Challenges and Opportunities in Model Checking Large Scale Distributed Systems

Rupak Majumdar
Max Planck Institute for Software Systems (MPI-SWS)
Germany
Amazon Web Services
USA
rupak@mpi-sws.org

ABSTRACT

The goal of the Must project is to provide design and verification support for industrial-scale distributed systems. We provide an overview of the project: its design goals, its technical features, as well as some lessons we learnt in the process of transferring academic research to an industrial tool.

ACM Reference Format:
Rupak Majumdar. 2024. Challenges and Opportunities in Model Checking Large Scale Distributed Systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3597503.3649398

1 OVERVIEW

The goal of the Must project is to enable programmers to design and implement high-confidence distributed applications. We started the Must project with two desired goals. First, the ultimate goal of Must is to provide full *unbounded* verification at the implementation level. Second, we require that all artefacts related to formal reasoning be owned and maintained by the development teams in the course of their usual workflows, not by additional teams of formal methods experts or consultants working in parallel.

The first goal is a point of departure from many projects on large-scale testing. While we are far from achieving this goal at the scale of industrial distributed systems, keeping an eye towards full verification helped in our design decisions along the way. I will argue that the goal of soundness pushed us to design better algorithms in the short run that, while not providing full and unbounded formal verification, already outperform more naive approaches in terms of coverage.

The second point led us to modeling systems directly at the level of programming languages already used by the developer teams, as opposed to modeling in domain specific and more abstract modeling languages. An additional language for formal work adds friction in the development process; our prior experience has been that formal artefacts not in the developer pipelines can quickly become stale. In addition, it should not come as a surprise to the software engineering community that maintaining a language and a development environment is hard. Domain specific languages grow over time, and ensuring a smooth developer experience (including providing all the software engineering scaffolding we take for granted) is difficult and expensive.

Instead, Must embeds an API for concurrent, message-passing, distributed systems directly within the Rust programming language. The API provides an abstract and parameterized interface for message-passing as well as core constructs for process creation and nondeterminism. Sequential computation is written directly in Rust; the type and module system of the Rust programming language provides mechanisms for compositionality and information-hiding "for free." Additional Rust APIs are simulated by internally translating their semantics to the core Must API.

Our initial focus is on systematic exploration of concurrency, rather than data nondeterminism. Thus, we focused on model checking as a first deliverable, because of its relatively easy integration into developer workflows and as a gateway to more expressive but more expensive formal methods involving user annotations and proofs. We have found that systematic exploration of concurrency, together with judicious symbolic analysis, can already be a powerful tool for distributed systems development.

2 THE MUST MODEL CHECKER

The technical core of Must is an API for process creation and for message-passing communication. Process creation is fairly straightforward and mimics thread creation APIs in Rust. The message passing primitives $\mathsf{send}(\mathsf{tid}, v)_{\mathsf{C}}$ and $\mathsf{recv}(\lambda x : e)_{\mathsf{C}}$ provide message send and (predicated) message receipts, respectively. The primitives are parameterized by a *communication model* C—different communication models are used to define different message passing semantics such as peer-to-peer, causal delivery, or full asynchrony that are used (often in combination) in distributed applications. The predicate $\lambda x : e$ expresses a predicate that must hold for the receive to succeed; predicated receives are a fairly powerful modeling and implementation primitive for many protocols. Additional communication APIs in Rust are simulated by translating their semantics to the core API. For example, we can transparently support standard Rust APIs for network communication and RPC communication.

At the theoretical level, we provide a novel *partial order* semantics to Must programs parameterized by the communication model. The semantics defines a set of *execution graphs* for a given program. In practice, the partial order view leads to a large reduction in the state space, and the model checker can systematically explore many protocols to completion.

ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3649398

The Must model checker implements a novel *optimal* dynamic partial-order reduction (ODPOR) algorithm to explore execution graphs. Optimality means that the model checker explores exactly one execution for each possible execution graph allowed by the semantics and does not perform futile explorations. The model checker only has a polynomial memory overhead on the execution—in practice, this means that the model checker has very predictable memory usage and can be run for several days on many cores without memory issues.

While the Must project is in a relatively early stage, we are encouraged by our progress and early signs of adoption. We have used Must as a model checker on top of existing distributed systems protocols, such as replicated logs, leader election and Paxos, and database transaction systems. Our initial experiments are encouraging—both in terms of developer engagement and in terms of scaling model checking. Along the way, we implemented various usability requirements—including integration into build systems, replayability and debugging, automated collection and analysis of metrics, and others—that are crucial to deploying formal tools in a development pipeline. Finally, the adoption and use of formal techniques is also a social process—we saw the importance of formal methods evangelism in achieving broad success of verification techniques in software development.

ACKNOWLEDGMENTS

Must is a collaborative effort with many researchers and engineers. I would like to thank my close collaborators Constantin Enea, Dimitra Giannakopoulou, and Michalis Kokologiannakis on the project, as well as Byron Cook, Ankush Das, Ankush Desai, Murat Demirbas, Cezara Dragoi, Ferhat Erata, Anjali Joshi, Rajeev Joshi, Mohammad Khoshechin, Paul Tillotson, Mark Tuttle, and many others for their advice and support.