My research focuses on self-adaptative software systems, developing techniques that infer models of software behavior, help both developers and the systems themselves to reason about that behavior, and enable systems to self-modify to improve their own behavior at runtime. I work with a broad community of researchers to identify the key challenges and a research roadmap for self-adaptive systems [B1, B2, B3, B4] and focus my own research on two thrusts: (1) automatically inferring precise models of software behavior [C9, J15, J21, C25, C24, C15, C17, C18, C20, J11, H14, H12, W8] and (2) automatically repairing software defects via behavioral reasoning [C26, C23, C19, J20, N7]. My research vision is for these two thrusts to come together to enable software systems to reason about their own behavioral models and to manipulate themselves to self-heal and adapt to changing environmental conditions and requirements.

I next list the research results I am best known for. This statement focuses on the first two results and on my future research vision.

- Techniques and tools for automatically inferring precise models of software behavior for serial [J15, C20, C15, C9, W8, S8, S5], distributed [J21, J11, C17, S9], and resource-constrained [C18, H12, S11] systems, including advances in interactive user interfaces for behavioral understanding [C18, H14, H4].
- Methods for measuring automated program repair patch quality [C23, N7], benchmarks for quality evaluation [J20, S12], and techniques for improving patch quality via behavioral reasoning [C26, C19, S13].
- The key challenges, a research roadmap, and evaluation benchmarks for self-adaptive software systems research (with a broad group of contributors) [B4, B3, B2, B1, C11, S4].
- Techniques and tools for using behavioral software models to improve software quality by identifying software defects, security vulnerabilities, and data errors [J17, J19, C25, C24, C22, C8, H11, H9, W11, S3].
- Identification of critical challenges in collaborative and interactive software development, and techniques, tools, and development environments to address these challenges [J16, J14, C16, C14, C10, H13, H8, S10, S7, S6].
- Improvements in patient care processes in hospital emergency departments via applying software engineering techniques, such as process-driven simulation and resource specification, to the healthcare domain [J18, C21, W13, W12].
- Technology transfer to industrial partners, including Microsoft using the collaborative development results [J14, C10, S6] to improve internal development practices, and Google using the behavioral model inferencing results [C18, C9, S11, S5] to improve software testing practices.

My approach to research is highly collaborative. It involves developing complex algorithms and implementing them within systems, applying these systems to large real-world software, and evaluating these systems via user studies with real users and large-scale benchmarks. These activities benefit from scientists with diverse expertise and thus my research often involves diverse collaborations. Since joining UMass in September 2012, I have co-authored with senior researchers all over the world [J21, J20, J19, J18, J16, J15, C25, C24, C20, C19, C17, C16, C15, C14, H13, U2], while achieving a good balance of publications led by my students [C23, C22, C21, C18, H9, H14, H12, H11, H10, W13, W12, W11] (some led by me [J17, J14, J13]), and publications involving no faculty more senior than myself [C26, C23, C22, C18, H14, H12, H11, H10]. I enjoy collaborative research and find it effective at producing high-impact results that can transition to industry and affect modern software development.

References to papers I have authored (ones prefixed by a letter, such as [B1]) refer to my curriculum vitae.
1 Precise behavioral model inference

Reasoning about software behavior above the level of small code fragments requires automatically modeling complex software behavior. Such models can accomplish multiple goals. They can enable automated reasoning, as is necessary for self-adaptive systems [B1], and help developers reason about software and reduce the inconsistency between what developers think their system does and what the system actually does. This inconsistency is a significant cause of software defects [5, 13].

A key requirement of such behavioral models is that they describe the behavior encoded by the software implementation, as opposed to the intended behavior represented in the documentation or in manually created models. Developers already rely heavily on logging to peek into the implementation’s behavior — logging is one of the most ubiquitous, simple, and effective debugging tools, and production systems at companies like Google generate billions of log events each day [19] — so these readily-available data-rich logs are a natural resource for inferring precise behavioral models.

Together with collaborators, we developed Synoptic [C9, S5], a model-inference algorithm that precisely and concisely summarizes logs. Synoptic infers a concise finite state machine (FSM) whose language contains all the observed executions while precisely generalizing to unobserved executions that the system is likely to be able to produce. To do this, Synoptic mines temporal invariants from the log, and then creates the most compact FSM that accepts all observed executions. This initial model is very compact, but not very precise: It accepts all observed executions but it also accepts many executions the system cannot produce. Iterating over the model, Synoptic finds paths that violate the invariants and refines the model to eliminate those paths, until no more such paths exist. While the problem of finding the minimal model is NP-complete [9, 2], Synoptic efficiently finds a good approximation. This formal approach allows us to prove that Synoptic always finds a solution, that the solution is locally-optimal, and that the solution accurately describes the behavior of the system by satisfying all the mined invariants and no others of the same form [C9, C15, J15].

Examining a Synoptic model helps identify bugs. For example, the shopping cart application model in Figure 1 makes it easy to see that an invalid coupon reduces the price. Finding this behavior in the logs would have been much more difficult. We showed a developer a Synoptic model of over 900,000 executions of reverse traceroute [10], a distributed system that has handled over 16 million requests and is internally deployed by a large Internet company. Within five minutes, the developer: (1) identified a previously unknown concurrency bug that the model evidenced as unexpected paths, and (2) verified the presence of an existing bug, evidenced as unexpected termination states, discovering it occurred in more ways than expected. After fixing the bugs, the new model helped verify that the bugs were removed [C9].

**Improving precision with data-based observations.** The key Synoptic insight is that the inferred models satisfy observed invariants, so the observed invariants are critical for success. Synoptic has shown that temporal invariants specified with linear temporal logic (LTL) are sufficient to encode behavior and help debugging and understanding tasks [C9]. At the same time, Synoptic ignores much of the rich data encoded in the logs, including the runtime data values and system resource use.

By mining data-value-based invariants of program behavior [7], the inference algorithm can discern which events can happen under which conditions, improving inferred model quality, as measured by precision and recall. With collaborators, we systematically explored the model quality of four strategies for using data-based and temporal execution information [C20]. We developed a
benchmark of nine popular libraries that enables objective comparisons of model-inference algorithms. Using only temporal information led to high precision (98%), but low recall (33%). Using only data-based information improved recall (75%) while preserving a high precision (97%), but greatly reduced scalability of the inference, and required manual data-invariant filtering. Combining temporal and data-based information can improve precision (99%) while ensuring high recall (75%), scalability, and robustness to input noise [20].

Modeling resource-use behavior. In addition to runtime data values, resource use is often a critical part of system behavior. Yet existing state-of-the-art model inference algorithms, e.g., [3, 6, 8, 16] abstract away resource use to simplify the models. Unfortunately, this abstraction hides important system behavior. Executions that appear identical when ignoring resource use may, in fact, be behaviorally different. As a result, models inferred using existing techniques often fail to encode cache bugs, timeout causes, and other, resource-related behavior.

Consider an example system of a network diagnosis tool that a server can use to identify problematic client network paths. The tool first determines if the client is using narrowband or broadband and then executes a series of queries. Based on the speed and characteristics of the client’s responses to the queries, the tool classifies the network path as OK or problematic.

The tool’s developer wants to know what factors cause the tool to report client paths as problematic. Runtime logs of the tool (Figure 2(a)) can help answer this question, but the information is hard to infer manually. The k-tail-inferred [3] model (Figure 2(b)) differentiates execution paths of broadband and narrowband clients, but fails to indicate the types of executions that suggest network problems because all paths pass through the common bottom node. Synoptic’s model (Figure 2(c)) correctly conveys that no network problems are reported for narrowband clients, but fails to differentiate which broadband clients experienced a network problem.

Enforcing temporal resource-use invariants can expose the relevant behavior, albeit likely at the cost of increased model size. Together with collaborators, I developed Perfume [18, 14, 12, 11] to infer behavioral models while enforcing timed propositional temporal logic [1] invariants. Perfume’s model (Figure 2(d)) reveals which types of executions imply a network problem: a broadband execution with a fast and then a slow query.

A user study compared system comprehension when using (1) logs, (2) Synoptic models, and (3) Perfume models. The users—previously unfamiliar with three subject software systems—were shown the logs or models and asked comprehension questions [18]. The study demonstrated that both temporal and resource-use invariant models improved comprehension as compared to using only logs. Perfume models led to answering 81.4% of the system behavior questions correctly,
using Synoptic models led to 78.0%, and using logs led to 72.4%. Further, compared to using logs, the average time taken to answer questions improved by 12% (682 sec. vs. 778 sec.) for Perfume models, and by 4% (712 sec. vs. 778 sec.) for Synoptic models [C18]. Using the study’s findings on effective model and execution visualization, and on model querying, we significantly improved Perfume’s user interface, and verified its effectiveness with another user study [H14].

2 Behavior-based automated program repair

My work on automated program repair focuses on the quality of the repaired programs. Automated program repair, e.g., [14, 17, 18], holds great potential to reduce debugging costs and improve software quality. These techniques use a buggy program and a specification of that program (without the bug) to produce a program variant that satisfies the specification. However, due to the partial nature of the specification, e.g., a set of tests describing the desired behavior, patches that satisfy that specification may not satisfy the intended, unwritten specification. Prior evaluations of automated repair have not focused on the quality of the patches.

But the quality of the patches is critical to the effectiveness of automated repair and to its applicability in practice. In fact, techniques that produce low-quality patches—patches that satisfy the available specification but significantly deviate from the intended specification—could do more harm than good, breaking significant portions of the program’s behavior while fixing relatively few.

Together with collaborators, we set out to prove patch quality an important aspect of program repair. We developed the IntroClass benchmark of 998 small C programs, with defects that occurred naturally during development [J20, S12]. Each IntroClass program has two independent, high-coverage test suites that allow for an objective evaluation of quality: automated repair techniques produce a patch using the specification encoded by one test suite (by definition, the patch passes all of these tests), and the second test suite enables independently measuring the patched program quality. This important methodology reveals that because existing automated repair techniques manipulate low-level source-code constructs, instead of reasoning about program behavior, patch quality is often low. AE [17], GenProg [18], and TrpAutoRepair [14] patches only pass 64.2%, 68.7%, and 72.1% of the independent test suite, respectively [C23]. The quality of the test suite used for repair plays an important role in patch quality, with higher coverage correlating with higher quality [C23, N7].

<table>
<thead>
<tr>
<th>tool</th>
<th>quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE [17]</td>
<td>64.2%</td>
</tr>
<tr>
<td>GenProg [18]</td>
<td>68.7%</td>
</tr>
<tr>
<td>TrpAutoRepair [14]</td>
<td>72.1%</td>
</tr>
<tr>
<td>SearchRepair [C26]</td>
<td>97.3%</td>
</tr>
</tbody>
</table>

Figure 3: Patches generated to pass 100% of the known tests often fail left-out, independent tests, resulting in low-quality repairs. SearchRepair produces high-quality patches by reasoning about program behavior.

Our and others’ [15] work on evaluating patch quality has led to an important movement toward quality-aware repair [C26, 11, 12]. With collaborators, we developed SearchRepair [S13], a tool that, on IntroClass produces patches that pass 97.3% of the independent test suite [C26]. The key to SearchRepair’s success is that it operates at a higher-level than prior repair techniques, replacing larger snippets of code wholesale with other human-written code. SearchRepair uses the large body of existing open-source code to find potential fixes by reasoning about the desired behavior. The central challenges lie in efficiently finding code behaviorally similar (but not identical) to defective code and then appropriately integrating that code into a buggy program. SearchRepair addresses these challenges by (1) encoding a large database of human-written code fragments (e.g., from open-source projects) as SMT constraints on input-output behavior, (2) localizing a given defect to likely buggy program fragments and deriving the desired input-output behavior for code to replace those fragments, (3) using state-of-the-art constraint solvers to search the database for

\(^2\)A short video demonstrating Perfume is available at http://perfume.cs.umass.edu/demo.
fragments that satisfy that desired behavior and replacing the likely buggy code with these potential patches, and (4) validating that the patches repair the bug against program test suites [C26]. While significant work remains on improving SearchRepair’s scalability, its success suggests repairing defects at a higher-level is better suited for properly encoding and repairing program behavior.

3 Research vision and future directions

My work has demonstrated that (1) high-quality models of system behavior can be inferred automatically and used for manual debugging and behavior understanding, and (2) software can effectively modify its own behavior, monitoring modification quality. These findings facilitate three research directions on automating system-building tasks I plan to pursue next.

Automated test generation. Inferred models can automate test generation, which can improve software quality. Inferred models describe not only the observed behavior but also unobserved combinations of partial observed behavior that comply with the observed invariants. The paths through models inferred from logs of existing test executions that are uncovered by existing tests represent this unobserved behavior. I have already used this approach to demonstrate automated behavioral exploration to identify security vulnerabilities [C24]. Taking the next step, generating inputs (via symbolic execution [4]) to enact these executions, and observing that the system behaves as predicted by the model verifies expected behavior. Deviations from the model either indicate software faults or model inaccuracies. Those inaccuracies are indicative of shortcomings of the existing test suite. Exploring almost-paths in the model (e.g., paths that add a single transition not in the model) can generate potential new tests that are even more distinct from the tested behavior (but may yield more false positive paths). My findings that test suite quality is correlated with repair quality [C23, N7] indicate that improving test suites in this behavior-aware way can improve automated repair mechanisms.

In-field runtime monitoring. Deviations from expected behavior can arise from bugs or unintended system uses, and may require the system to respond with self-adaptation. Models inferred from test suite executions can be used to detect such deviations by comparing an ongoing, in-field execution to these models. This process can suggest necessary improvements to the test suite, and new behavior the systems must handle. Classifying the new executions based on temporal safety properties may reveal executions that exploit security vulnerabilities. This information can be used to automatically shut down the system or modify the system execution, forcing it adhere to the observed, modeled behavior.

Behavioral-model-based self-adaptation. SearchRepair [C26] successfully demonstrated that offline defect repair via input-output-behavior search results in high-quality patches. But for such an approach to scale to repairing architecture-level defects and modifying complex, multi-component behavior, it must extend beyond input-output observations. Precise behavioral model inference [C20, C18, C9] can fill this need, enabling a search-based approach to identify components and sets of components whose behavioral models and interactions satisfy the desired requirements of a software system. These component-level requirements can be compiled from partially-specified, system-level requirements [C6] to produce requirement-based search queries. Putting the pieces together, identifying unexpected system behavior via runtime monitoring and compiling the requirements for restoring the expected system-level behavior enables search through a database of automatically-inferred component models to synthesize patches at runtime, leading to behavior-aware, runtime self-adaptation.

These advances will make important strides toward my vision of automating system building and enabling self-adaptive software, improving software quality.
Yuriy Brun’s Research Statement

References


