

Research Statement

Jinlin Yang

Always passionate about research that has practical value, I am interested in developing techniques for building dependable software systems, implementing those techniques in practical tools, and applying those tools to solve problems in the real world.

For my Master project, I, along with colleagues, conducted a case study about Bounded Exhaustive Testing (BET) on a large system that models and analyzes dynamic fault trees (DFT). BET tests a system by running all valid inputs that are within some size boundary. We found that BET can generate many “unusual” corner cases that detect bugs in the original system [ISSTA 04, TSE 05]. My main contribution was developing an abstract DFT grammar that enabled efficient generation of all valid test inputs up to a bound that is large enough to cover most feature interactions. Our paper was selected as one of the five best papers of ISSTA 2004.

For my PhD thesis, I developed a novel approach for generating a system’s finite-state-machine models and conducted extensive experiments on real systems. My key contributions include 1) developing a hierarchy of templates that capture the interesting relationships among events [PASTE04], 2) inventing and implementing an approximate algorithm that can handle imperfect traces, 3) devising and evaluating several heuristics for selecting interesting properties (e.g., locking disciplines or resource allocation/de-allocation), one of which combines static and dynamic analyses, and 4) experimentally demonstrating that my approach is scalable, effective, and useful in aiding program evolution [ISSRE 04], understanding, and verification on large systems including Windows kernel and JBoss [ICSE 06]. Next, I describe my thesis contributions and plans for future work.

Automatic Inference and Effective Application of Temporal Specifications

Formal verification is an important technique for addressing the grand challenge of building reliable software systems. Many verification tools require specifications of properties that a target system needs to satisfy. From a developer’s perspective, understanding these properties is important for writing correct programs. Most software systems use existing libraries, for which it is crucial to follow certain protocols. For example, *what rules should I follow when calling this function? What sequence of functions should I call when accessing this resource?* Ideally, the developers of the library should document these rules precisely. Unfortunately, such documentation is rarely available, especially for internal libraries, because correctly specifying these properties by hand is very tedious and error-prone. Even when such specifications exist, it is difficult to keep them up-to-date. Without such specifications, developers are forced to figure out the right rules through ad-hoc approaches, which often results in inconsistent usage. The unavailability of specifications also prevents the wide adoption of formal verification.

Hence, my work focused on an automatic way to generate specifications. These specifications can greatly facilitate the development and verification of dependable software systems. In addition, automatically generated specifications can also aid program evolution. For example, when modifying an existing system, the developers want to make sure changes do not break properties satisfied by the current system. We could automatically compute a signature for both the new and old programs. Then we can check whether the new version preserves those important properties by comparing the two signatures.

My Approach

I developed an approach for inferring finite-state-machine models from execution traces [PASTE 04]. FSM specifications constrain the occurrence and order of events. They can be used to represent application-specific requirements that are essential for correctness. My approach instruments a target program and runs a set of test inputs to produce execution traces. Then, the inference engine tries to match the traces against a set of pre-defined templates, producing a list of properties that the traces satisfy. Finally the post-processor removes the noise in the properties and presents a report to the users. I implemented my approach in a tool called *Perracotta*.

Many earlier inference techniques attempted to directly extract a complete FSM model with many states from the trace. Such approaches do not scale to realistic systems due to the NP-hardness of the fundamental grammar inference problem. Perracotta follows a novel approach by first inferring FSMs with a few states and then building more complex FSMs out of the smaller ones. The algorithm explores the logical relationships among properties to group related properties and remove redundant ones. For example, Perracotta inferred a 24-state FSM for the JBoss transaction management module that captures the object interaction diagram in the J2EE specification.

My initial experiments applying Perracotta to several small programs were encouraging. For example, I used it to compare the implementation of the hand-shake protocol in several versions of OpenSSL. The differences in the inferred FSM models led us to identify fixed bugs in older versions and to demonstrate that some desirable properties had been preserved as OpenSSL evolved [ISSRE 04].

However, several challenges arose when I applied Perracotta to larger systems. Perracotta did not infer some of the interesting properties that we expected it to find. In addition, Perracotta also produced many uninteresting properties. Next I will describe these two problems in detail and present some of the techniques I developed to address them. Details of our techniques, experiments, and results will be presented at ICSE 2006 [ICSE06].

Dealing with Imperfect Traces

As an intern at Microsoft in the summer of 2005, I applied Perracotta to several Windows kernel execution traces collected by developers for performance tuning or debugging. Initially, Perracotta did not infer some interesting properties because the traces are imperfect. There are several reasons for this. The most insurmountable one is the target program might have faults, which prevent the execution traces from recording perfectly correct behaviors. Although it is possible to remove traces that exhibit obviously wrong behaviors (e.g. crashes), there is no general way for identifying error traces when the faults are in the program's semantics. Therefore, a faulty program can produce traces that violate required properties, even though it runs without obviously abnormal behavior. Sampling or insufficient tracing infrastructures can also produce imperfect traces. For example, because of limitations of our instrumentation tools, our Windows kernel traces did not have object information (e.g. lock or handler) that is essential for recognizing type-state properties.

To address the challenge of handling imperfect traces, I developed a statistical algorithm that can detect *dominant* behaviors from imperfect traces. Instead of deciding whether a trace satisfies a template, the new algorithm partitions the trace into fragments and computes the *probability* that a fragment satisfies the template. We then rank the properties based on their satisfaction rate. This new algorithm can infer those interesting properties that are true most of the time in the traces and that would otherwise be missing because they are not completely satisfied by the traces. Using the approximation technique, we discovered many essential properties that were true only 90-99% of the time in the traces.

Heuristics for Selecting Interesting Properties

A general problem in specification inference is the burden of selecting interesting properties from the inference results. My initial experience of applying Perracotta to larger systems also reflected this problem. For example, my preliminary Windows experiments produced thousands of properties, the vast majority of which were uninteresting. Without an effective selection approach, such techniques can hardly be practical. Colleagues working in this area also reported similar experiences with other inference techniques, which emphasizes the need for addressing these problems that limit the usefulness of dynamic inference techniques on large systems.

To address this problem, I developed several heuristics for identifying the interesting properties. One heuristic uses a static call graph. A property is considered interesting only if the relationship between events is not captured in the call graph. This captures the intuition that such properties correspond to ordering constraints that are more easily forgotten. The second heuristic considers a property to be more interesting if its events are more similar lexically. This reflects the fact that many real systems are developed following naming conventions. Applying these heuristics to the Windows experiments was very fruitful. They

reduced the number of properties from 7600 to 142, a manageable number for manual inspection. A developer inspecting the properties found that 40% of the remaining properties are interesting.

Windows Results

It is very important to experimentally validate a new technique. Therefore, I devoted a significant portion of my research to conducting comprehensive experiments on large-scale systems including Windows and the JBoss application server. I applied Perracotta to analyze 17 Windows kernel traces containing 5.8 million total events. Using the heuristics described above, Perracotta identified 142 temporal rules for the Windows kernel APIs. A developer found 56 of them are obviously interesting, most of which are undocumented specifications about locking disciplines or resource allocation/de-allocation. We compared the inferred rules with the list of rules checked by the SLAM model checker (a public tool developed at Microsoft Research that is able to verify temporal safety rules for device drivers). We found many of the properties SLAM checks, as well as two properties it could check but does not check currently. We also checked the properties with the ESP static verifier and detected many previously unknown bugs in Windows. For example, one bug is a double-acquire of a lock in the NTFS file system that can potentially cause deadlock. These bugs were confirmed and fixed by the development team.

Future Research Plans

Property Inference

My future research includes expanding my dissertation work to infer more interesting and expressive specifications, to build better inference algorithms, and to study uses of inferred specifications in software development. In terms of more interesting specifications, one possible direction is to investigate how to combine temporal properties with data properties. Previous work on specification inference focused either on temporal or data invariants. Real program properties, however, often involve both aspects. A technique for extracting temporal data-properties can be used to more precisely capture and reason about a system's behaviors. Another possibility is to develop scalable techniques for inferring FSM models involving more states and more complex interaction. Because the fundamental problem of grammar inference is NP-hard, here the key challenge is to make the approach scalable. I have shown that it is feasible to efficiently build complex FSMs out of many simpler ones. I will develop more advanced mechanisms to build complex models from simpler ones.

In terms of better inference algorithms, I plan to study how to integrate static and dynamic inference techniques. The quality of dynamic inference depends on the quality of test inputs used to produce the traces. Sometimes it can be difficult to generate a high-quality test suite. I will explore how to use static inference to complement the dynamic inference results.

Formal specifications allow mechanical reasoning about large-scale software systems. I want to explore other applications of inferred properties. An interesting application area is dynamically enforcing safety properties. We could instrument a program to summarize its run-time behavior on-the-fly and raise warnings when deviant behaviors occur. Here the key challenge is to adapt the existing approach to minimize the performance degradation and limit the false positive rate. Another avenue of research is to investigate how to use inferred temporal specifications in bug localization or post-mortem analysis. Given a failure run and one or more successful runs of a target program, we could compute a property signature for each execution. We then summarize their key differences to provide a starting point for analyzing what might go wrong. Another research direction is to study how to use inferred temporal properties to select and prioritize test cases. Traditional statement, edge, or functional coverage criteria ignore the sequence of execution. On the other hand, path coverage is too expensive to realize. The temporal properties inferred by Perracotta provide a middle ground between the two ends. I plan to develop a new temporal test coverage criterion and study how to use it in test selection.

Other Directions

The increasing need for reliable software in diverse environments poses challenges and also provides

exciting opportunities for research. Software has been widely used in embedded systems (e.g., mobile phones and PDAs). Such systems demand higher reliability in the software because it is expensive to debug and fix problems after deployment. There are still many long-standing issues in how to effectively ensure the reliability of such software. In addition, more and more software is deployed as a web application. This allows users to easily access applications through a universal interface without the burden of installing new applications. Web applications are typically updated very frequently. This raises many challenging issues that have been largely ignored by the software engineering community. For example, how to assure the reliability of each update with the limited available time, how to ensure a safe update while the client sessions are still running, and how to change the data representation without damaging existing data.

I believe combining static and dynamic analyses could provide an effective solution to these new challenges. For example, the call graph heuristic I developed combined dynamic inference and static analysis to select interesting properties. My long-term research direction will focus on studying different ways to combine static and dynamic analyses. I plan to develop a framework for exploring different approaches and experimentally evaluate it on realistic systems.

Summary

In summary, I enjoy developing fundamental techniques for solving real problems in building reliable systems. I especially like the approach of experimentally evaluating my research on large systems. I want to evaluate my future research in a similar way. In addition to writing papers and giving presentations at conferences and seminars, I have made significant contributions to two grant proposals my advisor submitted to the NSF, supervised an undergraduate research student, and served as an external reviewer for numerous conferences. I am confident that I am well prepared for a productive academic career.