Research Statement

Klaus v. Gleissenthall

Research Interests

Computer systems are often incredibly complex; to get them right, programmers have to answer deep questions about the behavior of their systems, often with little or no support. I want to make this task easier by building methods and tools that help practitioners write correct and secure systems, while keeping programmer effort low. This often requires distilling core insights from the application domain: while my work draws from programming languages and verification, my research at UCSD has focused on tailoring these techniques to problems in various application areas, including distributed systems and hardware security. I see this research as part of a larger effort to enable true computational literacy by building tools that enable, not only specialists, but everyone to pursue complex projects of their own choosing.

Distributed Systems

Asynchronous distributed systems – the foundation of cloud computing – are viciously hard. Minor upsets like lost messages, slow nodes or failures can trigger cascades of unintended behaviors that spiral out of control [2]. A promising path towards more confidence in their correctness is to equip them with *mathematical proofs* which ensure that *any* of their infinitude of possible behaviors – proofs need to hold for any number of participating nodes – avoids failure. While proofs offer tantalizing guarantees, they come at a considerable cost which limits adoption: for example, the verified key-value store in [5] took 3.7 person years to complete.

Pretend Synchrony During my post-doc I developed an idea called *pretend synchrony*, which aims to reduce verification effort by soundly treating distributed cloud programs as if they were executing *in lock step, on a single machine*, thereby slashing the number of relevant behaviors – and with it proof complexity – to a degree that humans and machines can stomach. At its core, the technique combats asynchrony using Liptons method of reduction [12]. Intuitively, one can move every receive up to its matching send and thereby fuse the pair of asynchronous operations into a single synchronous assignment.

Consider, for example, the classic two-phase commit protocol [11], that is used, *e.g.*, to implement database transactions [3]. The protocol orchestrates the communication between a coordinator trying to commit a transaction and a number of storage nodes. In the protocol's first phase, the coordinator issues a tentative transaction. It then waits for answers from the storage nodes who either accept or abort. This initiates the second phase: if all nodes agree, the coordinator sends them a commit message; otherwise, it aborts. Finally, each node sends an acknowledgement.

Due to its asynchronous nature, proving correctness (*i.e.*, if committed, all storage nodes indeed agree on the same transaction) is difficult – even for such a simple protocol. A proof needs to case-split over the local state of individual nodes and describe network state. For example, the proof needs to consider the case where a storage node did not receive a tentative transaction message, but the message is currently in transition. Similarly, it needs to consider the case where the node received the message but did not reply yet, etc. Both case-splitting and network state make proofs hard to find and check.

Pretend synchrony combats this complexity by building on the insight that, even though the protocol is distributed when executed, we can soundly *pretend* that it is executed *one after the other*, on *a single machine*, when reasoning about its correctness. First, since nodes do not share memory, most actions by different nodes are independent and commute. For example, a proposal sent to one of the database nodes is independent of the messages to and from other nodes and, using reduction, we can limit ourselves to executions in which the proposal is received immediately after it is sent. Unfortunately, this reasoning breaks down in the second part of the protocol: what if there are multiple matching sends? For example, the coordinator is waiting to receive an accept or abort message from *any* storage node. The second crucial insight behind pretend synchrony is that, even though correct protocols contain races, these races are *well-structured*. For example, the races in two-phase commit are symmetric – all the storage nodes execute the same code and thus have the same future behavior [8]. Even though there is non-determinism with respect to *which* accept or abort message is received first, for verification, their order doesn't matter.

Using these insights, a proof can reason about the protocol's synchronization where all messages are directly received. This simplification has tangible implications for the proof: a simple loop invariant stating that, in the case of a commit, all nodes indeed assigned the proposed transaction is enough.

Applying Pretend Synchrony How can we leverage these ideas to help programmers write well-structured programs that minimize proof efforts? In search for an answer to this question, I have developed the following two verification methods.

Brisk (OOPSLA'17) Brisk [4] computes synchronizations for programs written in a library build on top of Cloud Haskell. We have used Brisk to build several software systems, including a MapReduce-framework and the Disco distributed file-system [1]. Brisk's library offers types, communication primitives and iteration constructs that help programmers structure communication and a *compiler* that rewrites programs into their synchronizations. Brisk does not require user annotations and automatically proves that a program is synchronizable and thereby deadlock free. Brisk only needs tens of milliseconds to check if a program can be synchronized (if not, it computes a counterexample). This makes it the first *first concurrency verification tool that is fast enough to be used interactively.*

Goolong (POPL'19) [17] extends these ideas to prove not just deadlock freedom, but functional correctness of realistic, complex consensus algorithms including Paxos [10] and Raft [13]. This requires dealing with message drops, node failures and intricate communication patterns. To prove correctness in Goolong, the user supplies "synchronous invariants", *i.e.*, invariants that talk about the synchronized program. I implemented Goolong in Go and used it to implement Raft leader election, single-decree Paxos, and a Multi-Paxos key-value store. Pretend synchrony simplifies verification, sometimes drastically. For example, a comparison of the verification effort against the state-of-the-art approach used by IronFleet [5] showed that *synchrony reduces the number of manually specified invariant annotations by a factor of 6.* Pretend synchrony also *shrinks the time taken to check the programs by three orders of magnitude* – from over twenty minutes with Dafny to just two seconds. This is not simply an artifact of careful engineering. Instead, there is a theoretical explanation for the speedup: synchronization often makes undecidable verification conditions decidable. Moreover, pretend synchrony does not affect performance: our implementation outperformed other verified key-value stores while staying within a 3x bound of an unverified state-of-the-art implementation

Sharpie (PLDI'16) Many of the proofs in [17] require reasoning about cardinalities, which is outside the scope of traditional verifiers such as Z3. I developed the foundations for this work in my dissertation research. In particular, I built a method called Sharpie [16] that automatically constructs expressive invariants that refer to the cardinality of some set of nodes. Sharpie *is the first technique that can automically construct such invariants.* Notably, Sharpie automatically determines *what to count* to prove a given property.

Hardware Security

More recently, I developed a method called Iodine ([18]) that automatically proves that a hardware design (*e.g.*, an FPU) exhibits no timing variability, *i.e.*, is *constant time*, given some assumptions on its usage (*e.g.*, no divisions are performed). This ensures that the hardware does not inadvertently leak "secret information" (*e.g.*, cryptographic keys) by taking different times to process inputs. Unlike previous work, this approach does not require constant time to hold unconditionally nor enforces it through a proxy like information flow control. This makes our method particularly suitable for checking existing hardware systems. We used our method to prove absence of timing variability (and find timing violations) for a number of hardware designs – including RISC-processors, FPUs and crypto cores, which we took from online source repositories.

Future Directions

I will now first discuss some of my near term and then longer term research plans.

Distributed Systems Transforming pretend synchrony from a simple whiteboard idea into a full algorithmic verification method was exciting, often challenging, but ultimately very rewarding. Even though the idea has matured significantly, there are many directions left to explore. I would like to extend pretend synchrony's theoretical foundations, *e.g.*, explore how to make proofs more modular by embedding pretend synchrony in a type or effect system [6, 7, 14], provide semantic descriptions of classes of synchronizable programs and broaden its scope, *e.g.*, by exploring how protocols with topologies [15] or protocols in a byzantine setting, *e.g.*, block-chain algorithms can be synchronized. On a practical level, I want to increase usability. For this, I want to explore *automating proofs* of distributed systems, *i.e.*, freeing the user of the burden of writing proofs altogether. While pretend synchrony gives us a lever to attack this problem, the task has interesting challenges: despite their impressive abilities, modern solvers are fundamentally constrained by the undecidability of the underlying decision problems. An interesting challenge lies therefore in designing languages that allow users to supply necessary hints to solvers in a natural manner. Usability can also be increased by helping users to *fix their code*, if it is incomplete or wrong. Since, to be synchronizable, code needs to be highly structured – a loop of sends implies existance of a matching loop of receives – this structure, along with a suitable specification, can be exploited to help guide synthesis.

Pretend synchrony can not only help simplify verification, but also make testing more efficient by skipping trivially equivalent schedules and focusing computation effort on schedules that have the potential to trigger different behaviors, only. I would like to investigate how to apply pretend synchrony in testing, *e.g.*, by exploiting high-level program structure that the programer makes explicit via user annotations.

Hardware Security Besides proving constant time execution of real hardware (ARM's data independent timing instructions promise to execute in constant time; we are in the process of talking to ARM about using IODINE to verify this) and simplifying usage (*e.g.*, by synthesizing preconditions under which a given hardware design is secure), I want to devise full-stack end-to-end guarantees for eliminating side channels.

Security under Realistic Execution Models My work often focuses on handling asynchrony – between independent network nodes who pass messages in a distributed system or between independent threads communicating through shared wires and registers in hardware. Another instance in which asynchrony complicates reasoning in unpredictable ways has recently emerged in hardware and threatens to invalidate current methods for proving security properties like information flow safety. Existing methods presuppose an execution model in which only valid branches are executed, one after the other, however, modern processors may asynchronously "fork" an execution of invalid branches whose results are rolled back, but which can modify the cache. Unfortunately, security guarantees given under the traditional models are no longer valid in the speculative setting: computation paths that were never deemed to execute can leak secrets through the cache. Speculative execution is by far not the only way in which the classic execution model differs from realistic modern processors. I want to explore theoretical background to work towards clean semantic foundations for realistic execution models of modern processors and build practical methods for strong security guarantees in their presence, building on my experiencec from [18].

Event loops for IoT Internet of Things (IoT) devices are often built using asynchronous event-driven loops that operate under real time constraints. I want to built methods to help eliminate subtle concurrency bugs – and their potentially devastating physical consequences – from IoT devices.

Modeling Blockchain Correctness Via Logic Based on my experience from [19], I want to model necessary preconditions for the correctness of block-chain algorithms using epistemic and temporal logic.

Biological Models Finally, I want to explore new application areas. Asynchrony is at the heart of biological models like gene regulatory networks. Cells communicate (asynchronously) through proteins (messages) whose presence in turn increases or inhibits production of (potentially different) proteins in other cells. Different orders of release might trigger different cell fates [9] – much like different outcomes of races can lead to different final states in programs. I would like to investigate how the methods for combating asynchrony in software and hardware can help build methods for biologist to understand and model cell behavior.

References

[1] http://discoproject.org.

[2] https://aws.amazon.com/message/41926/.

[3] https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_txns003.htm#ADMIN12222.

[4] Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. Verifying distributed programs via canonical sequentialization. In *OOPSLA*, 17.

[5] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *SOSP*, 2015.

[6] Kohei Honda. Types for dyadic interaction. In *CONCUR '93*.

[7] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

[8] C. Norris IP and David L. Dill. Better verification through symmetry. In *Formal Methods in System Design*, 1996.

[9] Ali Sinan Köksal, Yewen Pu, Saurabh Srivastava, Rastislav Bodík, Jasmin Fisher, and Nir Piterman. Synthesis of biological models from mutation experiments. In *POPL '13*.

[10] Leslie Lamport. Paxos made simple. In *https://www.microsoft.com/en-us/research/publication/paxos-made-simple/*, 2001.

[11] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. In *Technical report XEROX Palo Alto Research Center*, 1976.

[12] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. In *Communications of the ACM*, 1975.

[13] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC '14*, 2014.

[14] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP 2009*, 2009.

[15] Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[16] Klaus v. Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, 2016.

[17] Klaus v. Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *POPL*, 2019.

[18] Klaus v. Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. Verifying constant time execution of hardware. In *Draft*.

[19] Klaus v. Gleissenthall and Andrey Rybalchenko. An epistemic perspective on consistency of concurrent computations. In *CONCUR*, 2013.