

# Compiling Distributed System Specifications into Implementations

Matthew Do\*

Renato Mascarenhas\*

Finn Hackett<sup>†‡</sup>

Brandon Zhang\*

Yi Fan (Bob) Yang<sup>§‡</sup>

Adam Geller\*

Ivan Beschastnikh\*

## Abstract

Distributed systems are difficult to design and implement correctly. There is a growing interest in specification (spec) languages for distributed systems, which can be checked exhaustively or proved to satisfy certain properties. For example, Amazon uses TLA+ and PlusCal in building its web services [9]. However, there is no way to generate an implementation of spec in one of these modeling languages. Towards this end, we are building PGo. The goal of PGo is to reduce developer burden by providing a mechanical translation between an abstract spec and a concrete implementation.

## 1 Introduction and Related Work

Distributed systems involve many nodes running asynchronously, and must tolerate faults such as network failure and machine crashes. These properties make distributed systems difficult to reason about. Bugs in these systems can be subtle and have catastrophic consequences. As just one example, Amazon’s Elastic Compute Cloud (EC2) had a rare race condition which caused a major outage [1].

*Mace* and *P* are domain-specific languages for specifying distributed systems and asynchronous state machines, respectively [2, 7]. *Mace* and *P* can both be model checked and compiled into executables. However, model checking these languages may be impractical, since the spec writer must implement low-level details, contributing to state-space explosion.

*Verdi* is a framework which compiles a spec of a distributed system with a fault-tolerant implementation [10]. A *Verdi* spec is written in OCaml assuming an ideal network and verified with Coq, which requires extra developer effort. *Verdi* then transforms the spec into an equivalent implementation which is tolerant of network faults and node failures. Similarly, *IronFleet* is a proof system for Dafny [6]. *IronFleet* similarly requires significant developer effort to generate a proof of correctness.

*MODIST* and its successor *DEMETER*, are model checkers for unmodified distributed systems [5, 11]. *MODIST* bypasses the need for a formal spec language, as the implementation itself can be checked. However, using *MODIST* is

impractical for large systems, since the state space explodes due to irrelevant implementation details.

*TLA+* is a formal spec language for concurrent systems. *TLA+* is based on set theory, discrete mathematics, and the temporal logic of actions (TLA). A *TLA+* spec can be checked using the TLC model checker, and the *TLA* proof system (TLAPS) facilitates the writing of machine-checked proofs. *PlusCal* is a language that makes it easier to write *TLA+* specs. *PlusCal* has a C-style syntax and can be translated into *TLA+*. Model checking is tractable in *TLA+* and *PlusCal* because systems can be specified at an abstract level: the spec writer can write interfaces for the components of the system and check their implementations individually [8].

*PlusCal* has many useful abstractions for reasoning about concurrent programs. One notable abstraction is labeled critical sections, which act as atomic operation regions.

*TLA+* and *PlusCal* have been used by industry to verify models of deployed distributed systems. Amazon used *TLA+* and *PlusCal* to verify the systems running their web services, as a design tool, and as a form of documentation [9]. Geambasu et al. used *TLA+* to specify different distributed file systems [4], with the conclusion that formal spec enables the developer to reason about complex systems at a higher level of abstraction.

However, there is no way to relate a *PlusCal* spec to an implementation. The mathematical language used in modeling does not resemble implemented code. *PlusCal* requires developers to manually translate a spec into the implementation, which may introduce bugs due to translation and interpretation.

We introduce *PGo*, a compiler for *PlusCal* that reduces the effort needed to convert *PlusCal* into executable code. *PGo* can also compile specs from *Modular PlusCal* (MPCal), an extension of *PlusCal* that assists the developer with making *PlusCal* specs modular. *PGo* compiles a *PlusCal* or *MPCal* spec into Go while preserving its semantics, so that a verified *PlusCal* algorithm compiles to a correct Go implementation. *PGo*’s automation gives the developer the best of both worlds: tractable model-checking via abstract specs and a low-level implementation connected via mechanical translation.

Compilation of *PlusCal* presents several unique challenges. Because *PGo* must compile from a spec to an implementation, it must deal with differing semantics and abstractions while maintaining correctness. These challenges, and our solutions, are overviewed below.

\*Computer Science, University of British Columbia

<sup>†</sup>Work done while at the University of British Columbia

<sup>‡</sup>Computer Science, University of Waterloo

<sup>§</sup>Facebook

## 2 Design

### Preserving Concurrency Semantics.

PlusCal specs include precise information about the concurrency semantics of a program, including variable accesses, atomic operation sets, procedure synchronization, and scheduling. For non-distributed, multithreaded applications, PGo performs rudimentary static analysis to determine the groups of variables that must be guarded by a mutex. An atomic section is handled by generating a mutex that guards the variables accessed in the section. PGo handles synchronization by using Go's `sync.WaitGroup` and `channels` to ensure that procedures run to completion before exiting. PGo does not guarantee fair scheduling of Go procedures because controlling scheduling in Go is too expensive.

**Distributed State Management.** The correctness of PGo output hinges on its ability to reliably and correctly implement the global state in Go. The global state is shared global variables from the PlusCal spec. While correctness is PGo's top priority, performance is also a concern, especially for distributed algorithms. In a distributed context, poor object placement can lead to terrible performance.

Finding a good object placement requires static analysis on the part of PGo. One advantage of working with PlusCal specs is that we can use the PlusCal model checker to assist in the static analysis. As one example, we can test whether two procedures can read/write the same variable concurrently, in which case the writes would need to be serialized. In this case, a central placement would be preferable so less nodes are involved in reads/writes, especially if many procedures are reading/writing concurrently. Otherwise, if a variable is only written to by one procedure at a time, then a de-centralized model using a distributed lock would be the best option because the lock-fetching algorithm, while expensive, would be run infrequently, and variable accesses would be inexpensive.

Our plan is to support various object placements and access patterns configurations by writing Go libraries to implement them, which we would use in PGo's generated Go code.

**Making PGo Output Modular.** PGo output cannot be used along with other Go code without modification. This is because PlusCal algorithms cannot reference or interact with each other. As such, there is no way to link PGo-compiled implementations together, except by shared global variables. However, PlusCal assumes exclusive access to global variables, so using them to link algorithms together may break invariants.

A closely related issue is that PlusCal abstracts away devices, networks, and other details to make model checking tractable. However, this requires that these are added by the developer post-compilation in Go.

Both problems require the developer to modify PGo output, so the developer loses the provided guarantees. To get around these issues, we designed MPCal an extension of PlusCal. MPCal has four key features:

*Archetypes* are parameterized PlusCal processes that can communicate through their parameters. They are used to shift PlusCal's abstractions down to the granularity of their parameters. Archetypes are compiled to Go functions which analogously interact with the world via their parameters. Compiled archetypes accept any Go object matching a simple read/write interface, so they can easily be plugged in by the developer.

*Ref Parameters* allow Archetypes (and procedures called from within Archetypes) to write to parameters. Normal parameters are read-only.

*Mapping Macros* and *Interleavings* assist the developer in modeling the behavior of real systems. Mapping macros provide information to the model checker about the behavior of reads/writes for an archetype parameter. This acts as a spec for what will be passed to the compiled archetype. As one example, mapping macros can be used to model lossy, reordering networks. Interleavings specify a code block to execute between critical sections; they model partial or total failures.

## 3 Evaluation and Future Directions

We used PGo to compile several real-world PlusCal specifications: the  $n$ -queens algorithm, Dijkstra's mutual exclusion algorithm [3], and a distributed queue algorithm. To test PGo's compilation of MPCal, we are also working on a 2-phase commit spec.

We intend to come up with a strategy to verify PGo to guarantee the correctness of its output, and to find a way for compiled Go code to be quickly rechecked after small modifications.

## 4 Conclusion

We have introduced PGo, a compiler from PlusCal and MPCal formal specs into Go implementations. PGo is designed to reduce the developer burden of implementing a correct spec, and thus increases the developer's confidence in the correctness of the implementation. We are actively developing PGo<sup>1</sup>.

## References

- [1] Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region. <http://aws.amazon.com/message/65648/>, 2011.
- [2] DESAI, A., GUPTA, V., JACKSON, E., QADEER, S., RAJAMANI, S., AND ZUFFEREY, D. P. Safe Asynchronous Event-driven Programming. *SIGPLAN Not.* 48, 6 (June 2013), 321–332.
- [3] DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *Communications of the ACM* 8 (1965), 569.
- [4] GEAMBASU, R., BIRRELL, A., AND MACCORMICK, J. Experiences with formal specification of fault-tolerant file systems. In *International Conference on Dependable Systems and Networks (DSN)* (June 2008), pp. 96–101.
- [5] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical software model checking via dynamic interface reduction.

<sup>1</sup><https://github.com/UBC-NSS/pgo>

- In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 265–278.
- [6] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 1–17.
  - [7] KILLIAN, C., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. Mace: Language Support for Building Distributed Systems. *PLDI* 42 (2007), 179–188.
  - [8] LAMPORT, L. *Specifying Systems*. Addison-Wesley, 2002, ch. 10, pp. 135–168.
  - [9] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (Mar. 2015), 66–73.
  - [10] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. *PLDI* 50 (2015), 357–368.
  - [11] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI* (2009).