Allison K. Sullivan – Research Statement

Our society has become increasingly reliant on software systems; however, we are constantly reminded of how hard it is to produce correct software. Just in the last two years British Airways check-in system crashed canceling hundreds of flights, Israel's lunar lander collided into the surface of the moon, and Uber's self-driving car killed a pedestrian. My research directly targets this problem. I develop automated techniques to synthesize and test models of software systems and to provide faster analysis of those models, thus improving software reliability.

Declarative modeling languages improve the quality of software by catching design flaws early in development and verifying imperative implementations. However, the broader community has been slow to adopt these languages as they often have steep learning curves and lack robust development toolsets. A key insight behind my research is that proven imperative verification practices can be mapped to declarative languages, giving users a concrete path to reason over the correctness of their model and facilitating their accessibility to a larger, more diverse community of users. However, just finding bugs is not sufficient. I aim to help developers understand and address errors in their models and to design frameworks that help prevent errors. I strive to develop and evaluate my techniques on real systems of indicative size. To enable technology transfer and reproducible work, I release my tools and benchmarks to the community at large.

My research is well-received by the broader software engineering community. My work is currently funded by the National Science Foundation (No. CCF-1918189). Additionally, I have published at top conferences, including: The International Conference on Software Engineering, The International Conference on Automated Software Engineering, and The Symposium on the Foundations of Software Engineering.

Bringing Imperative Testing Frameworks to the Declarative World.

Software models can only help improve system reliability if they are correct. I design algorithms and tools that bring imperative testing practices into the workflow for declarative languages. My work largely focuses on Alloy, a first-order modeling language based on relational algebra. A strength of Alloy is its SAT-based automated analysis toolset the Analyzer which allows for scenario exploration.

Testing Declarative Languages. Prior to our work, Alloy had no formal notion of "testing." Experienced users would employ a range of ad-hoc, time consuming and error prone techniques, such as enumerating all scenarios and visually inspecting them for issues. Our key insight is that unit testing, the most effective way to validate *code*, provides a blueprint on how to validate *models*. Alloy executes in a declarative environment in which there is no notion of where the "execution" starts, what conditional branches it encounters, and how the values of the final return values are computed. AUnit navigates this situation to define: (1) what is a test case, (2) what test execution and outcomes are and (3) what are coverage criteria [Spin'14, ICST'18]. Thus, AUnit provides systematic method to verify the correctness of their models.

Debugging Declarative Languages. AUnit has served as a foundation for integrating well-established imperative testing techniques into Alloy. To build confidence in test suites, μAlloy introduces mutation testing with first-order AST-level mutant operators. In order to help reveal faulty models, we created two different approaches for automated test generation that have been shown to reveal faults in real-world models: $AGen_{Cov}$, a coverage-directed approach and $AGen_{\mu}$ a mutation-directed approach [ICST'17, ICSE'18]. Our most recent work switches from revealing the bug to locating it. While Alloy offers succinct formulation of complex properties, this is a draw back for localization. Our insight here is that commonly used spectrum-based localization techniques are based on an imperative notion of control flow, which does not translate to Alloy's execution environment. Therefore, our framework, $Alloy_{fl}$, combines spectrum-based with mutation-based fault localization. Experimental results show $Alloy_{fl}$ can find faults spanning multiple locations, and outperforms Alloy's current localization method, the minimum unsat core.

Moving Towards Correct From Construction Systems.

Even with verification techniques in place, writing correct Alloy models is difficult. I design algorithms and tools to synthesize portions of Alloy models. Through synthesis techniques, we establish correct from construction models, which, in turn, can be the basis to create correct from construction systems.

Synthesizing Declarative Languages. Sketching is a form of program synthesis designed to reduce the input burden on the user: the user supplies the high level details by providing a program with holes and a

specification, and the machine creates the low level details by filling in the holes to match the specification. Our key insight is that AUnit tests are a specification. $ASketch$ takes as input a partial model with holes, a generator that provides potential candidate fragments for each hole, and an AUnit test suite to outline the desired behavior. As output, $ASketch$ produces a completed model that passes all tests either by using constraint solving or constraint checking and enumeration [ABZ'18, FSE'18]. We pair our synthesis work with $RexGen$, a systematic approach to generate semantically non-equivalent relational algebra expressions, which allows our synthesis framework to be fully automated [ABZ'18]. Using $ASketch$, we can actually guide the user through the creation of a correct model by searching for the first two logically non-equivalent solutions, generating a counterexample for the user to turn into a test, and resuming the search.

Repairing Declarative Languages. Even though most of Alloy's grammar is finite, sketching suffers from scalability issues due the exponential complexity of sketching multiple expressions, which are not bound to a finite set by the grammar. However, sketching opens the door for automated repair, in which a faulty model is transformed into a corrected model, where "correct" means the model now passes all tests. Our repair framework, $ARepair$, identifies a faulty location using $Alloy_{fl}$, abstracts the formulas related to the location into a sketch, and then uses either an all-combinations or a base-choice methodology to explore candidate solutions. $ARepair$'s base-choice strategy is both efficient and still robust enough to successfully repair real-world faulty Alloy models [ASE'18, ICSE'19].

Ongoing and Future Work.

Improving Scenario Enumeration. When a command successfully executes, the Analyzer will present the user with the scenarios enabled by their model. However, this collection often ranges in the hundreds of thousands and is not presented to the user in any kind of order. Our recent work has focused on giving users more control over which scenarios are generated and how they are presented. Our key insight is to support abstract functions, which give the user a formal mechanism to specify how the scenarios must differ thus producing a smaller, higher quality set of scenarios [ICFEM'19].

Supporting Incremental Development. Model development is inherently incremental, where constraints are added or changed over time due to correcting faulty constraints or updating constraints as the system evolves. In the Analyzer, incremental development is ignored. When a command is re-executed, the entire model is re-translated into the same SAT problem and re-solved even if *nothing* has changed. Alloy, which relies on computationally intensive SAT solvers, can benefit substantially from support for incremental analysis. A novelty of this work is that we are exploring how to use our past work efficiently exploring scenarios with constraint checking to reduce the dependence on SAT solvers and make incremental analysis less computationally intensive.

Testing Unmanned Aerial Vehicles (UAVs). In collaboration with research scientists at the Naval Research Laboratory (NRL), we are working to update SCR, NRL's state-based modeling language, to perform cyber-physical verification of unmanned aerial vehicles. With the shifting focus to autonomous vehicles and their interactions with hardware and the environment, SCR needs richer language constructs to model continuous behavior, such as support for periodically monitored variable. Currently, we are building our approach based on models of Bitcraze's Crazyflie 2.1 behavior, with experiments being run from NRL. In additional to verifying a single UAV, our longer term goal is to verify the interaction between multiple UAVs.

Long Term Goals. In a society where software is becoming more important but not any less buggy, I ambition to build and expand verification and synthesis environments for declarative languages with the hope of encouraging the adoption of these languages to more software systems. I plan to continue to work closely with scientists in both academia and industry by strengthening my current collaborations with researchers from the University of Texas at Austin, the University of Illinois Urbana-Champagne, The University of Nebraska-Lincoln, NRL and Google. In particular, I am actively building my relationship with NRL and have established a link for students to intern at NRL during the summer. NRL offers exposure to industry scale evaluations of research efforts, as it is often asked to apply their work to active development projects. By maintaining and expanding these relationships, I will continue to focus my research on the problems that arise in real world systems, to evaluate the results of my research from a broader viewpoint, and to help apply my research in an industry setting.