PGo: Corresponding a high-level formal specification with its implementation

Brandon Zhang

Computer Science, University of British Columbia

Abstract

Distributed systems are difficult to design and implement correctly. There is a growing interest in specification 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 [14]. PlusCal is a formal specification language which has simple constructs for synchronization, nondeterminism, and specifying safety and liveness, which makes it an ideal choice to specify distributed algorithms. However, currently there is no tool to correspond a PlusCal specification with the implementation. Towards this end, we are building PGo, which can currently compile a considerable subset of PlusCal algorithms into Go programs.

1. Introduction

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. For example, Amazon's Elastic Compute Cloud (EC2) had a rare race condition which caused a major outage [1].

TLA+ is a formal specification language for concurrent systems. TLA+ is based on set theory, discrete mathematics, and the temporal logic of actions (TLA). A TLA+ specification 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+ specifications. 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 specification writer can write interfaces for the components of the system and check their implementations individually [9].

TLA+ and PlusCal have been used by industry to verify deployed distributed systems. Amazon uses TLA+ and PlusCal to verify the systems running their web services, as a design tool, and as a form of documentation [14]. Geambasu et al. used TLA+ to specify different distributed file systems [5], with the conclusion that formal specification enables the developer to reason about complex systems at a higher level of abstraction. Many research systems, such as SMART [12] and a Byzantine-fault-tolerant version of Paxos [2], have accompanying TLA+ specifications and proofs of correctness [7, 10].

While PlusCal can be verified using TLC and TLAPS, there is no way to correspond a PlusCal specification and its implementation. This limits confidence in the correctness of the implementation. For example, a developer who is implementing the specification may introduce bugs.

We introduce *PGo*, a compiler for PlusCal that reduces the amount of manual translation needed to convert PlusCal into executable code. PGo compiles a PlusCal specification into Go while preserving its semantics, so that a verified PlusCal algorithm compiles to a correct Go implementation.

2. Related work

Other frameworks have been developed with a similar goal of corresponding specification and implementation. A common point between these is that their specification languages lack PlusCal's useful abstractions. PGo aims to offer both the simplicity and fast model checking of PlusCal and the correctness of the implementation.

Mace and P are domain-specific languages for specifying distributed systems and asynchronous state machines, respectively [3, 8]. Mace and P can both be model checked and compiled into executables. However, these languages lack proof systems and model checking may be impractical, since the specification writer must implement low-level details, contributing to state-space explosion. PlusCal's language features make it simpler to express core concurrency features of a system.

SOSP'17 October 29–31, 2017, Shanghai, China

ISBN [to be supplied]. . . $15.00

Verdi is a framework which corresponds the specification of a distributed system with a fault-tolerant implementation [15]. A Verdi specification is written in OCaml assuming an ideal network and verified with the Coq proof system. Verdi then transforms the specification into an equivalent implementation which is tolerant of network faults and node failures. Verdi does not have a model checker, so Verdi systems must be verified by writing a Coq proof, which requires more developer effort than specifying correctness properties and using a model checker to verify them. Similarly, IronFleet is a proof system for Dafny [6]. IronFleet similarly requires significant developer effort to generate a proof of correctness.

MODIST is a model checker for unmodified distributed systems [16]. MODIST bypasses the need for a formal specification language, as the implementation itself can be checked. However, using MODIST is impractical for large systems, since the state space expands due to low-level implementation details.

3. Design

Compilation of PlusCal presents several challenges:

- Inferring types of an untyped language, and guaranteeing type safety of the compiled program.

  PlusCal is an untyped language, but it has well-defined primitive types: numbers, sets, maps, and tuples. PGo infers types of variables based on the TLA+ expressions used in assignments and variable declarations. PGo's intermediate type system allows it to detect type conflicts.

- Compiling complex expressions, such as set notations, without sacrificing readability of the compiled code.

  PGo includes the `pgoutil` Go library which implements complex TLA+ operators and data structures. This allows the compiled Go program to have readability similar to the PlusCal specification.

- Translating abstractions (e.g., constants defined to be arbitrary) into concrete implementations.

  PGo requires arbitrary constants to be annotated with a concrete value or compiles these into program inputs.

- Preserving concurrency semantics, especially synchronization and atomicity.

  Multiprocess PlusCal algorithms are compiled to Go programs that spawn a goroutine for each PlusCal process. Currently, PGo performs rudimentary static analysis to determine the groups of variables that must be guarded by a mutex. An atomic step (indicated by labels in PlusCal) is performed by locking the mutex which guards the variables accessed in that step.

4. Evaluation

We tested PGo on PlusCal specifications of the n-queens algorithm and on Dijkstra's mutual exclusion algorithm [4]. The PlusCal version of the n-queens algorithm is a single-process algorithm which includes several complex TLA+ set expressions and nontrivial PlusCal language constructs. The PlusCal version of Dijkstra's mutex is a simple multiprocess algorithm but is challenging to compile due to its use of maps and different labeled grains of atomicity. In both cases, PGo compiled executable Go code which produced the expected output when run without modification.

5. Future directions

We are actively developing PGo[1]. Here we overview some of our ongoing work.

Evolving specifications/implementations. Our goal is to expand PGo into a suite of tools that allow developers to manage the correspondence between a formal spec and an implementation, even as both of these change over time. Currently, any editing of the Go code compiled by PGo will weaken the correspondence with the specification. One possible solution for this is to use design by contract [13] for different components of the specification, and to insert runtime checks for the contracts. The developer could also annotate components of the PlusCal spec that will be modified after compilation and manually associate contracts with the component interface. A contract can be expressed in PlusCal with assertion statements, or as a TLA+ interface refinement.

Other language features. PGo is missing support for certain PlusCal language features, such as `await`. We plan to add support for `await` by using Go's `sync.Cond`.

IPC and networking support. We plan to support annotations that allow the developer to express how to compile inter-process communication in PlusCal. For example, PGo may autogenerate the necessary RPC logic in Go.

Beyond Go. PGo's intermediate structure can be compiled to other targets, such as C++ and Java.

Verifying PGo. PGo has not been verified. To be certain that it produces correct code, we would like to use existing compiler verification best practices [11].

6. Conclusion

We have introduced PGo, a compiler from PlusCal formal specifications into Go language. PGo can be used to reduce the developer burden of implementing a specification correctly, and thus increase their confidence in the correctness of the implementation.

Acknowledgments

Bob Yang developed the initial PGo prototype. This research was performed under the supervision of Ivan Beschastnikh, and was sponsored by NSERC and a Science Undergraduate Research Experience (SURE) award from UBC.

[1] `https://bitbucket.org/whiteoutblackout/pgo`

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] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[3] 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.

[4] DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *Communications of the ACM 8* (1965), 569.

[5] 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.

[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] HOWELL, J., LORCH, J., AND DOUCEUR, J. J. Correctness of Paxos with Replica-Set-Specific Views. Tech. rep., June 2004.

[8] KILLIAN, C., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. Mace: Language Support for Building Distributed Systems. *PLDI 42* (2007), 179–188.

[9] LAMPORT, L. *Specifying Systems*. Addison-Wesley, 2002, ch. 10, pp. 135–168.

[10] LAMPORT, L. Byzantizing Paxos by Refinement. In *Proceedings of the 25th International Conference on Distributed Computing* (Berlin, Heidelberg, 2011), DISC'11, Springer-Verlag, pp. 211–224.

[11] LEROY, X. Formal Verification of a Realistic Compiler. *Commun. ACM 52*, 7 (July 2009), 107–115.

[12] LORCH, J. R., ADYA, A., BOLOSKY, W. J., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., DOUCEUR, J. J., LORCH, J., AND BOLOSKY, B. The SMART way to migrate replicated stateful services. In *Proceedings of the 2006 EuroSys Conference* (Leuven, Belgium, April 2006), Association for Computing Machinery, Inc., p. 103115.

[13] MEYER, B. Applying "design by contract". *Computer 25*, 10 (Oct. 1992), 40–51.

[14] 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.

[15] 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.

[16] 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).