Billions of people rely on correct and efficient execution of large systems, such as the distributed systems that power Google and Facebook. Yet these systems are enormously complex and are challenging to build and understand. My research aims to improve the design, implementation, and operation of large systems by using techniques from the areas of **systems** and **software engineering**. I design, implement, and deploy a variety of experimental research systems, then leverage these experiences to develop techniques and tools that help developers of large systems better understand and debug their implementations.

Distributed systems

In building systems, I focus on formulating new designs and organization principles. The following is a summary of some of the systems that I have contributed to.

1. Scatter: scaling consistency

A distributed hash table (DHT) distributes a set of keys across multiple machines to achieve fault tolerance and scalability. To access a key's value in the DHT, a client sends a request to a machine in the system, which routes the request to the right machine. Traditional proposals for DHTs suffer from data and routing inconsistencies. These inconsistencies complicate the design of systems that use DHTs as middleware. Scatter [C2][1] is a DHT design achieves serializable consistency semantics for all operations in the system without sacrificing performance, scalability, and other DHT advantages. Scatter's key contribution is an abstraction of coordinating replica sets, which are implemented as groups of machines running the Paxos protocol.

2. Sonora: a platform for continuous mobile-cloud services

Numerous mobile applications stand to benefit from cloud computing and storage. However, mobile devices continuously generate information (e.g., GPS readings), while compute platforms, such as Hadoop, are designed for batch processing. The Sonora platform [N3] bridges this gap by coherently integrating a broad range of existing techniques, from streaming databases to mobile networking. Developers use a high-level event-driven API and a framework based on LINQ to express continuous mobile-cloud services with a single language. The platform supports disconnected operation, basic synchronization primitives, and other features essential for robust mobile applications. On the cloud side, Sonora implements advanced load balancing and failure recovery algorithms to provide scalable, responsive, and fault-tolerant computation.

3. Seattle[2] and SatelliteLab distributed testbeds.

Systems researchers rely on testbeds to evaluate their prototypes in realistic environments. The Seattle testbed [C4, C5] harnesses resources contributed by volunteer participants from around the world to provide researchers with a highly realistic peer-to-peer distributed testbed composed of a wide variety of devices. Seattle's design solves the unique challenge of executing untrusted code on user machines. Seattle has been used at over a dozen universities world-wide for teaching courses in networks and distributed systems. The goal of the SatelliteLab testbed [C6] is to improve heterogeneity of an existing parent testbed, such as PlanetLab. The key idea is to separate computation from traffic routing: computation resides on over-provisioned *planet* nodes, associated with the parent testbed, while light weight *satellite* nodes route traffic between planets. By placing satellites in edge networks, the traffic is exposed to more realistic Internet network conditions, which improves network heterogeneity and testbed realism.

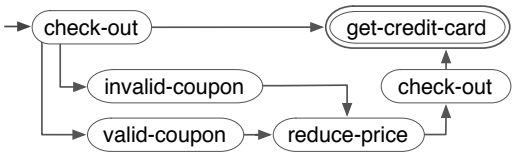
Modeling systems from observations of their behavior

My own experiences in system building motivated me to address the challenges that developers and operators face in debugging, evaluating, and *reasoning* about large systems. A key challenge is knowing whether an implementation does what it was designed to do. My research helps developers make sense of their systems by leveraging the extensive logs generated by large systems. I have developed techniques to infer

[1] References can be found in my curriculum vitae.
[2] https://seattle.cs.washington.edu

models from logs of sequential and distributed systems, and I have implemented these techniques as part of the **Synoptic** [C3, J1] and **Dynoptic** [U1, N1] tools. In evaluating these tools, we found that the generated models helped developers identify new bugs, confirm existing bugs, and increased developers' confidence in their implementations. Both tools process the logs most systems already produce and require developers only to specify a set of regular expressions for parsing the logs. To unify model inference algorithms I developed **InvariMint** [C1, N2], a technique to specify model inference algorithms declaratively. I will now describe each of the three tools in more detail[3].

1. Synoptic: inferring models of sequential systems



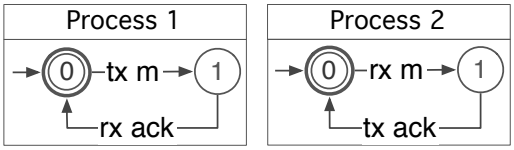
Synoptic infers a concise and accurate system model in the form of a finite state machine (FSM) from a set of logged execution traces. Developers can use these models to better understand their systems' behavior and find bugs. For example, the shopping cart application model at the right is simple to inspect and the buggy transition (`invalid-coupon` to `reduce-price`) is easy to spot, while the same bug would be hard to catch in a log. Synoptic models can also aid verification and test-case generation.

Two features distinguish Synoptic from other tools. First, Synoptic's models preserve key temporal invariants mined from the log, making them more accurate. Second, Synoptic uses refinement to derive the model, which is more efficient than traditional coarsening algorithms.

Synoptic's input is a log of system execution traces, each of which is a totally ordered sequence of events. (1) Synoptic mines a set of temporal invariants that capture the essential behavior traits of the system, such as "`open` *is always eventually followed by* `close`" and "`rcv` *always precedes* `snd`". (2) Synoptic builds an initial FSM model that accepts all of the logged executions, but also many invalid executions. (3) Synoptic iteratively identifies invalid paths in the FSM (those that violate the mined invariants) and eliminates them by refining the model. The final Synoptic-derived model always satisfies all of the mined invariants and is a locally-minimal model (finding the global minimum is an NP-complete problem).

Synoptic was applied to logs generated by the Reverse Traceroute system, which is deployed at a large Internet company. In the first five minutes of inspecting the model a developer identified a previously unknown bug, and confirmed an existing bug and better understood its scope. In a distributed systems class, students found protocol bugs with Synoptic-generated models and reported that the tool increased their confidence in their implementations.

2. Dynoptic: inferring models of networked systems



Synoptic works on totally ordered logs, like those generated by a sequential program. By contrast, Dynoptic infers a communicating finite state machine (CFSM) from a partially ordered log generated by a distributed or concurrent system. A CFSM model describes each process as an FSM extended with communication events; processes communicate by sending and receiving messages over reliable FIFO queues. The model to the right is an example CFSM for a two-process system that describes a simple stop-and-wait protocol: process 1 transmits a message (`tx m`) to process 2, which receives it (`rx m`) and responds with an `ack`. CFSMs are intuitive and simpler to comprehend than alternative formalisms (e.g., a Petri net). For example, a single process FSM in a CFSM can be inspected and understood without needing to understand the other process FSMs. Yet, CFSMs are powerful enough to model protocols and distributed systems.

Dynoptic is the first tool to automatically infer CFSM models from observations. Prior tools that infer CFSMs require significant manual user input, or are theoretical proposals that have not been experimentally validated. We evaluated Dynoptic by using it to reconstruct the models of the TCP opening and closing handshakes, as well as the replication strategy in Voldemort, a data storage system deployed at LinkedIn.

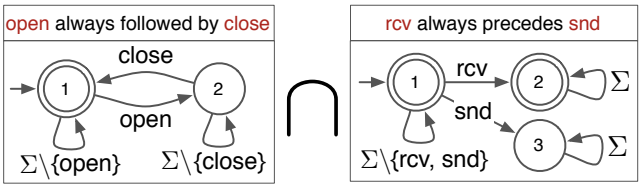
3. InvariMint: model inference as combination of model properties

Many model inference algorithms have been proposed in the research literature. These proposals vary in how they describe the algorithms, which makes it difficult to compare, combine, and understand these

[3] All tools are open source: http://synoptic.googlecode.com

algorithms. In coming up with new model inference algorithms, I realized that many algorithms can be expressed as combinations of ordering constraints mined from the input executions. That is, the models these algorithms infer always satisfy particular properties of the observed executions.

For example, the two FSMs on the right capture two example invariants that an invocation of Synoptic might preserve. Further, because intersection of FSMs can be done efficiently, the InvariMint-specified model can be derived faster than with Synoptic.



The above example illustrates the InvariMint approach to model inference, which uses a set of formalisms to express model inference algorithms *declaratively*. By specifying Synoptic and kTails (one of the most widely used model inference algorithms) with InvariMint, we showed that InvariMint leads to new insights and better understanding of existing algorithms. As well, InvariMint simplifies the creation of new algorithms, including hybrids that extend existing algorithms by including new property types. InvariMint also makes it easy to compare and contrast different approaches and is over 100 times faster than equivalent procedural implementations.

Software systems are necessarily social—the people involved and their practices (e.g., code review habits) shape the software artifact as much as the tools and technical specifications. My research in computer supported collaborative work focused on the practices of editors in the Wikipedia community [C7, C8, C9]. This work deepened our understanding of editor work practices, the site's policy and governance structure, and the influence of rhetoric and editor power on article content. This work received a **best paper award** at ICWSM 2008 and a **best paper nomination** at CSCW 2008.

Future research directions

I plan to continue bridging the areas of systems and software engineering by empowering systems developers with tools that help them understand and debug their implementations.

Near future: new approaches to model inference

In practice, model inference tools are unwieldy, as they only capture those behaviors that have been observed, and take a long time to run. In large systems, the set of observed behaviors is never complete, and it is expensive to re-compute the model whenever the system changes or new behaviors are observed. I will design techniques to update the inferred models to accommodate new behavior and remove old behaviors (e.g., if some behaviors are no longer valid because of a bug fix).

Current model inference algorithms focus on events, and reason only implicitly about state (e.g., as sequences of events). However, logs often contain *explicit* state information, or this information can be captured by instrumenting the system. I plan to develop techniques to explicitly integrate state into the model inference process to infer more accurate and richer models. The resulting models, with user-defined state labels, will be more comprehensive and can be used for new kinds of model-based analysis (e.g., test generation guided by abstract state coverage).

Longer term: testing large systems and energy debugging

One challenging aspect of building large systems that interests me is testing—large systems are difficult to test because they have many possible behaviors. Most systems developers, therefore, concentrate on testing a small set of carefully selected code paths. I want to understanding how developers determine which behaviors are most important to test, and will develop techniques to automatically generate tests for large systems. For this, I plan to leverage symbolic execution, repository mining techniques, and formal methods.

I plan to expand my focus towards important new problem domains, such as energy debugging on mobile devices. Energy bugs are a serious issue for mobile application developers. However, there are few tools that can help developers understand the energy profile of their code. I plan to develop dynamic and static analysis techniques that can guide developers to energy hot-spots in their code.