Design and Validation of Concurrent Systems
— Dagstuhl Seminar —

Cormac Flanagan[1], Susanne Graf[2], Madhusudan Parthasarathy[3] and Shaz Qadeer[4]

[1] University of California at Santa Cruz, USA
[2] VERIMAG - Gières, France
[3] University of Illinois at Urbana-Champaign, USA
[4] Microsoft Research, Redmond, USA

Abstract. The Dagstuhl Seminar 09361 "Design and Validation of Concurrent Systems" was held in Schloss Dagstuhl – Leibniz Center for Informatics from August 30 to September 4, 2009. During the seminar, participants presented their current research, and ongoing work and open problems were discussed. This paper reviews the seminar topics and goals, and provides abstracts of the presentations given during the seminar. Links to extended abstracts or full papers are provided, if available.

Keywords. Concurrency, Specification, Programming, Verification, Validation, Testing

1 09361 Summary – Design and Validation of Concurrent Systems

While concurrency has always been central to embedded and distributed computing, it has recently received increasing interest from other fields related to software engineering such as programming languages, compilers, testing, and verification. This recent interest has been fuelled by a disruptive trend in the evolution of microprocessors — the number of independent computing cores will continue to increase with no significant increase in the speed of each individual core. This trend implies that software must become increasingly concurrent in order to exploit current and future hardware.

At the same time, we are seeing an unprecedented penetration of embedded and distributed systems into everyday human life. Embedded devices, such as cell phones and media players, are ubiquitously used for communication and entertainment, and distributed control systems in cars and airplanes are increasingly safety-critical. Today, systems and software engineers face the challenging task of developing efficient and reliable software for concurrent, embedded, distributed, and multi-core platforms.

The presence of concurrency in a system severely increases its complexity due to the possibility of unexpected interactions among concurrently-executing components. System designers are invariably forced to make trade-offs between

productivity, correctness, and performance. Current practice includes "correct-by-construction" design methods that yield safe implementations; these implementations are unlikely to be the most efficient. Conversely, highly flexible design methods can yield efficient distributed or multithreaded implementations; these methods are labor intensive and may require expensive post-design validation. We believe these two approaches delimit a continuous spectrum of design and validation techniques. It is important to develop techniques that provide a principled but pragmatic tradeoff between the rigidity of "correctness-by-construction" and the difficultly of post-hoc verification of arbitrary systems.

This workshop brought together academic and industrial researchers who are interested in design and validation techniques for concurrent systems. We had a broad participation reflecting the various approaches to the problem, including language design, compiler construction, program analysis, formal methods, and testing. We believe this mix of participants generated interesting and lively discussions. Concretely, we addressed the following set of inter-related questions during the seminar:

- Specification and programming languages: How can a programmer specify correctness properties of a concurrent system? What are the right idioms for reasoning about concurrent programs? What concurrency-control mechanisms should be provided by the programming language? How do we enable programmers to write well-reasoned code that can be compiled for efficient execution on a multi-core platform? What kind of abstractions from the hardware/OS/runtime are useful and efficient?
- Design methods: How should a programmer choose the right design approach given the constraints, such as quality-of-service and reliability, that may be imposed on a given application domain? What are common patterns of non-interference, e.g. race-freedom, atomicity, and determinism, that help programmers avoid common concurrency-related pitfalls?
- Validation: How do we verify applications built using a given set of concurrency primitives? How do we verify implementations of algorithms realizing these primitives? How do we design efficient algorithms for verifying various forms of non-interference, and for explaining existing interference in terms understandable to the programmer? How do we test concurrent applications that may exhibit a high degree of, possibly uncontrollable, nondeterminism?

The following section contains abstracts of many of the presentations at the workshop.

2 Abstracts

Context-bounded analysis for concurrent programs with dynamic creation of threads

Mohamed-Faouzi Atig (LIAFA University Paris VII, FR)

Context-bounded analysis has been shown to be both efficient and effective at finding bugs in concurrent programs. According to its original definition, context-bounded analysis explores all behaviors of a concurrent program up to some fixed number of context switches between threads. This definition is inadequate for programs that create threads dynamically because bounding the number of context switches in a computation also bounds the number of threads involved in the computation. In this paper, we propose a more general definition of context-bounded analysis useful for programs with dynamic thread creation. The idea is to bound the number of context switches for each thread instead of bounding the number of switches of all threads. We consider several variants based on this new definition, and we establish decidability and complexity results for the analysis induced by them.

Keywords: Verification, Concurent programs

Joint work of: Atig, Mohamed-Faouzi; Bouajjani, Ahmed; Qadeer, Shaz

Deterministic Parallel Java: Towards "Deterministic by Default" Parallel Programming in an Object-Oriented Language

Robert L. Bocchino Jr. (University of Illinois - Urbana, US)

We say that a parallel program is deterministic if it produces the same output on every execution with a given input, regardless of the parallel schedule chosen. Determinism makes parallel programs much easier to write, understand, debug, and maintain. Further, many (though not all) parallel programs are intended to be deterministic. However, general-purpose languages particularly those that allow arbitrary updates to shared data, pointers, and aliasing typically do not guarantee determinism, leaving the burden on the programmer to ensure that the program is correctly synchronized.

In this talk, I will present work that I have done with Vikram Adve and others at the University of Illinois on Deterministic Parallel Java (DPJ). DPJ uses a sophisticated type and effect system to guarantee deterministic by default semantics for an object-oriented parallel language: that is, the program is guaranteed to be deterministic unless the programmer explicitly requests a controlled form of nondeterminism. I will give an overview of the typing mechanisms that DPJ uses to guarantee determinism, and I will illustrate some parallel patterns that DPJ can express with good performance. I will also discuss some limitations of DPJ, and directions for future work.

Keywords: Shared-memory concurrency, parallel programming, languages, type and effect systems, determinism

Joint work of: Robert L. Bocchino Jr.; Vikram Adve

A Calculus of Atomic Actions

Tayfun Elmas (Koc University - Istanbul, TR)

Concurrency-related bugs are notoriously difficult to discover by code review and testing. By doing a formal proof on the program text, one can statically verify that no execution of the program leads to an error. The effectiveness of the proof depends on the proper choice of the manual inputs such as code annotations. Deriving these annotations for a concurrent program requires complicated reasoning. The main reason behind this is the interaction between threads over the shared memory. While writing the proof, at every point, one has to consider the possible interleavings of conflicting operations. Existing proof methods including Owicki-Gries, rely/guarantee and concurrent separation logic are applied at the finest level of concurrency. Analyzing the program at this level requires complex annotations and invariants, along with many auxiliary variables.

In this talk, we present a new proof method that simplifies verifying assertions in concurrent programs. The key feature of our method is the use of atomicity as the main proof tool. A proof is done by rewriting the program with larger atomic blocks in a number of steps. In order to reach the desired level of atomicity, we alternate proof steps that apply abstraction and reduction, each of which improves the outcome of the other in a following step. Then, we check assertions sequentially within the atomic blocks of the resulting program. We declare the original program correct when we discharge all the assertions. Our proof style provides separation of concerns: At each step, we either use facts about a concurrency protocol to enlarge atomic blocks, or check data properties sequentially. Our software tool, QED, mechanizes proofs using the Z3 SMT solver to check preconditions of the proof steps. We demonstrated the simplicity of our proof strategy by verifying well-known programs using fine-grained locking and non-blocking data algorithms.

Keywords: Concurrent Programs, Static Verification, Atomicity, Reduction, Abstraction, Proof Idiom

Joint work of: Tayfun Elmas; Shaz Qadeer; Ali Sezgin; Serdar Tasiran

Full Paper:
https://research.microsoft.com/pubs/70608/popl09.pdf

See also: Elmas, T., Qadeer, S., and Tasiran, S. 2009. A calculus of atomic actions. In Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Savannah, GA, USA, January 21 - 23, 2009). POPL '09. ACM, New York, NY, 2-15.

Towards Effective Testing of Concurrent Programs

Azadeh Farzan (University of Toronto, CA)

We propose non-atomic runs as good candidates in concurrent programs to manifest concurrency-related bugs. In this talk, we will give an overview of dynamic atomicity checking for concurrent programs, including checking a single run for atomicity violations. We will then talk about how to infer non-atomic runs from any observed run for the effective testing of the concurrent program using those runs.

Keywords: Atomicity, testing, concurrency, model checking, nested locking

Full Paper:
http://www.springerlink.com/content/267n543l03u05026/

Dynamic Checking of Multithreaded Software

Cormac Flanagan (University of California - Santa Cruz, US)

Multithreaded programs are prone to scheduler-dependent bugs that are notoriously difficult to detect, reproduce, or eliminate. This talk overviews our recent work on dynamic analysis techniques to detect race conditions and violations of desired atomicity or determinism properties.

Keywords: Races, atomicity, determinism, concurrency

Joint work of: Cormac Flanagan; Stephen N. Freund; Caitlin Sadowski; Jaeheon Yi

Types For Concurrency

Stephen N. Freund (Williams College - Williamstown, US)

Concurrency errors are particularly difficult defects to identify and fix in software systems. This talk gives an overview of type-based analyses for specifying and verifying that such problems do not exist. The talk covers the basic foundations of type systems for preventing race conditions and atomicity violations, the limitations and strengths of types for concurrency, and our experiences in applying type-based tools to analyze software systems.

Joint work of: Cormac Flanagan; Stephen N. Freund; Shaz Qadeer

Underapproximation of Concurrent Systems

Pierre Ganty (IMDEA - Software, ES)

Many problems in the verification of concurrent software systems reduce to checking the non-emptiness of the intersection of two context-free languages, an undecidable problem.

We propose a decidable under-approximation, and a semi-algorithm based on the under-approximation, for this problem through bounded languages, which are context-free subsets of a regular language of the form $w_1^* w_2^* \dots w_k^*$ for some $w_1, \dots, w_k \in \Sigma^*$. Bounded languages have nice structural properties, in particular the non-emptiness of the intersection of a bounded language and a context free language is decidable.

Our main theoretical result is a constructive proof of the following result: for any context free language L, there is a bounded language $L' \subseteq L$ which has the same Parikh image as L. Along the way, we show an iterative construction that associates with each context free language a family of linear languages and linear substitutions that preserve the Parikh image of the context free language. We show an application of this result to underapproximate the reachability problem of multi-threaded procedural programs.

Keywords: Underapproximations, concurrent systems, language theory, algebraic techniques

Joint work of: Pierre Ganty; Rupak Majumdar; Benjamin Monmege

Message Passing: Formalization and Dynamic Verification

Ganesh Gopalakrishnan (University of Utah, US)

Consider a message passing library consisting of five action types: S(to:x), R(from:y), R(from:*), W(handle), and B, signifying a non-blocking send, non-blocking receive, non-blocking wildcard receive, wait (an S or an R), and barrier. We associate (upto) four states of progress with each of these action invocations: invoked, call returned, matched, and completed. We show that these events help define a state transition semantics relying upon a happens-before relation designed to support efficient message passing. We show that this formalization is the 'heart' of MPI, the lingua franca for high performance message passing based programming used in all sorts of high performance applications (e.g. earthquake modeling).

We conjecture that any attempt to achieve message passing efficiently in practice, and in a resource aware manner, will potentially lead to something like these four primitives. Our formalism allows one to mechanically calculate the state transition behavior of what used to be dismissed as "hairy MPI code." This deep understanding of MPI has been "baked into" our real-world deployable dynamic verification tool for MPI that comes with an elegant Eclipse-based front-end and is being downloaded worldwide. This is joint work with the Gauss Group, notably Vakkalanka, Vo, DeLisi, Humphrey, Derrick, Aananthakrishnan, and Kirby.

Keywords: Message Passing, Dynamic Verification, MPI, High Performance Computing

Full Paper:
http://www.cs.utah.edu/formal_verification/ISP-Eclipse

See also: FM 2009, Eindhoven, Nov 2-6, 2009 (accepted; covers part of this presentation)

Proving that non-blocking algorithms don't block

Alexey Gotsman (University of Cambridge, GB)

A concurrent data-structure implementation is considered non-blocking if it meets one of three following liveness criteria: wait-freedom, lock-freedom, or obstruction-freedom. Developers of non-blocking algorithms aim to meet these criteria. However, to date their proofs for non-trivial algorithms have been only manual pencil-and-paper semi-formal proofs. This paper proposes the first fully automatic tool that allows developers to ensure that their algorithms are indeed non-blocking. Our tool uses rely-guarantee reasoning while overcoming the technical challenge of sound reasoning in the presence of interdependent liveness properties.

Keywords: Non-blocking concurrency, shape analysis, program logics, liveness

Dynamic Detection of Atomic-Set-Serializability Violations

Christian Hammer (Purdue University, US)

Previously we presented atomic sets, memory locations that share some consistency property, and units of work, code fragments that preserve consistency of atomic sets on which they are declared. We also proposed atomic-set serializability as a correctness criterion for concurrent programs, stating that units of work must be serializable for each atomic set. We showed that a set of problematic data access patterns characterize executions that are not atomic-set serializable. Our criterion subsumes data races (single-location atomic sets) and serializability (all locations in one set). In this paper, we present a dynamic analysis for detecting violations of atomic-set serializability. The analysis can be implemented efficiently, and does not depend on any specific synchronization mechanism. We implemented the analysis and evaluated it on a suite of real programs and benchmarks. We found a number of known errors as well as several problems not previously reported.

Keywords: Concurrent Object-Oriented Programming, Data Races, Atomicity, Serializability, Dynamic Analysis

Joint work of: Christian Hammer; Julian Dolby; Mandana Vaziri; Frank Tip

Full Paper:
http://pp.info.uni-karlsruhe.de/uploads/publikationen/hammer08icse.pdf

See also: Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, Dynamic Detection of Atomic-Set-Serializability Violations, Proceedings of the 30th International Conference on Software Engineering, ACM, May 2008.

Concurrency Semantics for Transactions

Maurice Herlihy (Brown University - Providence, US)

Traditional transactional memory systems suffer from overly conservative conflict detection, yielding so-called false conflicts, because they are based on fine-grained, low-level read/write conflicts. In response, the recent trend has been toward integrating various abstract data-type libraries using ad-hoc methods of high-level conflict detection. These proposals have led to improved performance but a lack of a unified theory has led to confusion in the literature.

We clarify these recent proposals by defining a generalization of transactional memory in which a transaction consists of abstract data-type operations rather than simply memory read/write operations. We provide semantics for both pessimistic (e.g. transactional boosting) and optimistic (e.g. traditional TMs and recent alternatives) execution. We show that both are included in the standard atomic semantics, yet find that the choice imposes different requirements on the coarse-grained operations: pessimistic requires operations be left-movers, optimistic requires right-movers. Finally, we discuss how the semantics applies to numerous TM implementation details discussed widely in the literature.

Joint work of: Eric Koskinen; Matthew Parkinson; Maurice Herlihy

Symbolic Concurrent Semantics of Safe Petri Nets with Application to the Unfolding of Time Petri Nets

Claude Jard (IRISA - Rennes, FR)

Concurrent semantics aims at representing behaviors of concurrent systems with an explicit description of causal dependencies. This avoids to memorize all the inlerleavings of actions as often required when using a sequential semantics. Starting from ordinary safe Petri nets, we will show a way to deal with colored Petri nets on infinite domains. The resulting semantics is given in term of a symbolic event structure, mixing a graphical representation of causal dependencies with symbolic constraints associated with each event. The considered colored model is sufficiently powerful to code safe time Petri nets, for which it exists a finite complete prefix of the unfolding.

Keywords: Petri nets, unfolding, event structures, concurrent semantics, colored petri nets, time Petri nets.

Joint work of: Claude Jard; Thomas Chatain

Reasoning about Threads Interacting via Locks

Vineet Kahlon (NEC Laboratories America, Inc. - Princeton, US)

Precise dataflow analysis of concurrent programs relies on accurately capturing interference between threads.

Thread interference is governed by scheduling constraints imposed by synchronization primitives like locks, rendezvous, barriers, etc., as well as shared variables. Given a synchronization primitive p, gauging its effect on thread interference can be reduced to a model checking problem for threads interacting purely using p. In this talk, we consider the problem of model checking threads interacting via locks. We show the problem is decidable for threads interacting via bounded lock chains for certain fragments of Linear Temporal Logic (LTL). This enables dataflow analysis to be carried out for a larger class of programs than those allowed by the current state-of-the-art which can handle only nested locks. Time permitting, we also discuss the problem of model checking threads interacting via rendezvous (Wait/Notify style primitives).

Keywords: Dataflow Analysis, Model Checking, Locks, Wait/Notify

Interprocedural Analysis of Concurrent Programs under a Context Bound

Akash Lal (University of Wisconsin - Madison, US)

We address interprocedural analysis of concurrent programs with shared memory. Such an analysis is undecidable in the presence of multiple procedures. One approach used in recent work obtains decidability by providing only a partial guarantee of correctness: the approach bounds the number of context switches allowed in the concurrent program, and aims to prove safety, or find bugs, under the given bound. In this talk, we show how to obtain simple and efficient algorithms for the analysis of concurrent programs with a context bound. We give a general reduction from a concurrent program P, and a given context bound K, to a sequential program P_s^K such that the analysis of P_s^K can be used to prove properties about P. We also give a reduction for weighted pushdown systems that proves decidability of context-bounded analysis in the presence of complex abstractions, extending previously known results.

Keywords: Verification, Concurrency, Context-Bounded Analysis, Boolean programs, Weighted pushdown systems

Joint work of: Akash Lal; Thomas Reps; Tayssir Touili; Nicholas Kidd

Full Paper:
http://www.cs.wisc.edu/wpis/abstracts/fmsd09.abs.html

Formal and Executable Contracts for Transaction-Level Modeling in SystemC

Florence Maraninchi (VERIMAG - Univ. of Grenoble, FR)

Transaction-Level Modeling (TLM) for systems-on-a-chip (SoCs) has become a standard in the industry, using SystemC.

With SystemC/TLM, it is possible to develop an executable virtual prototype of a hardware platform, so that software developers can start writing code long before the actual chip is available. A hardware model in System- C/TLM can be very abstract, compared to the detailed RTL model. It is clearly component-based, with guidelines defining how components should be designed for use in any TLM context. However, these guidelines are quite informal for the moment. In this paper, we establish a structural correspondence between functional SystemC/TLM models and a formal component-model for embedded systems called 42, for which we have defined a notion of control contract, and an execution mode for systems made of components and contracts. This is a way of formalizing the principles of functional SystemC/TLM. Moreover, it allows the combined use of SystemC/TLM components with 42 components. Demonstrating that such a combined use is possible is key to the adoption of formal components and definitions in the community of TLM users.

Keywords: TLM, SoC, components, contracts, concurrent languages and MoCCs

Joint work of: Florence Maraninchi; Tayeb Bouhadiba

Full Paper:
http://www-verimag.imag.fr/~maraninx/spip.php?article168

Local testing of message-passing systems

Madhavan Mukund (Chennai Math. Institute - Madras, IN)

The only practical way to test distributed message-passing systems is to use local testing. In this approach, used in formalisms such as concurrent TTCN-3, some components are replaced by test processes. Local testing consists of monitoring the interactions between these test processes and the rest of the system and comparing these observations with the specification, typically described in terms of message sequence charts.

The main difficulty with this approach is that local observations can combine in unexpected ways to define implied scenarios not present in the original specification.

We first show that checking for implied scenarios is undecidable for regular specifications, even if observations are made for all but one process at a time.

To get around this, we append tags to the messages generated by the system under test. Our tags are generated in a uniform manner, without referring to or influencing the internal details of the underlying system. These enriched behaviors are then compared against a tagged version of the specification. The main result is that detecting implied scenarios becomes decidable in the presence of tagging.

Keywords: Message-passing systems, formal testing

Joint work of: Puneet Bhateja; Paul Gastin; K. Narayan Kumar

Priority scheduling based on knowledge and model checking

Doron A. Peled (Bar-Ilan University - Ramat-Gan, IL)

Priorities are used to control the execution of systems to meet given requirements for optimal use of resources, e.g., by using scheduling policies. For distributed systems, it is hard to find efficient implementations for priorities; because they express constraints on global states, their implementation may incur considerable overhead.

Our method is based on performing model checking for knowledge properties. It allows identifying where the local information of a process is sufficient to schedule the execution of a high priority transition. As a result of the model checking, the program is transformed to react upon the knowledge it has at each point. The transformed version has no priorities, and uses the gathered information and its knowledge to limit the enabledness of transitions so that it matches or approximates the original specification of priorities.

Keywords: Model checking, Priorities, Knowledge

Joint work of: Saddek Bensalem; Doron A. Peled; Joseph Sifakis

On Speculative Computation and Thread Safe Programming

Gustavo Petri (INRIA Sophia Antipolis - Méditerranée, FR)

We propose a formal definition for (valid) speculative computations, which is independent of any implementation technique. By speculative computations we mean optimization mechanisms that rely on relaxing the flow of execution in a given program, and on guessing the values read from pointers in the memory. Our framework for formalizing these computations is the standard operational one that is used to describe the semantics of programming languages. In particular, we introduce speculation contexts, that generalize classical evaluation contexts, and allow us to deal with out of order (or parallel) computations. We show that the standard DRF guarantee, asserting that data race free programs are correctly implemented in a relaxed semantics, fails with speculative computations, but that a similar guarantee holds for programs that are free of data races in the speculative semantics. We then introduce a language featuring an explicit distinction between shared and private variables, and show that there is a translation, guided by a type and effect system, that transforms a program written in this language into speculatively data race free code, which is therefore robust against aggressive optimizations.

Keywords: Shared-memory concurrency, design of concurrent programming languages, relaxed memory models

Joint work of: Gerard Boudol; Gustavo Petri

Logical Concurrency Control From Sequential Proofs

Ganesan Ramalingam (Microsoft Research India - Bangalore, IN)

In this talk, we consider the problem of making a sequential library safe for concurrent clients. Informally, given a sequential library that works satisfactorily when invoked by a sequential client, we wish to synthesize concurrency control code for the library that ensures that it will work satisfactorily even when invoked by a concurrent client (which may lead to overlapping executions of the library's procedures).

Formally, we consider a sequential library annotated with assertions along with a proof that these assertions hold in a sequential execution.

We show how such a proof can be used to derive a concurrency control for the library that guarantees that the library's execution will satisfy the same assertions even when invoked by a concurrent client.

We extend the approach to guarantee linearizability: any concurrent execution of a procedure is not only guaranteed to satisfy its specification, it also appears to take effect instantaneously at some point during its execution.

Keywords: Synthesis, concurrency control, isolation, atomicity, shared memory

Joint work of: Jyotirmoy Deshmukh; Ganesan Ramalingam; Venkatesh-Prasad Ranganath; Kapil Vaswani

Resilient Verification for Optimistic Concurrent Algorithms

Noam Rinetzky (University of London, GB)

We present a novel approach to the verification of highly concurrent optimistic algorithms. Our key observation is that, rather than making reasoning harder, the extreme high degree of concurrency in these algorithms actually makes it possible to use extremely simple reasoning techniques which reduce the verification effort. Based on this observation, we propose to verify these algorithms using a restricted form of assertion, resilient assertions, which keeps no correlations between the local state of a thread, the local states of all other threads, and the shared mutable state. Sequential program proofs which use resilient assertions are interference-free by construction, making the concurrent verification process compositional and proof checking of the same difficulty as sequential verification.

We demonstrate the value of resilient verification by presenting a proof of memory safety, structural (shape) invariants, and linearizability for a state of the art highly-concurrent optimistic set algorithm. Our approach for showing linearizability is of particular interest as it suggests a new non-constructive proof technique, formalized by a hindsight lemma, which sidesteps the need to find linearization points.

Keywords: Verification, concurrency, linearizability, resilient, hindsight

Joint work of: Noam Rinetzky; Peter O'Hearn; Martin Vechev; Eran Yahav; Greta Yorsh

Asserting and Checking Determinism for Multithreaded Programs

Koushik Sen (University of California - Berkeley, US)

The trend towards processors with more and more parallel cores is increasing the need for software that can take advantage of parallelism. The most widespread method for writing parallel software is to use explicit threads. Writing correct multithreaded programs, however, has proven to be quite challenging in practice. The key difficulty is *nondeterminism*. The threads of a parallel application may be interleaved nondeterministically during execution.

In a buggy program, nondeterministic scheduling will lead to nondeterministic results—some interleavings will produce the correct result while others will not.

We propose an assertion framework for specifying that regions of a parallel program behave deterministically despite nondeterministic thread interleaving. Our framework allows programmers to write assertions involving pairs of program states arising from different parallel schedules. We describe an implementation of our deterministic assertions as a library for Java, and evaluate the utility of our specifications on a number of parallel Java benchmarks. We found specifying deterministic behavior to be quite simple using our assertions. Further, in experiments with our assertions, we were able to identify two races as true parallelism errors that lead to incorrect nondeterministic behavior. These races were distinguished from a number of benign races in the benchmarks.

Full Paper:
http://srl.cs.berkeley.edu/~ksen/papers/detcheck.pdf

See also: Jacob Burnim and Koushik Sen, Asserting and Checking Determinism for Multithreaded Programs, in 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 09), Amsterdam, The Netherlands, August 2009. ACM.

The price to pay for mixed transactional programs

Vasu Singh (EPFL - Lausanne, CH)

The semantics of interactions between transactions managed by a transactional memory (TM) and non-transactional operations, while widely studied, lacks a clear formal specification. Those interactions can vary, sometimes in subtle ways, between TM implementations and underlying memory models.

We formalize a correctness condition, parameterized opacity for TM, which captures the two following intuitive requirements: first, every transaction appears as if it is executed instantaneously with respect to other transactions and non-transactional operations, and second, non-transactional operations conform to a given memory model.

We use our formalization to theoretically investigate the inherent cost of implementing parameterized opacity. We first prove that parameterized opacity requires either instrumenting non-transactional operations (for most memory models) or writing to memory by transactions using potentially expensive read-modify-write instructions (such as compare-and-swap). Then, we show that for a class of relaxed memory models, parameterized opacity can indeed be implemented with constant-time instrumentation of non-transactional writes and no instrumentation of non-transactional reads.

We show that, in practice, parameterizing the notion of correctness with respect to relaxed memory models allows to develop more efficient TM implementations.

Keywords: Transactional memories, Relaxed memory models

Joint work of: Rachid Guerraoui; Thomas Henzinger; Michal Kapalka; Vasu Singh

Verifying Optimistic Concurrency: Prophecy Variables and Backward Reasoning

Serdar Tasiran (Koc University - Istanbul, TR)

Several static proof systems have been developed over the years for verifying shared-memory multithreaded programs. These proof systems make use of auxiliary variables to express mutual exclusion or non-interference among shared variable accesses. Typically, the values of these variables summarize the past of the program execution; consequently, they are known as history variables. Prophecy variables, on the other hand, are the temporal dual of history variables and their values summarize the future of the program execution. In this paper, we show that prophecy variables are useful for locally constructing proofs of systems with optimistic concurrency. To enable the fullest use of prophecy variables in proof construction, we introduce tressa annotations, as the dual of the well-known assert annotations. A tressa claim states a condition for reverse reachability from an end state of the program, much like an assert claim states a condition for forward reachability from the initial state of the program.

We present the proof rules and the notion of correctness of a program for two-way reasoning in a static setting: forward in time for assert claims, backward in time for tressa claims. Even though the interaction between the two is non-trivial, the formalization is intuitive and accessible. We demonstrate how to verify implementations based on optimistic concurrency which is a programming paradigm that allows conflicts to be handled after they occur. We have

incorporated our proof rules into the QED verifier and have used our implementation to verify a handful of small but sophisticated algorithms. Our experience shows that the proof steps and annotations follow closely the intuition of the programmer, making the proof itself a natural extension of implementation.

Keywords: Concurrency, Program Verification, Static Analysis

Joint work of: Serdar Tasiran; Ali Sezgin; Shaz Qadeer

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2010/2430

Full Paper:
http://research.microsoft.com/pubs/103176/tressa.pdf

See also: Microsoft Research Technical Report MSR TR-2009-142, October 13, 2009

A Type System for Data-Centric Synchronization

Frank Tip (IBM TJ Watson Research Center - Hawthorne, US)

Data-centric synchronization groups the fields of objects into atomic sets to indicate which fields must be updated atomically.

Each atomic set has a number of associated units of work, code fragments that preserve the consistency of that atomic set. This paper presents a type system for data-centric synchronization that enables separate compilation and supports atomic sets that span multiple objects, thus allowing recursive data structures to be updated atomically. The type system also supports full encapsulation of objects for more efficient code generation. We evaluate our proposal in the context of AJ, an extension to the Java programming language with data-centric synchronization. We report on the implementation of an extended compiler as an Eclipse plugin, and on refactoring classes from the Java Collections Framework as well as SPECjbb2005, a well-known multi-threaded benchmark, to use atomic sets. Our results suggest that data-centric synchronization has the benefit of very low annotation overhead, while enforcing a rigorous correctness criterion that rules out data races.

Towards full verification of concurrent libraries

Viktor Vafeiadis (Microsoft Research UK - Cambridge, GB)

Modern programming platforms (such as .NET) provide libraries of efficient concurrent data structures. Formal verification of these libraries is particularly challenging, as they involve low-level pointer manipulations and fine-grained non-blocking concurrency.

Nevertheless, full verification of such libraries is possible and, to a large extent, can be done automatically. One way to achieve this is to combine a separation logic shape analysis with rely-guarantee inference and a clever proof search.

These techniques allow us to prove both linearizability (atomicity and functional correctness) and non-blocking liveness properties (such as lock-freedom).

Keywords: Rely-guarantee, linearizability, separation logic, lock-freedom

Language Support for Speculative Parallelization

Kapil Vaswani (Microsoft Research India - Bangalore, IN)

With the advent of multi-core processors, programmers face the arduous task of extracting and expressing parallelism in everyday applications.

However years of experience has shown that parallelization is hard. A fundamental problem with parallelizing programs is the presence of sequential computation that cannot be easily decomposed into independent tasks.

In this talk, I will describe speculative parallelization, a algorithm design and programming idiom that can be quite effective in parallelizing seemingly sequential algorithms. I will also describe two language extensions, a speculative task and a speculative parallel for loop, that let programmers express speculative parallelism naturally.

Keywords: Speculation, parallel programming, language design

Joint work of: Kapil Vaswani; Prakash Prabhu; Ganesan Ramalingam

Abstraction-Guided Synthesis of Synchronization

Eran Yahav (IBM TJ Watson Research Center - Hawthorne, US)

We present a novel framework for automatic inference of efficient synchronization in concurrent programs, a task known to be difficult and error-prone when done manually.

Our framework is based on abstract interpretation and can infer synchronization for infinite state programs. Given a program, a specification, and an abstraction, we infer synchronization that avoids all (abstract) interleavings that may violate the specification, but permits as many valid interleavings as possible.

Combined with abstraction refinement, our framework can be viewed as a new approach for verification where both the program and the abstraction can be modified on-the-fly during the verification process. The ability to modify the program, and not only the abstraction, allows us to remove program interleavings not only when they are known to be invalid, but also when they cannot be verified using the given abstraction.

We implemented a prototype of our approach using numerical abstractions and applied it to verify several interesting programs.

Keywords: Synthesis, verification, concurrency

Joint work of: Martin Vechev; Eran Yahav; Greta Yorsh

Parallelizing a Symbolic Compositional Model-Checking Algorithm

Lenore Zuck (US National Science Foundation - Arlington, US)

We describe a parallel, symbolic, model-checking algorithm, built around a compositional reasoning method. The compositional method, called lo- cal reasoning, builds a collection of per-process (i.e., local) invariants, which together imply a desired global safety property. The local proof computation is a simultaneous fixpoint evaluation, which lends itself to parallelization. Moreover, the locality of the computation makes it possible to partition work across several threads, each with its own BDD manager, while limiting the amount of cross- thread synchronization. Experimental results are encouraging, and show that the parallelized computation can achieve substantial speed-up, often without incur- ring significant memory overhead.

Keywords: Symbolic model checking, verification, multi-core

Joint work of: Ariel Cohen; Kedar Namjoshi; Yaniv Sa'ar; Lenore Zuck; Katya Kisyova