

Statement of Research Interests¹

Azadeh Farzan

I work in the areas of *software engineering*, *programming languages*, and *verification*. The goal of my research is to investigate programming and reasoning methods which facilitate 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 which makes the problem of reliability of concurrent software a very relevant problem in the research scene of today.

Modeling concurrency. A burning problem in program verification today is how to model the world of concurrent systems. The 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 modes of interaction of system components, particularly those beyond the reach of sequential models. 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].

Static analysis. My work suggests that *causality* is an appropriate notion to capture the flow of control in concurrent programs, and consequently a nice basis for static analyses. Hence, (Mazurkiewicz) Traces and Petri nets, which depict the causal structure of concurrent program in natural way, are proper candidates to model concurrent programs. I introduced the control net model [4], a Petri net-based sound control abstraction for concurrent programs. It 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* [4] and *dataflow analyses* [1], 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.

Dynamic analysis. The true concurrency model of rewriting logic provides an excellent framework for modeling concurrent systems and languages. I have developed a set of language-independent techniques which exploit the advantages of this model in order to build formal analysis tools for concurrent software systems. Moreover, the framework supports a seamless combination of different techniques used for ensuring software reliability including model checking, theorem proving, monitoring, and testing. [3] demonstrates an instance of such combination.

For the software reliability to be commercially realized, I believe that one as a researcher should go beyond the traditional models. One interesting example is the conventional imperative language model that simulates execution as a series of statement execution. This is while a lot of application software is programmed with an *event-driven* mentality (supported by modern languages) and consequently the conventional model fails to detect errors caused by this programming paradigm. Several bugs found in the Internet Explorer browser recently are excellent examples of such situation. Although my past work in language-independent software analysis indirectly addresses this problem, I would like to investigate this problem more at depth and come up with new models that are attuned to the modern programming paradigms.

¹Citations refer to the list of publications as they appear in my CV.

Language-independent techniques for software analysis. I developed JavaFAN [6,7,8], a tool for formally analysis of Java programs (interpretation, LTL model checking, searching for safety violation patters) at both bytecode and source code level, based on definitions of Java language and the Java Virtual Machine (JVM). This work was the first in a series of contributions to introduce the use of language definitions (more specifically rewriting logic semantics in Maude language) as a basis for generic program analysis tools. This method of developing 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 demonstrates competitive performance compared to other (non-generic) Java analysis tools which supports the effectiveness of these ideas. These experiments include the discovery of the deadlock bug in NASA's *remote agent* component in fraction of a second time.

To improve the performance of these generic tools, I next proposed a new approach to build language-independent *partial order reduction* (POR) [5] in the above framework. Getting the POR capabilities does not require making any changes to the underlying model checking algorithms. The generic POR unit can be customized for any language L , under very minimal assumptions, with minimal interface, and with relatively little effort. My experiments with the several languages including Java, JVM, Promela, and Maude indicate that significant state space reductions and time speedups can be gained for tools generated this way. In fact, comparisons with POR unit of the SPIN model checker for programs in Promela show that my generic POR unit works as well as (and in some cases slightly better than) the SPIN's specific POR unit designed for Promela. I plan to work on the advancement of the existing tools by rigorous experimentation on real programs on one hand, and investigating additional optimizations sculpted specifically for this framework such as abstraction techniques on the other hand. Moreover, the very modular nature of this framework suggests enormous potential for the application of several orthogonal optimization techniques that can be combined in the same manner.

Notions of atomicity. Atomicity as a correctness criteria for concurrent programs has been widely studied during the past few years. The notion of atomicity has been inspired by and borrowed from the concept of *serializability* in database systems. I formulated *causal atomicity* as a new notion of atomicity based on causality which is provably a slightly weaker (in the logical sense) notion of atomicity than serializability which in my point of view is more appropriate for programming languages, as it captures the way people write concurrent programs. In fact, one can show that Lipton's transactions are the strongest local property that imply causal atomicity (a global correctness property) which brings all these different notions of atomicity in the literature nicely together. 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. These results can be useful in database concurrency control 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. In fact, one of few commercial successes in the area of software verification is the case of a static analysis suite, which has been incorporated into Microsoft code generation routine and is now used by all programmers there. Checking for causal atomicity [4] 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. In a similar framework, I formulated the *causal concurrent dataflow* (CCD) framework [1] 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.

I would like to investigate how *incremental* and *compositional* techniques can be used to scale the algorithms for detecting the causal atomicity, computing the dataflow solutions and other static analyses that one may define in this framework. This can help apply these algorithms to large real programs. Moreover, it can be a big step towards the ultimate goal of software engineers of combining design and verification of software.

Exploiting concurrency in verification. I plan on dividing my research efforts for reliability of concurrent software on two fronts: (1) developing language constructs that make concurrent programming safe and secure, and (2) defining appropriate correctness criteria sculpted to the concurrent setting and developing efficient algorithms to check them. Software transactional memory (STM) is an example of the former. Researchers across several disciplines in computer science have been intensively working on STM, but most of this work has been focused on performance issues and little work has been done on the issue of reliability and verification of the software written based on STM principles. It has been argued that STM should facilitate the task of ensuring reliability of the software, but little evidence has been provided for the claim. This is an area that I would like to explore more, especially since it may spark collaboration across several fields in a department.

Atomicity as a correctness condition is an example of the latter. 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 can be defined by combining the knowledge from theory of concurrency with the observation of behavioral patterns of programming that programmers commonly use today.