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 run-time behavior of their systems, often with little or no support. I work on making this task easier by building methods and tools that help practitioners write correct, secure and performant systems, while keeping programmer effort low. This often requires domain insights: 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.

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 observing that even though the protocol is distributed when executed, when reasoning about its correctness, we can pretend that it is executed one after the other, on a single machine. 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. We can therefore pick an arbitrary winner.

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? I have developed these ideas in 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 complicated 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 automatically 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, our 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. To measure timing in the presence of pipelining, we rely on the notion of live value. A value is live if it belongs to the current (i.e., not some outdated) computation. For a design to be constant time, live values must take the same time to flow from input to output, in any two executions that satisfy the usage assumptions. 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 Work

I am excited to build on these ideas, over the course of the coming years. Below, I will discuss some of my near term and then long term plans.

**Distributed Systems** Besides exploring pretend synchrony’s theoretical foundations (e.g., how to make proofs more modular by embedding pretend synchrony in a type or effect system \([6, 7, 14]\)) and extending its scope (e.g., by exploring how protocols with topologies \([15]\) or protocols in a byzantine setting, e.g., blockchain algorithms can be synchronized), I want to work on making pretend synchrony more usable. For this, I want to explore automating proofs of distributed systems, *i.e.*, freeing the user of the burden of writing proofs altogether. While the simplification of proofs that pretend synchrony achieves gives us a lever to attack this problem, the task still presents some interesting challenges: even though modern solvers can find complicated proofs, their power is not unlimited – this is unlikely to change due to the undecidability of the underlying decision problems. An interesting challenge lies therefore in designing languages that allow users to supply necessary hints and directions to solvers in a way that is natural for the user. Another way to improve usability is to help users *fix their code*, in case it is incomplete or wrong. Since, to be synchronizable, code needs to be highly structured, *e.g.*, the presence of a loop sending a message to a number of clients implies that those clients must have matching receive statements of the correct type, this structure, together with a suitable correctness specification, can be exploited to help guide synthesis tools. My experience from \([4, 17]\) and \([16]\) puts me in a unique position to pursue these directions.

**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 end-to-end guarantees for eliminating (timing) side channels from the software/hardware stack. This requires building infrastructure that can prove suitable guarantees on software, threading these guarantees through a compiler and finally, verifying that the guarantees match proven assumptions on hardware. Such end-to-end guarantees open up avenues for increased performance: while a security aware engineer might omit certain optimizations for fear they might jeopardise system guarantees, a system with proofs allows performing aggressive optimization whenever possible: if the system passes the check, we know that these optimizations are indeed secure.

**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 expected 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 as well as practical tools that allow providing strong security guarantees, in the presence of a realistic execution model for modern processors.

**Biological Models and Cryptographic Algorithms** 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. Asynchrony is also key in cryptographic protocols. These protocols come with an additional complication: cryptographic protocols rely on probabilistic arguments. This makes (manual) reasoning challenging and calls for automation support.
References


