Research Statement
Tayfun Elmas

My research interests are formal methods, algorithms, and tools for the debugging, analysis, and verification of concurrent software. I focus on concurrent implementations of data structures and services that form the backbone of many widely-used systems such as databases, Internet servers, and file systems.

In order to respond efficiently to a large number of clients accessing concurrently, such software makes use of sophisticated synchronization techniques including fine-grained locking and non-blocking operations and creates internal threads, for example, to re-balance a B-link tree. Incorrect use of these techniques makes software prone to concurrency bugs, which can have serious consequences, such as data corruption, operating system crash, or even more catastrophic results, e.g., failure of an aircraft flight control system. Therefore, the functional correctness of software is as fundamental as its performance.

Concurrency-related bugs are difficult to detect, reproduce, and diagnose using code review and testing-based techniques originally developed for sequential programs. This difficulty creates demand for new program analyses that are capable of examining the software with high coverage of its executions and catching subtle bugs. My research goal is to develop methods and software tools to respond to this demand, selecting between runtime and static techniques that best fit the objectives of the work. In addition, I believe that the interaction between the user and a software validation tool is essential for the adoption and effective use of the tool. Thus, allowing developers to conveniently express their design intent and correctness arguments on a program is an important concern. While researching concurrent programs, I have also explored possibilities of transferring ideas from the state of the art in analyzing sequential programs.

In this statement, I first summarize the research I have conducted during my graduate study, then describe my ongoing work, and finally, outline the topics I plan to investigate in the future.

Research During Graduate Study

During my graduate study at Koc University, I led both the conceptual and practical development of three projects: Vyrd, Goldilocks, and QED. These projects were supported by The Scientific and Technological Research Council of Turkey (TUBITAK), The Turkish Academy of Sciences (TUBA), and Microsoft Research.

M.S. Thesis: Runtime Checking of Refinement for Concurrent Software

Standard approaches to testing lack observability of concurrency-related errors for several reasons. First, because it is difficult to determine the right conditions to check throughout a concurrent execution, most checks are applied at the final state of the run. Second, since it is costly to keep track of thread schedules, only the test inputs are considered when analyzing the cause of an error. As a result, many concurrency-related bugs become difficult to diagnose or are not detected.

The outcome of my M.S. thesis is a runtime monitoring and bug detection framework, Vyrd (Elmas et al., PLDI’05), that performs more thorough checking than standard approaches to testing. I applied Vyrd to the distributed B-link tree implementation of the Boxwood storage infrastructure software from Microsoft Research, the Scan file system implemented in Windows NT, and Java class libraries. Vyrd detected subtle concurrency bugs in Boxwood and Scan of which their developers had not been aware even after several months of use, in addition to known or manually inserted bugs in the Java class libraries.
The primary contribution of this work was two novel notions of refinement: I/O and view refinement. These definitions formalize the requirement that a concurrently-accessed data structure implementation conform to an executable specification containing atomic versions of the data structure’s operations. Motivated by our experience with industrial-scale, intricate data structure implementations, refinement provides 1) high observability over executions for effective bug detection, 2) more relaxed definitions than the standard notions of atomicity and linearizability, and 3) conditions that can be checked efficiently at runtime.

While designing the Vyrd tool, we paid special attention to minimizing the concurrency and performance impact on the program being analyzed. This was achieved by using a log-based record/replay mechanism, which enabled Vyrd to perform the checking offline or in parallel with the program. Our experience showed that Vyrd scales well to industrial software, causing low runtime overhead and requiring little manual effort.

We extended Vyrd in several directions and called the resulting tool Vyrd+. First, we re-implemented our refinement checking technique on top of the Java PathFinder (JPF) model checker (Elmas et.al., RV’05). Second, we formulated the location pairs (LP) coverage metric (Tasiran et.al., RV’05), to quantify the adequacy of the testing performed and to direct a model checker towards unexplored executions of the program. Third, we developed and checked in Vyrd+ a new non-interference criterion, rollback-atomicity (Tasiran et.al., RV’07), as a special case of view refinement. Making use of the synergy between runtime verification and model checking, Vyrd+ combines the most essential features of a bug-detection tool: high observability and coverage, automation, and general applicability.

Ph.D. Thesis: Techniques for Runtime Monitoring and Static Verification of Concurrent Software

My Ph.D. thesis consists of two projects, Goldilocks and QED: While Goldilocks aims to detect concurrency bugs at runtime, QED aims to statically verify assertions and linearizability.

Goldilocks: Data-race and Transaction-Aware Java Runtime

In the Goldilocks project, motivated by two facts, we built a Java runtime that continuously monitors the program for data-races (Elmas et.al., PLDI’07). First, data-races, which are the most prevalent symptoms of concurrency bugs, are difficult to detect and reproduce. Second, racy program semantics, which are precisely defined by the Java Memory Model (JMM), are difficult to understand and write correct programs with. On the other hand, the JMM guarantees that when a program is free of races, it will operate under the easy-to-use sequentially-consistent semantics.

When a data-race is about to occur, our runtime throws a Java runtime exception called DataRaceException. This prevents unintended consequences of the data-race and brings the situation to the attention of the programmer. In this way, an actual race condition becomes analogous to other checked errors in Java, for example, null-pointer dereferences or out-of-bound array accesses.

Our PLDI’07 paper was the first to propose DataRaceException and show that, in addition to being a debugging tool during software development, DataRaceException is a viable mechanism for the enforcement of program safety after deployment. For this, our paper has been selected for publication in the SIGPLAN CACM Research Highlights (Nominated in January 2009).

Our runtime provides three key benefits. First, it helps to identify concurrency-related bugs that manifest themselves as data-races. By interrupting the execution when a data-race is detected, our runtime allows the programmer to analyze the bug before it causes an error that may be impossible to diagnose later. Second, if no DataRaceException is thrown during an execution, an error in that execution can be analyzed by assuming sequential-consistency and ruling out many possibilities of missing synchronization. Third, by catching the DataRaceException, the programmer can terminate gracefully or roll-back the effects of the operation that triggered the race. Alternatively, he can use DataRaceException as a conflict-detection mechanism in optimistic uses of concurrency, e.g., software transactional memory (STM).
In order to detect data-races at runtime, we developed Goldilocks, a lockset-based algorithm for computing the happens-before relation as defined in the JMM. Goldilocks is the first algorithm to have the three essential features necessary for the implementation of DataRaceException: First, it is sound, i.e., it detects all actual data-races. Second, it is precise, i.e., it never gives false decisions. Third, it is efficient, i.e., it has tolerable runtime overhead, which is competitive with that of the vector-clock and Eraser algorithms. Furthermore, Goldilocks provides the first formalization of race conditions for programs using software transactional memory along with other synchronization primitives in Java.

I implemented Goldilocks inside the Kaffe JVM, a full-fledged Java virtual machine written in C. The algorithm was also implemented in the Chess model checker of Microsoft Research and the RoadRunner infrastructure and taught in concurrency-related courses at several universities.

**QED: Static Proof System based on Atomicity**

The goal of the QED project is to simplify the static verification of the (partial) correctness of concurrent programs. The interaction between threads over shared memory complicates static verification: while proving a program, one must consider all possible interleavings of conflicting operations. A typical proof of assertions requires the user to write a code annotation, which must be stable under interference from other threads, for every interleaving point. A typical proof of linearizability requires, along with these annotations, the user to write an abstraction map that relates every state of a concurrent data structure to a state of a sequential specification. Under fine-grained concurrency, meeting these requirements becomes a challenge.

The primary contribution of this work is a static proof system, called QED, for proving assertions (Elmas et.al., POPL'09) and linearizability (Elmas et.al., TACAS'10). QED provides a novel proof strategy in which *atomicity is used as a proof tool*: a program with fine-grained concurrency is transformed iteratively to make it consist of larger atomic actions, and the correctness conditions are checked after the program reaches an atomicity level appropriate for local analyses. This gradually reduces the influence of thread interleavings on the complexity of the reasoning, and therefore permits significantly more tractable proofs than those provided by existing methods. For example, we prove assertions by performing sequential (local) checks within atomic blocks of the final program and prove linearizability by computing the abstraction map iteratively during the proof.

Proofs in QED replace complex annotations and abstraction maps with code transformations, which are based on novel applications of *reduction* and *abstraction* techniques. In contrast to the standard uses of these techniques, reduction and abstraction in QED are *symbiotic*, which allows us to reach the desired level of atomicity even when there is apparent interference between threads. Moreover, the transformations provide strong guarantees so that one can simplify a program with larger atomic blocks using QED and continue the proof with another method, such as separation logic.

I implemented the proof system in a software tool, also called QED, which is open source and can be downloaded from [http://qed.codeplex.com](http://qed.codeplex.com). The user interacts with the tool via proof commands, each directing a code transformation at the programming language level. QED checks the preconditions of the proof commands using the Z3 theorem prover, and then if applicable, performs the required transformations automatically. Since the proof script for a program serves as a *reproducible* documentation of its correctness, QED provides a proof repository for a collection of concurrent software.

We evaluated our tool by proving assertions in and the linearizability of a wide-range of programs from the literature. Our experience has produced strong evidence for two claims. First, QED is *effective*: the QED’s notions of reduction and abstraction are flexible enough to construct proofs of a wide spectrum of programs using fine-grained locking, non-blocking operations, and optimistic concurrency. Second, QED is *usable*: its proof strategy captures the design intent of the programmer by separating concerns about concurrency and data-flow during the proof, and thus helps him to express and check each correctness argument on a program in a clear way.
Ongoing Work and Short-Term Research Agenda

My ongoing research builds upon the work presented in my Ph.D. thesis and aims to: 1) improve the effectiveness and usability of the QED proof system and 2) apply the ideas and concepts in QED to other research topics in concurrency.

Improving the effectiveness and usability of QED

In order to improve the effectiveness of QED, I am focusing on increasing the scalability and efficiency of the mover (commutativity) check, a fundamental operation for reduction. QED performs the check by generating a verification condition (VC) and calling a theorem prover to decide the validity of the VC for every pair of atomic actions in the program. Consequently, the memory and time cost of performing a mover check becomes considerably higher while verifying large programs. I am investigating how to reduce this cost by 1) exploiting the modular design of the program and 2) making use of low-cost commutativity checks.

It is common that two modules either use distinct sets of global variables or share a small set of global variables following a simple access pattern. Because of this, I am formulating a modular check to decide the mover type of an atomic action within the containing module. This will allow QED to apply reduction to every module separately and to re-use the proof of a module while the rest of the program changes.

In addition, type and effect systems, such as those in Atomizer and Deterministic Parallel Java (DPJ), provide lightweight algorithms to check sufficient conditions for commutativity, including basic notions of movers and data-race freedom. I am extending reduction in QED to use these systems as short-circuit mover checks to show atomicity in straightforward cases so that the theorem prover is called only when there is a complex argument on the mover type of an atomic action that cannot be expressed by a type and effect system.

I am developing proof idioms to improve the usability of QED. A QED proof consists of scenarios in which the user indicates a fact about the program, for example, that a variable is always lock-protected, or reduces a compound statement to a single action. Such scenarios may require the application of multiple, low-level proof rules that are difficult for programmers to use. I observed that the proof rules applied for every instance of a scenario follow a common pattern and started using proof idioms to document these patterns and mechanize their application. An idiom is defined through a proof script template containing the low-level rules required for the scenario. The user completes the template by providing concise and application-specific hints and runs the completed script in the QED tool automatically. We presented the first set of proof idioms for common synchronization mechanisms, such as mutex and reader/writer locks in (Elmas et.al., PADTAD’09). I am working on documenting the common proof scenarios I encountered while proving various programs. My goal is to keep the user’s interaction during the proof as minimal and clear as possible by, for example, constructing a proof only in terms of proof templates.

Applying the ideas and concepts in QED to other research topics in concurrency

While building the QED proof system we developed novel, more semantic, and therefore more widely applicable notions of reduction and abstraction. I foresee that these techniques will improve the work in related areas on concurrent software. For instance, the existing approaches to proving determinism and parallelizing sequential programs are based on showing that different threads access distinct memory locations. These approaches become ineffective when parallel tasks interfere with each other on the shared memory and the program uses synchronization primitives. In these cases, a commutativity analysis can further the determinism proof and the code parallelization by showing that executions of conflicting code blocks in different orders may all result in semantically equivalent states. The mover check in QED is well-suited to perform such analyses precisely and automatically using a theorem prover. I am investigating how the mover check can be used as a general purpose reasoning tool to complement existing commutativity
analyses. This also includes identifying the fragments of logics in which the mover check remains decidable and developing algorithms to generate VCs that can be solved efficiently by theorem provers.

LONG-TERM RESEARCH GOALS

Because of emerging technologies such as multicore processors and grid computing, concurrency is becoming an important issue for today’s software systems and is likely to become even more so in the future. My long-term research goal is to provide software developers with bug detection and verification tools they can use conveniently to build concurrent software that is both efficient and reliable.

Programming platforms (languages, runtimes, and disciplines) have considerable influence on the effectiveness and successful use of bug detection and verification tools. For example, the C language allows lots of features and flexibility for low-level programming, which makes it rather complicated to reason about the functionality of C programs. This has led researchers to develop restricted versions of C, e.g., Cyclone, that provide strong safety guarantees, such as no dangling pointers, and are therefore easier to analyze. Additionally, the TBB, Cilk, and Open MP platforms offer carefully selected programming idioms, such as structured fork/join parallelism, that implicitly restrict the use of concurrency and prevent the mistakes that can occur in a more flexible platform. In addition, they provide easy-to-use, dynamic data-race checkers, which allow programmers to identify concurrency bugs during development. I want to investigate how similar programming environments can lead to debugging and verification tools that are highly tuned to a particular platform, and thus more effective than general-purpose tools in catching bugs and verifying programs running on that platform.

I believe that programmer’s help is necessary for software validation tools to support complex and large programs. Programming platforms for sequential programs have been successful in providing developers with a carefully designed set of programming constructs and code annotations to add specifications about the program. For example, Spec# allows the programmer to give loop invariants and procedure contracts as annotations and define variables of non-null types, which enables precise and modular (thus efficient) verification of the program. In my research, I want to investigate a natural way of allowing programmers to provide similar annotations for concurrent programs. In QED, the user’s help through program annotations was essential to support, for example, recursion (Elmas et.al, POPL’09) and synchronization mechanisms (Elmas et.al., PADTAD’09). By having a convenient interface with the verification tool, the programmer can express his reasoning about the functionality of the software by, for example, indicating thread local variables and an execution ordering between parallel statements, and letting the verification tool capture this reasoning to apply a more tractable (e.g., local) analysis than would be possible in a standard approach.

When verifying complex systems, static and runtime techniques can be used together in order to address each other’s limitations. I want to develop new methods in which reasoning about the correctness of a program is divided into a set of proof obligations, each separately checked by a static or runtime technique. For example, in the Goldilocks project, we used static analysis tools to eliminate checks for variables that these tools have proved to be race-free and reduced the cost of runtime checking dramatically. Theorem prover-based deductive techniques and model checking-based state-space exploration techniques can be combined for similar purposes. For instance, a strict runtime execution ordering between the program statements can eliminate many commutativity checks in reduction. While such an ordering is difficult to express and check using a theorem prover, a model checker can easily verify the ordering. I want to attack the following challenges in developing such hybrid methods: 1) how to divide the whole verification task into pieces manageable by each technique, and 2) how to ensure the soundness of the combined analysis, i.e., the static and runtime (or the deductive and algorithmic) checks together imply the correctness of the software.