performed. As soon as that is done, a reference to variable `a` can be returned to P_3, who had been waiting for some time. P_1 then moves on to release `b` (from P_3) and `c` (from P_2).

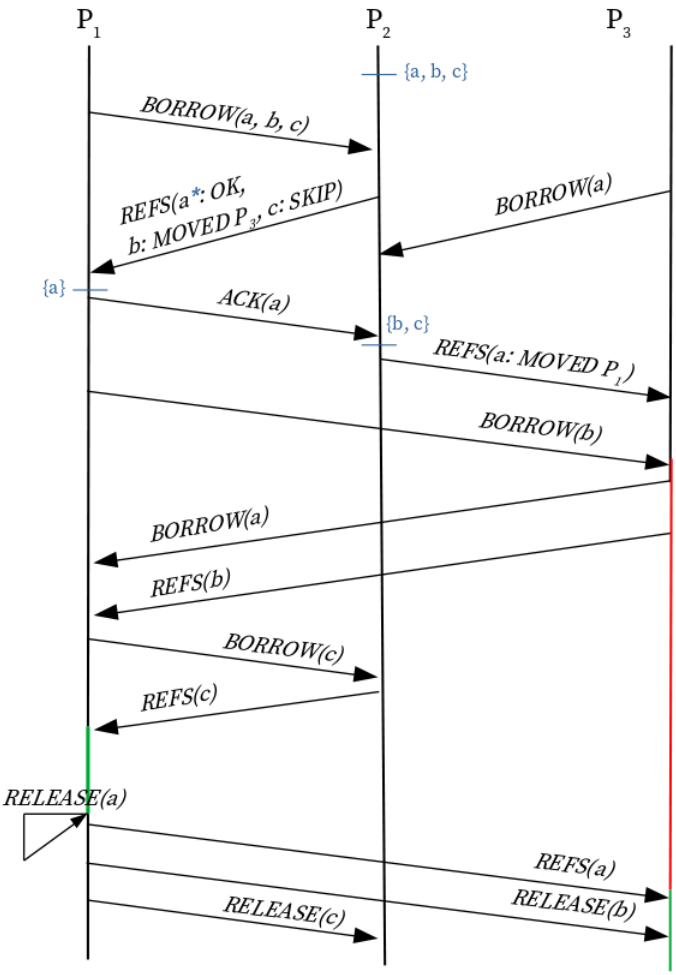

Figure 4.5: A complex interaction between processes P_1, P_2 and P_3 where ownership is moved and concurrent access to global variables needs to be resolved. Names in curly braces indicate which variables are owned by the processes at different times.

4.3 Communication Channels

The global state infrastructure provided by PGo exposes a shared-memory abstraction to the resulting distributed system: every process can read state that is shared, and updates are immediately visible after the corresponding action completes. Despite its simple semantics, distributed global state can come at a significant performance cost. A high contention environment, where multiple processes race to read and write shared state, can slow the system down, leading to degraded performance. In the context of distributed systems, however, it is common to think of the interactions between the components of a system in terms of the messages they exchange. To support this common pattern, PGo provides an archetype resource implementation that allows processes to *send messages* to one another.

Message-passing support in PGo aims to be an implementation of a communication channel with TCP-like semantics: messages are received in the order they were sent; no messages are lost, duplicated or corrupted. Each process communicates with other components in the system by posting messages to their *mailbox*; the idea is inspired by how actors communicate in an actor-based model [1, 19]. The communication abstraction targeted by this implementation, in the form of a Modular PlusCal mapping macro, is defined in Listing 4.2. Figure 4.6 illustrates a system of three processes using the message-passing implementation described in this section.

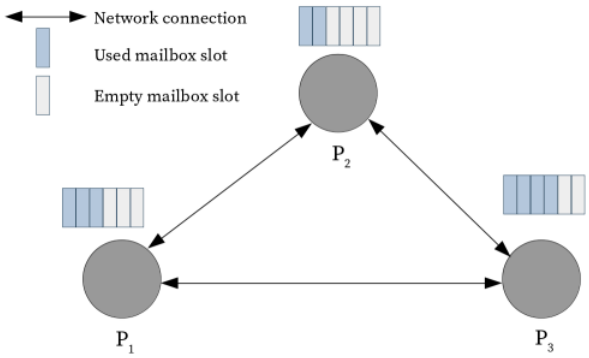

Figure 4.6: Three processes connected to each other. Each have a local mailbox of fixed capacity; when processes want to communicate, they send a message that is appended to the receiver's mailbox.

```
1  mapping macro ReliableMailboxes {
2      read {
3          await Len($variable) > 0;
4          with (msg = Head($variable)) {
5              $variable := Tail($variable);
6              yield msg;
7          };
8      }
9
10     write {
11         await Len($variable) < CAPACITY;
12         yield Append($variable, $value);
13     }
14 }
15
16 variable connections = [id \in NodeIds |-> << >>];
17
18 process (P \in NodeIds) == instance SomeArchetype(connections)
19     mapping connections[_] via ReliableMailboxes;
```

Listing 4.2: Definition of the message-passing abstraction supported by the PGo runtime.

The implementation of this mailbox-based resource is a simple wrapper on top of the underlying TCP stack. However, it is important to notice two important aspects of the `ReliableMailboxes` abstraction as defined previously:

- A PlusCal process that attempts to read a message from its mailbox is not enabled until there is at least one message to be read. To preserve these semantics, the implementation provided by PGo employs the following strategy: if an attempt to read a message is made when the mailbox of the corresponding process is empty, the `Read` call fails with an `AbortRetryError`, causing the action to be restarted. With enough time, the expected message should eventually reach the running process (as this was behavior tested in the specification).

- Similarly, a process that attempts to send a message to another process (post a message to the target's mailbox) will not be enabled if the destination mailbox is full (number of queued messages is equal to `CAPACITY`). In practice, the mailbox capacity is set to relatively low values for model checking purposes (to avoid exponential state-space growth) but can be set to arbitrarily

large values in a real implementation. PGo's strategy for this case is to have the receiver return an error to the caller when a message cannot be received due to a full mailbox The `Write` call on the connection, then, fails with `AbortRetryError`, causing the action to be restarted. Eventually, the receiver has processed enough messages to allow the sending of the message.

4.4 Other Common Resources

Apart from valid implementations of the archetype resource API (discussed in section 3.3) for the communication of two processes across the network, this work also provides a number of other implementations that are useful for building real systems. In this section, we briefly describe some of them.

File System Access. Reading and writing to persistent storage is a common concern that is abstracted away in specifications written in Modular PlusCal. One way to reconcile these abstractions and a concrete implementation is by passing a file system parameter to archetypes and mapping it as a function. The PGo runtime provides a `FileSystemResource` that implements the `ArchetypeResourceCollection` interface (see Listing 3.4). When given an arbitrary path in the system, this implementation is able to read and write files on behalf of the application and has the semantics expected by the resource API.

Locally Shared Data. Since archetypes can be seen as defining the API used to interact with the system, it is possible that multiple of the generated Go functions will be invoked in the same OS process. If they need to share data while executing concurrently (for instance, in different Go routines), they need to make sure their use of shared data is consistent with their atomic steps. PGo provides an implementation of local, shared data that processes can use in order to preserve Modular PlusCal execution semantics.

Time. Time is another entity that is represented abstractly in Modular PlusCal specifications. One alternative to model time is to pass an argument that is abstracted as a simple counter in the specification; however, a concrete im-

plementation can use PGo's `TimeResource` object that returns the current time when read.

Chapter 5

Implementation

In this chapter, we briefly describe implementation details of the techniques described in previous chapters. Section 5.1 outlines the changes in the PGo compiler performed in this work, and section 5.2 discusses updates in the execution runtime. Table 5.1 summarizes the implementation effort involved in this work.

Component	Language	Modules	LOC
Compiler	Java	Archetype Resource Usage Analysis	538
		Go Code Generation	2658
		Local State Snapshots	379
		Total	3575
Runtime	Go	Synchronized Start	275
		Distributed Global State	724
		Communication Channels	453
		Other Archetype Resources	267
		Total	1719

Table 5.1: Lines of code (LOC), excluding comments and blank lines, changed for the components of PGo updated in this work.

5.1 PGo Compiler

This work was implemented as an extension of the PGo compiler. Support for the verification of the Modular PlusCal language already existed, by translating it to PlusCal. The changes performed to support this work involved:

Static analysis of the specification. Once the Modular PlusCal model is parsed, we traverse the abstract syntax tree (AST) to determine which archetype resources are used in each atomic action. The compiler uses this information in order to generate code that calls the `Acquire` function with the appropriate `ResourceAccess` object (see Listing 3.4).

Code generation. A new compilation pass was added to PGo that transforms the Modular PlusCal AST into a Go AST. This involves: implementing the action execution algorithm described in Figure 3.1; acquiring (releasing) archetype parameters on the start (end) of every action; creating a snapshot of local state at the start of every label so that actions can be restarted; and transforming resource reads and writes into appropriate calls to the archetype resource API. In addition, we also keep track of every resource mapped as a function that is acquired in each action to avoid acquiring the same resource twice (since they are not statically distinguishable).

5.2 Distributed Runtime

Specifications of distributed systems compiled with PGo rely on a runtime to preserve the execution semantics of the specification. This work extended the PGo runtime by including the definitions of archetype resources presented in Chapter 3, as well as multiple implementations of the interface (described in Chapter 4). All inter-process communication, necessary to provide synchronized start (4.1), global state (4.2) and communication channels (4.3), is built on top of Go's RPC library[1].

[1] https://golang.org/pkg/net/rpc/

Chapter 6

Evaluation

In this chapter, we evaluate the techniques and mechanisms described in the previous chapters. In particular, we are interested in answering the following questions:

1. Is the implementation sufficiently robust to support the compilation of complex specifications?
2. Do systems produced by PGo have behavior that is defined by the specification?
3. What is the performance of systems compiled by PGo and how does it compare with similar, handwritten implementations?

Section 6.1 addresses the first question by describing the specifications compiled in this work and their complexity. Section 6.2 discusses whether the resulting system has behaviors that are valid according to the specification. Finally, section 6.3 describes performance measurements and discusses results.

6.1 Specification Complexity

Our evaluation is based on two models of distributed systems written in Modular PlusCal. Table 6.1 summarizes their complexity. The behavior defined by each model is described below:

Load Balancer. This model defines the interaction between three components of a system: a set of clients, a set of servers, and a load balancer. Clients send requests for files to the load balancer, which in turn redirects them to one of the servers. The load balancer chooses the server to redirect requests to in a round-robin fashion. While simple, this specification illustrates the use of communication channels across all components defined in the model; and, the use of a file system abstraction coupled with the corresponding resource provided by the PGo runtime allowing clients to request real files under the directory that the servers are running on.

Replicated Key-Value Store. This model defines the behavior of a key-value store with serializable key-value consistency semantics. It specifies a replicated state machine (RSM) that uses Lamport logical clocks [25] to determine ordering and stability, as described in [37]. In summary, this setting allows all replicas to be consistent without ever communicating with one another. Clients broadcast write (`Put`) operations to all replicas and send periodic clock-update messages. Clients are also allowed to *disconnect*: in this case, their clocks are no longer considered by the replicas when determining message stability.

Specification	Archetypes	Abstractions	#Lines
`load_balancer`	3	2	79
`replicated_kv`	5	6	291

Table 6.1: Complexity of the specifications used in the evaluation. **Abstractions** are counted based on the number of implementation-specific concerns that were not included in the model, each expressed as one or multiple mapping macros; **#Lines** includes lines of Modular PlusCal, excluding comments and blank lines.

PGo was able to compile both specifications, generating a function for each archetype defined in the models. A *main* function was manually written for each system that bootstraps the synchronized start protocol described in section 4.1 and provides concrete implementations for the abstractions used in the models. Both implementations use communication channels (section 4.3) to send messages be-

tween processes. Table 6.2 summarizes the output produced by PGo, and the effort required to get a running system from the generated code.

Specification	Generated LOC	Manual LOC
`load_balancer`	494	85
`replicated_kv`	3 395	234

Table 6.2: Generating a running implementation from the models evaluated. **Generated LOC** indicates lines of code produced by PGo from the archetype definitions; **Manual LOC** counts lines of code manually written to bootstrap the system. Both numbers exclude comments and blank lines.

6.2 Semantic Equivalence

While a proof that the systems we generated are semantically equivalent to the Modular PlusCal models is beyond the scope of this work, we were interested in having higher confidence that the resulting systems meet basic functional requirements. For this reason, we wrote a set of tests that exercised different aspects of each system.

For the load balancer, we performed a series of operations with different numbers of clients and servers, each requesting different files. Concurrency across requests was also tested. At the end of the process, we verified that: every client received a response for the requests it made; the contents of the responses are as expected (matching the underlying file system); and that each server handled the correct number of requests. In all cases, the resulting implementation presented expected behavior.

The replicated key-value store was tested similarly. We wrote a test generator that produces a sequence of operations to be performed by clients; keys and values are combinations of randomly generated bytes of configurable length. We performed tests with variable numbers of clients and replicas and also in highly concurrent environments where every operation is happening concurrently (in a separate Go routine). We verified that: every client terminated successfully; and that the database was consistent and identical on every replica at the end of the process. We found the system to behave as expected on every scenario and to be,

to the best of our knowledge, a correct implementation of the model.

6.3 Performance Comparison

We ran performance evaluations of both the load balancing system as well as the replicated key-value store against handwritten implementations. The manual implementation of the load balancer was written by the author with the goal of evaluating this work. The manual implementation of the key-value store was a slightly modified version of the best performing student implementation of the same system, which was a required assignment of the graduate course on Distributed Systems at UBC, offered in the winter term of 2019. We ran the same test suite described in section 6.2 and found out that every submitted implementation had at least one concurrency bug. This highlights the challenges of writing distributed systems that work as expected even for seemingly simple systems and demonstrates how a tool like PGo can help avoid these mistakes. Table 6.3 shows the effort required in the manual implementation of both models.

Specification	LOC
`load_balancer`	156
`replicated_kv`	406

Table 6.3: Effort required to implement both models evaluated in this work, in terms of the number of lines of code involved. Numbers exclude comments and blank lines.

6.3.1 Experimental Setup

All tests described in this section were performed by deploying all system processes in the same host; this is to reduce the impact of network latency and jitter in the numbers we aggregate and focus evaluation on overheads added by the PGo runtime. The host machine features an eight-core i7 1.80GHz processor with 16GB of memory and runs Linux 4.20.7. We used Go version 1.10.3. All numbers reported in the following graphs are averages of ten executions.

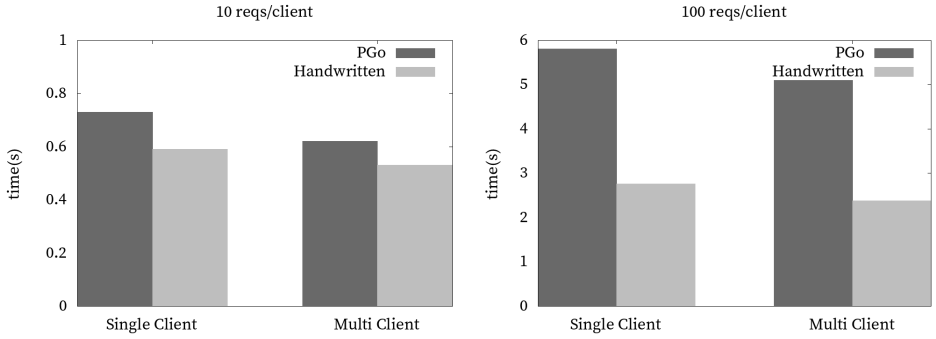

Figure 6.1: Execution time for the load balancer system, with one or multiple clients performing 10 (left) or 100 (right) requests per client.

6.3.2 Results

The load balancer system was run in two different scenarios: single client and multi-client. In both cases, clients performed multiple requests in sequence and waited for a response. The load balancer redirects requests to one of three servers. The files requested were randomly selected from a collection of previously downloaded web pages from popular websites. In the multi-client scenario, ten clients make requests concurrently to the load balancer. Figure 6.1 shows execution time for the two scenarios for different numbers of requests performed. The system compiled by PGo does not scale as nicely as the handwritten implementation due to the performance penalty incurred by the action restart mechanism described in Chapter 3.

The replicated key-value store was tested in an environment with two replicas and three clients. Each client performed one hundred randomly generated operations. Keys were set to be 32 bytes long and values 64 bytes long. Clock updates happened every 100ms. In addition, we tested both sequential and concurrent execution within a client: in concurrent mode, every operation runs in a separate Go routine. Since the Get and Put operations in the replicated key-value store require a response from the replica, we create a new mailbox whenever a client performs one of these operations. Figure 6.2 illustrates the results obtained in this experiment. As can be seen, the system compiled by PGo works correctly even under a highly concurrent environment. Although more extensive performance tests are

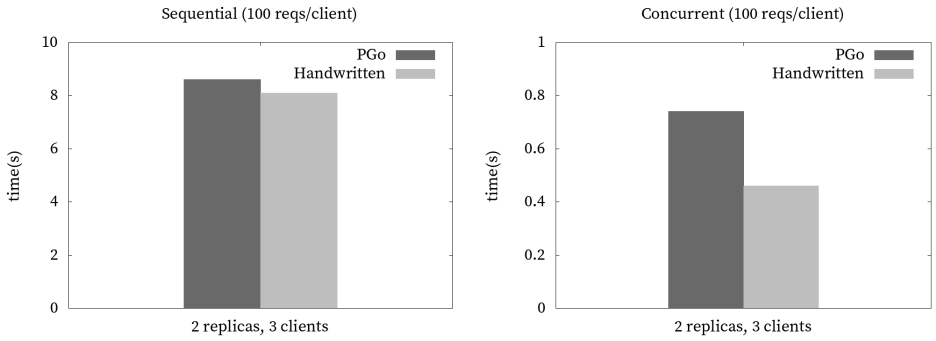

Figure 6.2: Time it takes for three clients to perform 100 operations, first sequentially (left), and then with Go routines (right).

needed to make conclusive claims, these preliminary results indicate that PGo has performance that is comparable to handwritten implementations in most cases but that may degrade if actions need to be restarted multiple times.

Chapter 7

Discussion

In this chapter, we address limitations of the proposed design and current implementation, and discuss future work.

Compilation is not verified. To trust that the specification meets the correctness properties and that the implementation refines the specification, the following need to be trusted: the TLC model checker, the PGo compiler, the Java compiler and runtime, the Go compiler and runtime, and the operating system. In addition, the archetype resources passed to the functions generated by PGo need to be correct implementations of the abstractions used in the specification. The only verified part in the workflow proposed in this work is that the Modular PlusCal model satisfies the correctness properties specified by the user. Providing stronger correctness guarantees of the compilation from Modular PlusCal to Go remains future work.

Fault tolerance. As discussed in section 3.6, there are limited ways to deal with failures while preserving the execution model followed in this work. However, they require significant effort from the developer, who needs to adapt both the specification and the implementation. PGo could make the process easier by providing fault-tolerance models built into the Modular PlusCal language: the compiler generates an abstraction of the failure model to PlusCal, allowing the designer to verify that the system is correct under that failure model. PGo can also provide implementations of fault-tolerant

components that developers can use, matching abstract failure models with resilient implementations the same way that environment abstractions are matched with concrete implementations in this work. This remains as future work.

Unsupported TLA$^+$ expressions. As described in Chapter 2, TLA$^+$ is a declarative specification language. Some expressions are very high level and difficult to translate to a concrete implementation (for example, the `CHOOSE` operator that selects an element from a set that satisfies a series of conditions). Further work needs to be done to support a larger portion of TLA$^+$.

Performance. While systems compiled with PGo have behavior that is defined by the specification they originated from, performance is an area that still needs further work. In particular, the process of aborting and retrying an atomic action is expensive, as local state needs to be reverted and recomputed. Instead of restarting an action from scratch, PGo could leave local state unmodified and just attempt to re-acquire all archetype resources used in the action. This, however, assumes that local computation is deterministic, which may not always be the case.

Fairness considerations. PlusCal processes (i.e., instances of Modular PlusCal archetypes) can be scheduled under unfair, weak, or strong fairness levels (for discussion and definitions, see [27]). Correctness properties written by the system designer may only hold under a specific fairness assumption. The PGo compiler and runtime take a best-effort approach to allow actions to be scheduled infinitely often if they remain enabled. However, many scheduling decisions are still made by the Go runtime, which is known to favor performance over fairness. Further work is required to enforce a more strict approach to matching TLA$^+$ fairness semantics with the way actions are scheduled in the implementation.

Chapter 8

Related Work

PGo is closely related to a vast body of previous work that attempted to bring the power of formal methods to the context of concurrent and distributed systems. In this chapter, we review some of the most closely related work previously published and how they compare with the approach proposed in this thesis.

Proof assistants. Verifying distributed systems by writing proofs is complex, leading researchers to look for ways to make the process more approachable. Verdi [38] enables developers to write proofs about the behavior of distributed protocols assuming a lossless network and provides automated transformers that produce a verified distributed system that works under different, more realistic network semantics. IronFleet [18] provides tools that allow developers to prove that realistic implementations refine a high-level specification based on the Dafny language [32]. The verification process supports both safety and liveness properties, although in a limited way.

The work presented in this thesis differs from this category by relying on model checking of specifications rather than formal proofs of correctness. While model checking by state-space exploration generally does not prove systems to be correct, it requires significantly less effort from the developer of the system. More specifically, PGo enables developers to reason about their system abstractly and have sufficient confidence (but no guarantee) in its correctness, while at the same time freeing developers from the arduous

task of writing proofs, a process that is known to require a significant amount of effort, expertise and time [23].

Domain-Specific Languages and Compilers. DSLs have been extensively used in the past to provide a higher-level, more expressive way to build distributed systems. Previous work has demonstrated that generating source code from such alternative representations can be beneficial and is similar to our approach. Mace [21, 22] is a C++ language extension and compiler that generates a distributed system implementation from a state-machine representation. The approach taken by Mace is similar to the one we used in our work; however, PGo better supports the evolution of such systems by providing a clear boundary between protocol specification and implementation details due to its modular approach. P [4] is a domain-specific language for asynchronous, event-driven programs. It was used successfully to build the USB stack on Microsoft Windows 8. P and Modular PlusCal share the goal of isolating implementation details that are not relevant for the model being checked. P# [8] is an evolution of that work: it supports modeling and systematic testing of distributed systems. P# has been used in production at Microsoft [9]; however, the developer needs to provide both an implementation and an abstract model of the components of the system that are not under test to avoid state-space explosion.

Model Checking Implementations. Previous work has applied the idea of state-space exploration directly to implementations of systems, as opposed to abstractions of their behavior. VeriSoft [15] is a model checker for arbitrary C programs that uses a stateless exploration algorithm combined with partial-order reduction techniques to reduce the state space. MODIST [17, 39] extends the idea and is able to model check the implementation of distributed systems. While both of these systems have the advantage of verifying actual implementations, the approach is subject to the state-space explosion problem. PGo aims to allow model checking to remain efficient while still providing many of the advantages that make checking implementations attractive by automating the transition from an abstract model to a concrete implementation.

Chapter 9

Conclusion

In this thesis, we have proposed a technique to automatically generate implementations of distributed systems from models of their behavior in the Modular PlusCal specification language. The process is built on the idea of matching components defined abstractly in a model with concrete implementations that present the same behavior. We also discussed how to preserve execution semantics of Modular PlusCal; in particular, how to provide behavior equivalent to the atomic steps of TLA$^+$ while allowing as much concurrency as possible, for performance reasons. Our evaluation has shown that the current implementation is able to handle complex specifications and generate implementations that satisfy the correctness properties previously model-checked.

We are still working to make PGo a viable option for the development of practical distributed systems. In particular, we hope to provide better support for the construction of fault-tolerant systems and to optimize the generated code and execution runtime to have performance closer to that of handwritten implementations.

Bibliography

[1] G. Agha. An Overview of Actor Languages. In *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming*, OOPWORK '86, pages 58–67, New York, NY, USA, 1986. ACM. ISBN 0-89791-205-5. doi:10.1145/323779.323743. URL http://doi.acm.org/10.1145/323779.323743. → page 42

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, Sep 1987. ISSN 1432-0452. doi:10.1007/BF01782772. URL https://doi.org/10.1007/BF01782772. → page 6

[3] Amazon. Amazon S3 Availability Event, 2008. URL https://status.aws.amazon.com/s3-20080720.html. → page 1

[4] V. and Gupta, E. Jackson, , and S. a. Rajamani. P: Safe Asynchronous Event-Driven Programming. Technical report, November 2012. URL https://www.microsoft.com/en-us/research/publication/p-safe-asynchronous-event-driven-programming/. → page 57

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-65703-7. URL http://dl.acm.org/citation.cfm?id=646483.691738. → page 7

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi:10.1145/1281100.1281103. URL

http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/1281100.1281103. → page 1

[7] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18 (6):1157–1210, Sept. 2010. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1891823.1891830. → page 6

[8] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson. Asynchronous Programming, Analysis and Testing with State Machines. *SIGPLAN Not.*, 50(6):154–164, June 2015. ISSN 0362-1340. doi:10.1145/2813885.2737996. URL http://doi.acm.org/10.1145/2813885.2737996. → page 57

[9] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, and W. Schulte. Uncovering Bugs in Distributed Storage Systems During Testing (Not in Production!). In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 249–262, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-28-7. URL http://dl.acm.org/citation.cfm?id=2930583.2930602. → page 57

[10] M. Demirbas. TLA+ Specifications of the Consistency Guarantees Provided by Cosmos DB. Microsoft Research talk, Nov. 2018. → page 8

[11] C. Flanagan and P. Godefroid. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi:10.1145/1040305.1040315. URL http://doi.acm.org/10.1145/1040305.1040315. → page 7

[12] GitHub. October 21 post-incident analysis, 2018. URL https://github.blog/2018-10-30-oct21-post-incident-analysis. → page 1

[13] GitLab. Postmortem of database outage of January 31, 2017. URL https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/. → page 1

[14] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg, 1996. ISBN 3540607617. → page 7

[15] P. Godefroid. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi:10.1145/263699.263717. URL http://doi.acm.org/10.1145/263699.263717. → page 57

[16] Google. Google Compute Engine Incident 17003, 2018. URL https://status.cloud.google.com/incident/compute/17003. → page 1

[17] H. Guo, , L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. Symposium on Operating Systems Principles (SOSP), October 2011. URL https://www.microsoft.com/en-us/research/publication/practical-software-model-checking-via-dynamic-interface-reduction/. → page 57

[18] C. Hawblitzel, , , J. Lorch, , M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015. URL https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed-systems-correct/. → pages 18, 56

[19] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1624775.1624804. → page 42

[20] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.*, 6(1):109–133, Feb. 1988. ISSN 0734-2071. doi:10.1145/35037.42182. URL http://doi.acm.org/10.1145/35037.42182. → page 34

[21] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1973430.1973448. → pages 2, 57

[22] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. *SIGPLAN Not.*, 42(6): 179–188, June 2007. ISSN 0362-1340. doi:10.1145/1273442.1250755. URL http://doi.acm.org/10.1145/1273442.1250755. → page 57

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi:10.1145/1629575.1629596. URL http://doi.acm.org/10.1145/1629575.1629596. → page 57

[24] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, Mar. 1977. ISSN 0098-5589. doi:10.1109/TSE.1977.229904. URL https://doi.org/10.1109/TSE.1977.229904. → page 6

[25] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi:10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563. → page 49

[26] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994. ISSN 0164-0925. doi:10.1145/177492.177726. URL http://doi.acm.org/10.1145/177492.177726. → page 7

[27] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 032114306X. → pages 2, 7, 15, 31, 55

[28] L. Lamport. The PlusCal Algorithm Language. *Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.*, January 2009. URL https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/. → pages 2, 15

[29] L. Lamport. The PlusCal Algorithm Language. *Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.*, January 2009.

URL https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/. → page 9

[30] L. Lamport. A PlusCal's User Manual. Online material, Aug. 2018. → page 15

[31] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. ISSN 0164-0925. doi:10.1145/357172.357176. URL http://doi.acm.org/10.1145/357172.357176. → page 27

[32] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4. → page 56

[33] R. J. Lipton. Reduction: A New Method of Proving Properties of Systems of Processes. In *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '75, pages 78–86, New York, NY, USA, 1975. ACM. doi:10.1145/512976.512985. URL http://doi.acm.org/10.1145/512976.512985. → page 18

[34] B. Liskov. Distributed Programming in Argus. *Commun. ACM*, 31(3): 300–312, Mar. 1988. ISSN 0001-0782. doi:10.1145/42392.42399. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/42392.42399. → pages 30, 34

[35] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, Mar. 2015. ISSN 0001-0782. doi:10.1145/2699417. URL http://doi.acm.org/10.1145/2699417. → page 8

[36] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi:10.1109/SFCS.1977.32. URL https://doi.org/10.1109/SFCS.1977.32. → page 7

[37] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990. ISSN 0360-0300. doi:10.1145/98163.98167. URL http://doi.acm.org/10.1145/98163.98167. → page 49

[38] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. *SIGPLAN Not.*, 50(6):357–368, June 2015. ISSN 0362-1340. doi:10.1145/2813885.2737958. URL http://doi.acm.org/10.1145/2813885.2737958. → page 56

[39] J. Yang, , , Z. Xu, , H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228. USENIX, April 2009. URL https://www.microsoft.com/en-us/research/publication/modist-transparent-model-checking-of-unmodified-distributed-systems/. → page 57

[40] B. Zhang. PGo: Corresponding a High-Level Formal Specification with Its Implementation. *SOSP SRC*, 2016. → pages 3, 12