

Research Statement¹

Azadeh Farzan

My research fits in the areas of *verification, formal methods, programming languages, and software engineering*. The focus of my research is on the study of programming and reasoning methods which lead to reliable *concurrent* software. Towards this goal, I have developed techniques for *concurrent software analysis* which draw on the *true concurrency* models. With the establishment of the multicore processors and the introduction of language-level thread primitives in languages like Java and C#, and the advent of distributed web services, concurrency has become commonplace even in application software. The design of concurrent software is notoriously error-prone due to the nondeterministic interaction among concurrently executing threads. This makes testing concurrent software extremely hard, and consequently the problem of reliability of concurrent software, a very relevant problem in the research scene of today. My research efforts has been focused on this problem.

Language-independent techniques for software analysis. I developed *JavaFAN* [10,11,12], a tool for formally analyzing (interpretation, LTL model checking, searching for safety violation patterns) *concurrent* Java programs at both source and bytecode levels. JavaFAN is based on a novel approach of using the rewriting logic semantics of programming languages as basis for formal analyses tools. This approach to the development of program analysis tools has several advantages over conventional methods, including generality, relative ease of development, modularity, and potential for formal verification of compilers/translators. JavaFAN demonstrated competitive performance in comparison with other Java model checkers which supports the effectiveness of this framework. These experiments include the discovery of the deadlock bug in NASA's *remote agent* component in fraction of a second time. Completing this project made me an expert in developing rewriting logic semantics (comparable to structural operational semantics) for programming languages and systems. I am currently using this knowledge in two different projects: (1) devising a new language based on the idea of dataflow networks as part of huge project of addressing how we have to rethink concurrency as the result of the new multicore technology, and (2) incorporating formal analyses within the BioNetGen which is a software for development and simulation of rule-based models of biochemical systems, including signal transduction, metabolic, and genetic regulatory networks.

Motivated by the JavaFAN framework, I developed a language-independent *partial order reduction* (POR) unit [4,9] that with a minimally customized interface can be added to a model checker for any language for which the rewriting logic semantics is available. I instantiated this POR unit for Java, Promela, and the Maude language for which significant speedups and space reductions were observed during my experiments. In fact, comparisons with POR unit of the SPIN model checker for programs in Promela showed that my generic POR unit works as well as (and in some cases slightly better than) SPIN's POR unit. I believe there yet remain several interesting research directions to follow in this framework which I plan to follow in future; other state space reduction techniques such as abstraction directly come to mind. Also, it would be very interesting to see this frame work incorporate several orthogonal space reduction techniques in a modular way. My work [8] explores one of these reduction techniques exploiting invisible transitions and *coherence* properties.

Modeling concurrency. The focus of my work is on using the true concurrency models as a basis for software reliability and safety for both static and dynamic analyses [0]. Excellent models of sequential behavior that have evolved during the past thirty years of sequential program verification do not adequately reflect the nature of a concurrent universe. This has spurred interest in true concurrency, namely, in modeling concurrency in a way that is faithful to all currently understood

¹Citations refer to the list of publications as they appear in my CV. Also, I acknowledge the contribution of my coauthors who are all cited when applicable, and for brevity skip mentioning all the names in this text.

modes of interaction of system components, particularly those beyond the reach of sequential models. I introduced the control net model [7], a Petri net-based sound control abstraction for concurrent programs. The *control net* captures the control flow in a concurrent program, and gives a translation from programs to Petri nets that explicitly abstracts data and captures the control flow in the program, as a candidate for standard notion of control flow graph in the concurrent setting. Defining and solving two important and relevant static analysis problems, namely *atomicity* [7] and *dataflow analyses* [5,6], based on the control net model (and its partial order semantics) corroborates the pertinence of causality as the right notion for static analyses of concurrent programs.

Notions of atomicity. Atomicity as a correctness criteria for concurrent programs has been widely studied during the past few years. It expresses a non-interference property for blocks of code that as logical units are supposed to run in the concurrent setting as if no interference from other threads can change the result of their computation. In practice it has been shown that atomicity raises fewer false positives in comparison with race checking which is the traditional way of looking for possible concurrency errors in programs. With the control net as the underlying model, I formulated a new notion of *atomicity* for concurrent programs, called *causal atomicity* [7], based on causality, and an algorithm to statically check whether specified program blocks are causally atomic. Later, in the course of exploring other notions of atomicity, I adapted the traditional notion of *serializability* from database concurrency theory to programming languages and devised an algorithm to dynamically monitor a run of a program for serializability violations [1,3]. This algorithm prompted a very important result which states that model checking a program for serializability is decidable. This result was wrongly concluded in the previous work, which I fixed. At the moment, I am working on building a tool for detecting atomicity related bugs in concurrent programs based on the algorithms I have developed. In future, I would like to investigate weaker notions of atomicity (such as *purity*) in the causal setting that can capture the programming patterns in a more precise way and consequently reduce the number of falsely reported errors in programs which will make this tool more applicable for real programs. These results can be useful in the design of new models of programming (such as software transactional memory) as well as software reliability and program verification.

Static Analysis for concurrency. Static analysis is a natural solution to overcome the high complexity of software verification and analysis. Checking for causal atomicity [7] was the first problem that I investigated in the context of static analysis of concurrent programs. I introduced efficient algorithms to check causal atomicity by enriching the control net with colors and reducing the problem of checking for causal atomicity to a *coverability* query on the corresponding colored Petri net. The coverability queries are checked by the PEP tool, which works based on net unfoldings (the partial order semantics for Petri nets), and hence the checks are performed very efficiently [7].

Furthermore, I formulated the *causal concurrent dataflow* (CCD) framework [5,6] based on *causal flows* in a program which is the first (up to my knowledge) definition of *concurrent* dataflow analyses. This work offers algorithmic solutions to causal flow analyses when the domain of flow facts is a finite set D by exploring the partially-ordered runs of the program as opposed to its interleaved executions. I came up with provably efficient algorithms for the class of *distributive* CCD problems which work fast in early experiments. I believe the causal setting and partial order semantics to be very appropriate concurrent static analyses problems and would like to investigate other static analysis problems from this point of view. As a concrete example, I would like to use the control net and net unfoldings as a basis to develop an abstract interpretation theory for concurrent programs the same way that the control flow graph was used for this purpose in the sequential setting.

Automated Compositional Verification. Compositional verification is an essential technique for addressing the state explosion problem in model checking a concurrent system composed of several components by proving properties of a system by checking properties of its components in an assume-guarantee style. Recently, a paradigm of automated compositional reasoning through

learning (finite state automata) was suggested. This paradigm falls short when it comes to learning *liveness* properties which are very essential in verification of systems. My recent work [2] extends this paradigm by presenting an algorithm to learn an arbitrary ω -regular language (which includes liveness properties). This work addresses the very important problem of learnability of the set of ω -regular languages that has been open for quite a while. With this algorithm at hand, the components and properties can be modeled as Büchi automata and assumptions can be automatically computed as Büchi automata through our learning algorithm. In future, I would like to investigate how *incremental* and *compositional* techniques can be used to scale the static analyses of concurrent software. This can help apply these algorithms to large real world programs.

General Future Research Directions. Besides specific plans already mentioned, I plan on dividing my research efforts for reliability of concurrent software in two fronts: (1) developing language constructs that make concurrent programming safe, secure and reliable, and (2) defining appropriate correctness criteria sculpted to the concurrent setting and developing efficient algorithms to check them. My current work on dataflow network type of languages for concurrent programming and software transactional memory (STM) are examples of the former, which in my point of view, requires people from different disciplines to work together. I believe researchers have the rare opportunity to re-invent some of the cornerstones of computing, provided they simplify the efficient, safe, and reliable programming of highly parallel systems. This can also give the researchers in my field the valuable chance of combining design and verification of systems which could help us tackle problems that are too hard to solve at the moment. Most of the research that I have done in the past fits in the latter category. Continuous Widespread use of concurrency in applications will make (and already has) the security of these software applications a serious issue. Web browsers are very good examples of this context. These security holes go beyond the traditional definitions of bugs (such as races, deadlocks, etc) that have been known for many years. I believe these more sophisticated correctness conditions (such as atomicity) can be defined by combining the knowledge from theory of concurrency with the observation of behavioral patterns of programming that programmers commonly use today.