Report from Dagstuhl Seminar 15191

Compositional Verification Methods for Next-Generation Concurrency

Edited by

Lars Birkedal[1], Derek Dreyer[2], Philippa Gardner[3], and Zhong Shao[4]

1 **Aarhus University, DK,** birkedal@cs.au.dk
2 **MPI-SWS – Saarbrücken, DE,** dreyer@mpi-sws.mpg.de
3 **Imperial College London, GB,** pg@doc.ic.ac.uk
4 **Yale University, US,** zhong.shao@yale.edu

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 15191 "Compositional Verification Methods for Next-Generation Concurrency". The seminar was successful and facilitated a stimulating interchange between the theory and practice of concurrent programming, and thereby laid the ground for the development of compositional verification methods that can scale to handle the realities of next-generation concurrency.

Seminar May 3–8, 2015 – http://www.dagstuhl.de/15191
1998 ACM Subject Classification D1.3 Concurrent Programming, D.2.4 Software/Program Verification, D.3 Programming Languages
Keywords and phrases Verification of Concurrent Programs (Models, Logics, Automated Analysis), Concurrent Programming
Digital Object Identifier 10.4230/DagRep.5.5.1

1 Executive Summary

Lars Birkedal
Derek Dreyer
Philippa Gardner
Zhong Shao

One of the major open problems confronting software developers today is how to cope with the complexity of reasoning about large-scale concurrent programs. Such programs are increasingly important as a means of taking advantage of parallelism in modern architectures. However, they also frequently depend on subtle invariants governing the use of shared mutable data structures, which must take into account the potential interference between different threads accessing the state simultaneously. Just figuring out how to express such invariants at all has proven to be a very challenging problem; even more challenging is how to support *local* reasoning about such invariants, i.e., confining the reasoning about them to only the components of the program that absolutely need to know about them.

Fortunately, we are now at a point where verification research has produced the critical foundations needed to tackle this problem: namely, *compositional* methods, which exploit the inherently modular structure of realistic concurrent programs in order to decompose

verification effort along module boundaries. Fascinatingly, a variety of different but related compositional methods have been developed contemporaneously in the last several years:

- **Separation logics:** Separation logic was developed initially as a generalization of Hoare logic – supporting local, compositional reasoning about sequential, heap-manipulating programs – and much of the early work on separation logic has been successfully incorporated into automated verification tools like Smallfoot [2], SLAyer [3], Abductor [6], etc., scaling to handle millions of lines of code. Recently, there have been a series of breakthroughs in adapting separation logic to handle concurrent programs as well. Concurrent separation logic [17] provides course-grained local reasoning about concurrent programs; combining this local reasoning with rely-guarantee reasoning [26] provides fine-grained concurrent reasoning; intertwining abstraction with local reasoning enables a client to reason about the use of a set module [8] without having to think about the underlying implementation using lists or concurrent B-trees; and, very recently, all this has been extended to account for higher-order programs as well [21].
- **Kripke models:** There is a long line of work on the use of semantic models like *Kripke logical relations* [1, 9] (and more recently *bisimulations* [19, 20]) for proving observational equivalence of programs that manipulate local state. Observational equivalence is useful not only for establishing correctness of program transformations (e.g., in compiler certification) but also as a verification method in its own right (e.g., one can prove that a complex but efficient implementation of an ADT is equivalent to a simple but inefficient reference implementation). However, it is only in the last few years that such models have been generalized to account for the full panoply of features available in modern languages: higher-order state, recursion, abstract types, control operators, and most recently concurrency, resulting in some of the first formal proofs of correctness of sophisticated fine-grained concurrent algorithms in a higher-order setting [1, 9, 23]. These advances have come about thanks to the development of more elaborate Kripke structures for representing invariants on local state.
- **Hoare type theory:** Dependent type theory provides a very expressive compositional verification system for higher-order functional programs, so expressive that types can characterize full functional correctness. Traditionally, however, dependent type theories were limited to verification of *pure* programs. Recent work on Hoare type theory (HTT) [15] has shown how to integrate effects into dependent type theory by incorporating Hoare triples as a new primitive type, and prototypes of HTT have been implemented in Coq [7, 16], allowing for imperative programs to be verified mechanically as they are being written. Moreover, first steps of extending HTT with concurrency have recently been taken [14], thus giving hope for a potential future integration of design and verification for higher-order concurrent programs.

All in all, the field of modular concurrency verification is highly active, with groundbreaking new developments in these and other approaches coming out every year. Particularly fascinating is the appearance of deep connections between the different methods. There are striking similarities, for instance, between the advanced Kripke structures used in recent relational models of higher-order state and the semantic models underlying recent concurrent separation logics.

Nevertheless, there are a number of ways in which the advanced models and logics developed thus far are still, to be honest, in their infancy. Most of these approaches, for example, have only been applied to the verification of small, self-contained ADTs and have not yet been scaled up to verify large-scale modular concurrent programs. Moreover, even

the most state-of-the-art compositional methods do not yet account for a number of the essential complexities of concurrent programming as it is practiced today, including:

- **Weak memory models:** The vast majority of state-of-the-art compositional verification methods are proved sound with respect to an operational semantics that assumes a sequentially consistent memory model. However, modern hardware implements weak memory models that allow for many more reorderings of basic operations. Thus there is a clear gap between the verification theory and practice that needs to be filled (for efficiency reasons we, of course, do not want to force programmers/compilers to insert enough memory fence operations to make the hardware behave sequentially consistent). This problem has been known for the last decade, but it is only in the last year or two that formal descriptions of the behavior of programming languages with weak memory models have been developed. Given this foundation, we should now be able to make progress on extending compositional verification methods to weak memory models.
- **Higher-order concurrency:** Higher-order functional abstraction is an indispensable feature of most modern, high-level programming languages. It is also central to a variety of concurrent programming idioms, both established and nascent: work stealing [4], Concurrent ML-style events [18], concurrent iterators [13], parallel evaluation strategies [22], STM [11], reagents [24], and more. Yet, only a few existing logics have been proposed that even attempt to account for higher-order concurrency [21, 14, 12], and these logics are just first steps – for example, they do not presently account for sophisticated "fine-grained" concurrent ADTs. Verification of higher-order concurrent programs remains a largely open problem.
- **Generalizing linearizability:** Sophisticated concurrent data structures often use fine-grained synchronization to maximize the possibilities for parallel access. The classical correctness criterion for such fine-grained data structures is *linearizability*, which ensures that every operation has a linearization point at which it appears (to clients) to atomically take effect. However, existing logics do not provide a way to exploit linearizability directly in client-side reasoning, and moreover the notion does not scale naturally to account for operations (such as higher-order iterators) whose behavior is not semantically atomic. Recently, researchers have started to investigate alternative approaches, based on *contextual refinement* [10, 23]. And methods for reasoning about operations with multiple linearizability points are also being developed.
- **Liveness properties:** Synchronization of concurrent data structures can also affect the progress of the execution of the client threads. Various progress properties have been proposed for concurrent objects. The most important ones are wait-freedom, lock-freedom and obstruction-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which method calls are guaranteed to successfully complete in an execution. Traditional definitions (which are quite informal) of these progress properties are difficult to use in modular program verification because they fail to describe how the progress properties affect clients. It is also unclear how existing separation logics, which were primarily designed for proving partial correctness, can be adapted to prove progress properties. Recently, researchers have started to combine *quantitative reasoning* of resource bounds with separation logics, which offer new possibilities for verifying both safety and liveness properties in a single framework.

Grappling with these kinds of limitations is essential if our verification technology is to be relevant to real-world programs running on modern architectures, and as such it poses exciting new research questions that we as a community are just beginning to explore.

In this seminar, we brought together a wide variety of researchers on concurrency verification, as well as leading experts on concurrent software development in both high- and low-level languages. The goal was to facilitate a stimulating interchange between the theory and practice of concurrent programming, and thereby foster the development of compositional verification methods that can scale to handle the realities of next-generation concurrency.

Among the concrete research challenges investigated in depth during the seminar are the following:

- What are good ways of reasoning about weak memory models? It should be possible to reason about low-level programs that exploit weak memory models (e.g., locks used inside operating systems) but also to reason at higher levels of abstractions for programs that use sufficient locking.
- What is the best way to define a language-level memory model that is nevertheless efficiently implementable on modern hardware. C11 is the state of the art, but it is flawed in various ways, and we heard about a number of different ways of possibly fixing it.
- What is the best way to mechanize full formal verification of concurrent programs, using interactive proof assistants, such as Coq.
- How can we adapt existing and develop new compositional techniques for reasoning about liveness properties of concurrent programs? Can we apply quantitative techniques to reduce the proof of a liveness property to the proof of a stronger safety property? Also, recent work on rely-guarantee-based simulation can prove linearizability of a sophisticated concurrent object by showing the concurrent implementation is a contextual refinement of its sequential specification. We would hope that similar techniques can be used to prove progress properties as well.
- Only recently have researchers begun to propose logics and models for higher-order concurrency [23, 21]. What are the right concurrency abstractions for higher-order concurrent programming idioms as diverse as transactional memory [11], Concurrent ML [18], joins [25], and reagents [24], among others? What is the best way to even specify, let alone verify, programs written in these idioms, and are there unifying principles that would apply to multiple different idioms?
- Most verification work so far has focused on shared-memory concurrency, with little attention paid to message-passing concurrency (except for some recent work on verifying the C♯ joins library). Can the models and logics developed for the former be carried over usefully to the latter, and what is the connection (if any) with recent work on proof-theoretic accounts of session types [5]? Can session types help to simplify reasoning about some classes of concurrent programs, e.g., those that only involve some forms of message passing and not full shared memory?
- A number of recent Kripke models and separation logics have employed *protocols* of various forms to describe the invariants about how the semantic state of a concurrent ADT can evolve over time. But different approaches model protocols differently, e.g., using enriched forms of state transition systems vs. partial commutative monoids. Is there a canonical way of representing these protocols formally and thus better understanding the relationship between different proof methods?
- There seem to be tradeoffs between approaches to concurrency verification based on Hoare logic vs. refinement (unary vs. relational reasoning), with the former admitting a wider variety of formal specifications but the latter offering better support for reasoning about atomicity. Consequently, a number of researchers are actively working on trying to combine both styles of reasoning in a unified framework. What is the best way to do this?

- To what extent do we need linearizability to facilitate client-side reasoning? Is it possible in many cases for clients to rely on a much weaker specification ? And which ways are there to formalize looser notions, e.g. where there are multiple linearization points?
- Now that we are finally developing logics and models capable of verifying realistic concurrent algorithms, can we abstract away useful proof patterns and automate them? What is needed in order to integrate support for concurrent invariants into automated verification tools like SLAyer and Abductor?

These different challenges were discussed through talks and discussions by participants, see the list of talk abstracts below.

References

1. Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL*, 2009.
2. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
3. Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011.
4. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *JPDC*, 37(1):55–69, August 1996.
5. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, 2010.
6. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6), 2011.
7. Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.
8. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, June 2010.
9. Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4&5):477–528, August 2012.
10. I. Filipović, P.W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, 2009.
11. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP*, 2005.
12. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.
13. Doug Lea. The java.util.concurrent ConcurrentHashMap.
14. Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.
15. Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008.
16. Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, 2010.
17. Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 2007.
18. John H. Reppy. *Higher-order concurrency*. PhD thesis, Cornell University, 1992.
19. Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *LICS*, 2007.

20 Eijiro Sumii. A complete characterization of observational equivalence in polymorphic λ-calculus with general references. In *CSL*, 2009.

21 Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, 2013.

22 P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *JFP*, 8(1):23–60, January 1998.

23 A Turon, J Thamsborg, A Ahmed, L Birkedal, and D Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.

24 Aaron Turon. Reagents: expressing and composing fine-grained concurrency. In *PLDI*, 2012.

25 Aaron Turon and Claudio Russo. Scalable join patterns. In *OOPSLA*, 2011.

26 Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

2 Table of Contents

3 Overview of Talks

3.1 Analysing and Optimising Parallel Snapshot Isolaiton

Andrea Cerone (IMDEA Software – Madrid, ES)

Joint work of Bernardi, Giovanni; Cerone, Andrea; Gotsman, Alexey, Yang; Hongseok

Large-scale Internet services often rely on distributed databases that provide consistency models for transactions weaker than serialisability. Unfortunately, we currently lack a systematic understanding of when programmers can use such models without violating correctness. And when an application is correct on a given consistency model, we do not know whether the model can safely be weakened even further to improve performance.

I will present work in progress to address these issues. In the talk I will concentrate on a promising consistency model of Parallel Snapshot Isolation (PSI), which weakens the classical snapshot isolation in a way that allows more efficient distributed implementations. I will present a formalisation of PSI, a criterion for ensuring correctness of applications using it, and a way of optimising the applications to improving performance.

3.2 Phantom Monitors: A Simple Foundation for Modular Proofs of Fine-Grained Concurrent Programs

Adam Chlipala (MIT – Cambridge, US)

Joint work of Bell, Christian J.; Lesani, Mohsen; Malecha, Gregory; Boyer, Stephan; Wang, Peng

I introduce a new approach to verifying fine-grained shared-memory concurrent programs modularly, not based on program logics. Rather, we define an instrumented operational semantics that includes fictitious code to watch all memory accesses and potentially signal a failure, embodying some formal protocol for object sharing. Several variants of the framework have been implemented in Coq at different levels of completeness, and one of our focuses is supporting mostly automated proofs for client code of intricate data structures.

3.3 A Calculus for Relaxed Memory

Karl Crary (Carnegie Mellon University, US)

Joint work of Crary, Karl; Sullivan, Michael J.
Main reference K. Crary, M. J. Sullivan, "A Calculus for Relaxed Memory," in Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'15), pp. 623–636, ACM, 2015; pre-print available from author's webpage.
URL http://dx.doi.org/10.1145/2676726.2676984
URL http://www.cs.cmu.edu/~crary/papers/2015/rmc.pdf

We propose a new approach to programming multi-core, relaxed-memory architectures in imperative, portable programming languages. Our memory model is based on explicit, programmer-specified requirements for order of execution and the visibility of writes. The

compiler then realizes those requirements in the most efficient manner it can. This is in contrast to existing memory models, which – if they allow programmer control over synchronization at all – are based on inferring the execution and visibility consequences of synchronization operations or annotations in the code.

We formalize our memory model in a core calculus called RMC. Outside of the programmer's specified requirements, RMC is designed to be strictly more relaxed than existing architectures. It employs an aggressively nondeterministic semantics for expressions, in which actions can be executed in nearly any order, and a store semantics that generalizes Sarkar, *et al.*'s and Alglave, *et al.*'s models of the Power architecture. We establish several results for RMC, including sequential consistency for two programming disciplines, and an appropriate notion of type safety. All our results are formalized in Coq.

3.4 Modular Termination Verification for Non-blocking Concurrency

Pedro Da Rocha Pinto (Imperial College London, GB)

We present Total-TaDA, a program logic for verifying the total correctness of concurrent programs: that such programs both terminate and produce the correct result. The termination behaviour of a single thread can be conditional on the behaviour of its concurrent environment. With Total-TaDA, we are able to specify such constraints. This allows us to verify total correctness for non-blocking algorithms, such as a counter and a stack. Moreover, our approach is modular: we can verify the operations of a module independently, and build up modules on top of each other.

3.5 Speculation in Higher-Order Separation Logics (and why it's tricky)

Thomas Dinsdale-Young (Aarhus University, DK)

Joint work of Dinsdale-Young, Thomas; Svendsen, Kasper

When relating an implementation to an abstract specification, the abstract behaviours can depend on future concrete behaviours. We briefly motivate why this occurs, and consider how we might reason about a simple example in a modular way using a separation logic. Unfortunately, the naive approach is not sound, and we see why it leads to inconsistency.

3.6 Compositional C11 Program Transformation

Mike Dodds (University of York, GB)

Joint work of Batty, Mark; Dodds, Mike; Gotsman, Alexey

One objective for language-level relaxed memory models is to support program transformations – i.e. compiler optimisations. However, it's extremely subtle to calculate which

transformations are valid. This talk is about a theory for program transformations on the C11 relaxed model. Our theory is compositional: for each transformation, a limited number of executions represent all interactions with the context. To express these interactions, we use a partially-ordered record called a history (the set of histories could be seen as a kind of denotation). Our theory builds on ideas from C11 library abstraction: replacing a specification with an implementation is one instance of program transformation. This work is still in progress, but we already cover the core of the C11 model and many important transformations.

3.7 Static Verification of GPU Kernels

Alastair F. Donaldson (Imperial College London, GB)

Joint work of Betts, Adam; Chong, Nathan; Donaldson, Alastair F.; Ketema, Jeroen; Qadeer, Shaz; Thomson, Paul; Wickerson, John
Main reference A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, J. Wickerson, 'The Design and Implementation of a Verification Technique for GPU Kernels," ACM Transactions on Programming Languages and Systems (TOPLAS), 37(3):10:1–10:49, May 2015.
URL http://dx.doi.org/10.1145/2743017

During the presentation I gave a demonstration of GPUVerify, a static race-freedom verification tool for GPU kernels. GPUVerify enables scalable verification of massively parallel kernels through a combination of abstraction and sequentialization. Abstraction is applied to reduce the verification problem to the task of checking whether it is possible for two arbitrary threads to race. Sequentialization then exploits properties of the barrier-based GPU synchronization model so that verification for a pair of threads boils down to checking assertion-based correctness of a sequential program, whose size is linear in that of the source code for the original kernel (which itself is independent of the number of threads that execute the kernel).

This is join work with the Multicore Programming Group at Imperial, and with Shaz Qadeer at Microsoft Research, and is described in a recent TOPLAS journal article.

References

1 Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10:1–10:49, May 2015.

3.8 Making Sense of Rust (Work in Preservation)

Derek Dreyer (MPI-SWS – Saarbrücken, DE)

Joint work of Dreyer, Derek; Jung, Ralf; Turon, Aaron

Rust is a new language for "safe systems programming" developed at Mozilla, which uses an affine type system to guarantee type/memory safety and data race freedom. While the core type system is relatively simple and restrictive, essentially prohibiting aliased mutable state, many Rust libraries make significant internal use of "unsafe blocks" in order to escape this restriction. These uses of "unsafe" are supposedly encapsulated behind safe interfaces, but

they also fundamentally affect the meaning of types. For example, Rust libraries like Rc and Cell exhibit the phenomenon of "interior mutability", whereby a supposedly "immutable" (roughly, "read-only") operation on an object may in fact mutate its private state, so long as it does so in a way that does not violate the views of other aliases to the object. This has significant implications for concurrency, in particular leading to the need for a "Send" trait describing when a type is "thread-safe". It's also easy to get wrong, as evidenced by a recent soundness bug that was uncovered in the "scoped threads" API. In this work, which we have not yet even begun (!), we aim to develop a semantic model of Rust's type system, based on Kripke logical relations and concurrent separation logic, which will enable us to make sense of what Rust types mean and to verify that the unsafe implementations of Rust libraries in fact preserve the end-to-end safety guarantees of the language.

3.9 An operational approach to relaxed memory models

Xinyu Feng (Univ. of Science & Technology of China – Suzhou, CN)

Joint work of Zhang, Yang; Feng, Xinyu
URL http://staff.ustc.edu.cn/~xyfeng/research/publications/OHMM.html

We present OHMM, an operational variation of the Happens-before Memory Model (HMM), the basis of Java memory model (JMM). OHMM is specified by giving an operational semantics to a language running on an abstract machine designed to simulate HMM. Thanks to its generative nature, the model naturally prevents out-of-thin-air reads. On the other hand, it uses a novel replay mechanism to allow instructions to be executed multiple times, which can be used to model many useful speculations and optimization. The model satisfies DRF- guarantee. It is weaker than JMM for lockless programs, thus can accommodate more optimization, such as the reordering of independent memory accesses that is not valid in JMM. Also many of the "ugly" examples in JMM are no longer ugly in our model. We hope OHMM can serve as the basis for new memory models for Java-like languages.

3.10 Formally Specifying POSIX File Systems

Philippa Gardner (Imperial College London, GB)

File system operations exhibit complex behaviour: they perform multiple actions affecting different parts of the state. This is further exacerbated when the operations are used concurrently. POSIX is a standard for operating systems, with a substantial part devoted to specifying file system operations. The specification is given in English, contains ambiguities and is generally under-specified with respect to concurrent behaviour. Therefore, it is not clear what clients may expect and what implementations must do. We extend modern concurrent program logics with a novel formalism for specifying multiple actions performed by an operation, which may be atomic, non-atomic or a combination of both, and give proof rules for client and implementation reasoning. With this formalism we give a formal specification to a common fragment of POSIX file system operations, and reason about clients such as lock files and an implementation of half-duplex pipes.

3.11 An Unsophisticated Higher-Order-ish Logic for Modular Specification and Verification of Total Correctness Properties of Fine-Grained Concurrent Imperative Programs

Bart Jacobs (KU Leuven, BE)

Many powerful higher-order logics have been proposed for the modular specification and verification of fine-grained concurrent imperative programs. In this talk, I present a logic that is fairly close to what my VeriFast modular verification tool for C and Java implements. To achieve higher-order-ishness (higher-order assertions, nested triples, assertions in the heap, etc.), a relatively simple approach is followed: assertion lambda applications and nested triples may occur only in positive positions. Negative facts can be passed around in the form of lemma lambdas, i.e. ghost command lambdas. We prove termination of such higher-order ghost code using call permissions, a technique we are presenting at ECOOP 2015 this summer.

3.12 Reasoning about possible values in concurrency

Cliff B. Jones (Newcastle University, GB)

In joint research with Ian Hayes (Queensland) we are using a notation to express the 'possible values' of variables. So, for example, in a post condition of one process (we can not only talk about the initial and final values of a variable which might be changed by another process) – we can also specify in terms of the set of values that the environment might assign to a shared variable. Combined with rely/guarantee reasoning, this appears to offer clear and tractable specifications and reasoned designs. The possible values notation was shown on the example of Simpson's four-slot implementation of Asynchronous Communication Mechanisms (ACMs).

3.13 Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning

Ralf Jung (MPI-SWS – Saarbrücken, DE)

Joint work of Jung, Ralf; Swasey, David; Sieczkowski, Filip; Svendsen, Kasper; Turon, Aaron; Birkedal, Lars; Dreyer, Derek

Main reference R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, D. Dreyer, "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning," in Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'15), pp. 637–650, ACM, 2015.

URL http://dx.doi.org/10.1145/2676726.2676980

We present Iris, a concurrent separation logic with a simple premise: monoids and invariants are all you need. Partial commutative monoids enable us to express – and invariants enable

us to enforce – user-defined protocols on shared state, which are at the conceptual core of most recent program logics for concurrency. Furthermore, through a novel extension of the concept of a view shift, Iris supports the encoding of logically atomic specifications, i.e., Hoare-style specs that permit the client of an operation to treat the operation essentially as if it were atomic, even if it is not.

3.14 The Push/Pull Model of Transactions

Eric Koskinen (IBM TJ Watson Research Center – Yorktown Heights, US)

Joint work of Koskinen, Eric; Parkinson, Matthew
Main reference E. Koskinen, M. J. Parkinson, "The Push/Pull model of transactions," in Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'15), pp. 186–195, ACM, 2015; pre-print available from author's webpage.
URL http://dx.doi.org/10.1145/2737924.2737995
URL http://researcher.watson.ibm.com/researcher/files/us-ejk/pushpull.pdf

We present a general theory of serializability, unifying a wide range of transactional algorithms, including some that are yet to come. To this end, we provide a compact semantics in which concurrent transactions PUSH their effects into the shared view (or UNPUSH to recall effects) and PULL the effects of potentially uncommitted concurrent transactions into their local view (or UNPULL to detangle). Each operation comes with simple criteria given in terms of commutativity (Lipton's left-movers and right-movers).

The benefit of this model is that most of the elaborate reasoning (coinduction, simulation, subtle invariants, etc.) necessary for proving the serializability of a transactional algorithm is already proved within the semantic model. Thus, proving serializability (or opacity) amounts simply to mapping the algorithm on to our rules, and showing that it satisfies the rules' criteria.

3.15 Curry-Howard for GUIs via Linear Temporal Classical Linear Logic

Neel Krishnaswami (University of Birmingham, GB)

Modern graphical user interface are structured with an event-driven architecture: programmers write programs as a collection of small imperative callbacks, which are invoked by an event loop as program events occur. That is, they must write higher-order imperative programs in continuation-passing style, which is notoriously challenging.

Using ideas from realizability theory, it is possible to build a model of classical linear logic on top of an event-based architecture. Since classical linear logic has a proof theory in terms of process calculi, we gain a neat explanation of why programmers talk about GUI programs in terms of concurrency, even though they implement them in terms of state and control. Furthermore, we now also have a type structure upon which we can build powerful abstractions – historically the bane of UI toolkits.

3.16 Owicki-Gries Reasoning for Weak Memory Models

Ori Lahav (MPI-SWS – Kaiserslautern, DE)

Joint work of Lahav, Ori; Vafeiadis, Viktor
Main reference O. Lahav, V. Vafeiadis, "Owicki-Gries Reasoning for Weak Memory Models," in Proc. of the 42nd International Colloquium on Automata, Languages, and Programming (ICALP'15) – Part II, LNCS, Vol. 9135, pp. 311–323, Springer, 2015.
URL http://dx.doi.org/10.1007/978-3-662-47666-6_25

We show that even in the absence of auxiliary variables, the well-known Owicki-Gries method for verifying concurrent programs is unsound for weak memory models. By strengthening its non-interference check, however, we obtain OGRA, a program logic that is sound for reasoning about programs in the release-acquire fragment of the C11 memory model. We demonstrate the usefulness of this logic by applying it to several challenging examples, ranging from small litmus tests to an implementation of the RCU synchronization primitives.

3.17 A Program Logic for Contextual Refinement of Concurrent Objects under Fair Scheduling

Hongjin Liang (Univ. of Science & Technology of China – Suzhou, CN)

Joint work of Liang, Hongjin; Feng, Xinyu

Existing program logics on concurrent object verification either ignore progress properties, or aim for non-blocking progress (e.g., lock-freedom and wait-freedom), which cannot be applied to blocking algorithms that progress only under fair scheduling.

We present a new program logic for compositional verification of contextual refinement of concurrent objects under fair scheduling. As a key application, we show that starvation-freedom and linearizability of concurrent objects with blocking algorithms can be reformulated as contextual refinement, which can be verified using our program logic. With the logic, we have successfully verified starvation-freedom of simple algorithms using ticket locks, the two-lock queue algorithm and the lock-coupling list algorithm.

3.18 Formal Verification and Linux-Kernel Concurrency

Paul McKenney (IBM – Beaverton, US)

Main reference P. E. McKenney, "Formal Verification and Linux-Kernel Concurrency," presentation to Dagstuhl workshop 15191.
URL http://materials.dagstuhl.de/files/15/15191/15191.PaulMcKenney.Slides.pdf

This presentation reviews Linux-kernel validation, including its occasional use of formal verification, and presents conditions that a formal-verification tool would need to meet in order to be useful as part of the Linux kernel's regression testing.

3.19 Linearizability: Who Really Needs It?

Paul McKenney (IBM – Beaverton, US)

Main reference P. E. McKenney, "Linearizability: Who Really Needs It?," Presentation to Dagstuhl 15191.
URL http://materials.dagstuhl.de/files/15/15191/15191.PaulMcKenney1.Preprint.pdf

Critique of the overuse of linearizability.

3.20 Some Examples of Kernel-Hacker Informal Correctness Reasoning

Paul McKenney (IBM – Beaverton, US)

Main reference P. E. McKenney, "Some Examples of Kernel-Hacker Informal Correctness Reasoning," Technical Report paulmck.2015.06.17a.
URL http://www2.rdrop.com/users/paulmck/techreports/IntroRCU.2015.06.17a.pdf

The examples include: (1) split counters, (2) RCU infrastructure, (3) RCU Small Bag use case, RCU Large Bag use case.

Also illustrates kernel-hacker reasoning surrounding RCU, along with one method of restoring consistency when using RCU. (Yes, there are other methods.)

3.21 Designing a Lock-Free Range Management Algorithm

Maged M. Michael (IBM TJ Watson Research Center – Yorktown Heights, US)

The talk describes the design process of a lock-free algorithm for allocation and deallocation of arbitrary large ranges. The algorithm is targeted to serve as a backend for known bounded-block-size lock-free memory allocators. It can serve as a user-level alternative to the mmap and munmap system calls in cases where the latter are unavaiulable or unsuitable. The algorithm aims to guarantee full coalescing. It uses only single word primitives: read, write, compare-and- swap; and it does not require any operating system calls. The algorithm supports continuous space availability, i.e., the space unavailable for allocation is bounded by the sum of allocated space and pending allocation requests.

3.22 Viper – A Verification Infrastructure for Permission based Reasoning

Peter Mueller (ETH Zürich, CH)

Joint work of Juhasz, Uri; Kassios, Ioannis T.; Müller, Peter; Novacek, Milos; Schwerhoff, Malte; Summers, Alexander J.
Main reference U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," Unpublished manuscript.
URL http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=JKMNSS14.pdf

The automation of verification techniques based on first-order logic specifications has benefited greatly from verification infrastructures such as Boogie and Why. These offer an intermediate language that can express diverse language features and verification techniques, as well as back-end tools such as verification condition generators.

However, these infrastructures are not well suited for verification techniques based on separation logic and other permission logics, because they do not provide direct support for permissions and because existing tools for these logics often prefer symbolic execution over verification condition generation. Consequently, tool support for these logics is typically developed independently for each technique, dramatically increasing the burden of developing automatic tools for permission-based verification.

In this talk, we present a verification infrastructure whose intermediate language supports an expressive permission model natively. We provide tool support, including two back-end verifiers, one based on symbolic execution, and one on verification condition generation; this facilitates experimenting with the two prevailing techniques in automated verification. Various existing verification techniques can be implemented via this infrastructure, alleviating much of the burden of building permission-based verifiers, and allowing the developers of higher-level techniques to focus their efforts at the appropriate level of abstraction.

3.23 Structures with Intrinsic Sharing, Subjectively

Aleksandar Nanevski (IMDEA Software – Madrid, ES)

Joint work of Nanevski, Aleksandar; Ilya Sergey; Anindya Banerjee

The talk presents a new design pattern for proving correctness of data structures with deep sharing, such as graphs. The idea is to use subjective kind of auxiliary state, based on PCMs, which allows for threads to record their own changes to the datastructure, as well as the modifications performed by the interfering threads. The talk also discusses a rule for hiding, which introduces new auxiliary state within a delimited scope

3.24 An operational semantics for C/C++11 concurrency

Kyndylan Nienhuis (University of Cambridge, GB)

Joint work of Nienhuis, Kyndylan; Memarian, Kayvan; Pichon-Pharabod, Jean; Batty, Mark; Sewell, Peter

The axiomatic style of the C11 concurrency model makes it difficult to explore the possible execution of programs without exhaustive enumeration of all their candidate executions. Furthermore, since the rest of C is defined in an operational style, it is difficult to extend the concurrency model to a semantics for the whole language.

We present ongoing research on an operational concurrency model for C11 that is equivalent to the axiomatic model, executable, and integratable with an operational semantics for sequential C. This work also reveals omissions in the definition of C: notions such as lifetime and undefined behaviour are defined for sequential C only, and we discovered that their definitions do not generalise to concurrent C.

3.25 Investigating Weak Memory Performance

Scott Owens (University of Kent, GB)

Joint work of Ritson, Carl; Owens, Scott

This talk will describe some preliminary and ongoing work into the real-world performance implications of fence placement strategies on ARM and POWER architectures.

3.26 Polarized Substructural Session Types

Frank Pfenning (Carnegie Mellon University, US)

We provide an overview of session-typed message-passing concurrent programming, which arises from a Curry-Howard interpretation of (intuitionistic) linear logic. Most recent work considers multiple structural properties (linear, affine, and unrestricted) connected by modal operators. The same modal operators (often called "up" and "down") can also be used to mediate between positive and negative linear proposition, one corresponding to output and one to input.

References

1 Frank Pfenning and Dennis Griffith. Polarized substructural session types. In A. Pitts, editor, *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, pages 3–22, London, England, April 2015. Springer LNCS 9034. Invited talk.

3.27 An attempt at fixing C11 concurrency

Jean Pichon-Pharabod (University of Cambridge, GB)

The memory model of the C programming language, which defines what values a read in a concurrent program can read, allows reads to read value that are not constructed by the program, but appear "out of thin air". We argue that this problem is due to the memory model considering the wrong objects, namely configurations in the naive event structure of the program. We propose an alternative memory model for locks and non-atomic and relaxed accesses based on considering the whole event structure.

3.28 Automated and Modular Refinement Reasoning for Concurrent Programs

Shaz Qadeer (Microsoft Corporation – Redmond, US)

Joint work of Hawblitzel, Chris; Petrank, Erez; Qadeer, Shaz; Tasiran, Serdar
Main reference C. Hawblitzel, E. Petrank, S. Qadeer, S. Tasiran, "Automated and Modular Refinement Reasoning for Concurrent Programs," in Proc. of the 27th International Conference on Computer Aided Verification (CAV'15), LNCS, Vol. 9207, pp. 449–465, Springer, 2015; preliminary technical report available, MSR-TR-2015-8, Microsoft Research, 2015.
URL http://dx.doi.org/10.1007/978-3-319-21668-3_26
URL http://research.microsoft.com/apps/pubs/?id=238907
URL http://research.microsoft.com/apps/pubs/?id=258112

We present CIVL, a language and verifier for concurrent programs based on automated and modular refinement reasoning. CIVL supports reasoning about a concurrent program at many levels of abstraction. Atomic actions in a high- level description are refined to fine-grain and optimized lower-level implementations. Modular specifications and proof annotations, such as location invariants and procedure pre- and post-conditions, are specified separately, independently at each level in terms of the variables visible at that level. We have implemented CIVL as an extension to the Boogie language and verifier. We have used CIVL to refine a realistic concurrent garbage collection algorithm from a simple high-level specification down to a highly-concurrent implementation described in terms of individual memory accesses.

3.29 CoLoSL: Concurrent Local Subjective Logic

Azalea Raad (Imperial College London, GB)

Joint work of Raad, Azalea; Villard, Jules; Gardner, Philippa
Main reference A. Raad, J. Villard, P. Gardner, "CoLoSL: Concurrent Local Subjective Logic," in Proc. of the 24th European Symposium on Programming on Programming Languages and Systems (ESOP'15), LNCS, Vol. 9032, pp. 710–735, Springer, 2015.
URL http://dx.doi.org/10.1007/978-3-662-46669-8_29

A key difficulty in verifying shared-memory concurrent programs is reasoning compositionally about each thread in isolation. Existing verification techniques for fine-grained concurrency

typically require reasoning about either the entire shared state or disjoint parts of the shared state, impeding compositionality. In this work we introduce the program logic CoLoSL, where each thread is verified with respect to its subjective view of the global shared state. This subjective view describes only that part of the state accessed by the thread. Subjective views may arbitrarily overlap with each other, and expand and contract depending on the resource required by the thread. This flexibility gives rise to small specifications and, hence, more compositional reasoning for concurrent programs. We demonstrate our reasoning on a range of examples, including a concurrent computation of a spanning tree of a graph.

3.30 Concurrency-Aware Linearizability

Noam Rinetzky (Tel Aviv University, IL)

Joint work of Hemed, Nir; Rinetzky, Noam
Main reference N. Hemed, N. Rinetzky, "Brief Announcement: Concurrency-Aware Linearizability," in Proc. of the 2014 ACM Symp. on Principles of Distributed Computing (PODC'14), pp. 209–211, ACM, 2014; pre-print available from author's webpage.
URL http://dx.doi.org/10.1145/2611462.2611513
URL http://www.cs.tau.ac.il/~maon/pubs/podc14.pdf

Linearizabilty allows to describe the behaviour of concurrent objects using sequential specifications. Unfortunately, as we show in this paper, sequential specifications cannot be used for concurrent objects whose observable behaviour in the presence of concurrent operations should be different than their behaviour in the sequential setting. As a result, such concurrency-aware objects do not have formal specifications, which, in turn, precludes formal verification.

In this paper we present Concurrency Aware Linearizability (CAL), a new correctness condition which allows to formally specify the behaviour of a certain class of concurrency-aware objects. Technically, CAL is formalized as a strict extension of linearizability, where concurrency-aware specifications are used instead of sequential ones. We believe that CAL can be used as a basis for modular formal verification techniques for concurrency-aware objects.

3.31 Anatomy of mechanized reasoning about fine-grained concurrency

Ilya Sergey (IMDEA Software – Madrid, ES)

URL http://ilyasergey.net/slides/2015-Sergey-al-Dagstuhl.pdf

In this talk, I will give a quick hands-on demo, explaining the structure of the proofs when verifying fine-grained concurrent programs in the recently proposed Coq-based framework of Fine-grained Concurrent Separation Logic.

I will outline key stages of formalization of characteristic concurrent protocols, explaining the encoding of atomic actions and stable specifications. I will also outline typical proof patterns, appearing during the reasoning about composition of concurrent specifications.

3.32 Using Iris as a meta-language for logical relations

Kasper Svendsen (Aarhus University, DK)

Joint work of Svendsen, Kasper; Birkedal, Lars; Askarov, Aslan; Krog-Jespersen, Morten

In this talk, I argue that Iris is well-suited as a meta-language for defining binary Step-indexed Kripke Logical Relations. Step-indexed Kripke Logical Relations provide a very powerful proof technique for reasoning about realistic languages. However, they can be difficult to define and work with directly, requiring explicit reasoning about steps and the existence of recursively-defined Kripke worlds. Using Iris as a meta-language, we can hide the steps and avoid the construction of recursively-defined worlds, by piggy-backing on Iris' impredicative invariants and monoids.

3.33 Verifying Read-Copy-Update in a Logic for Weak Memory

Joseph Tassarotti (Carnegie Mellon University, US)

Joint work of Tassarotti, Joseph; Dreyer, Derek; Vafeiadis, Viktor
Main reference J. Tassarotti, D. Dreyer, V. Vafeiadis, "Verifying read-copy-update in a logic for weak memory," in Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'15), pp. 110–120, ACM, 2015.
URL http://dx.doi.org/10.1145/2737924.2737992

Read-Copy-Update (RCU) is a technique for letting multiple readers safely access a data structure while a writer concurrently modifies it. It is used heavily in the Linux kernel in situations where fast reads are important and writes are infrequent. Optimized implementations rely only on the weaker memory orderings provided by modern hardware, avoiding the need for expensive synchronization instructions (such as memory barriers) as much as possible.

Using GPS, a recently developed program logic for the C/C++11 memory model, we verify an implementation of RCU for a singly-linked list assuming "release-acquire" semantics. Although release-acquire synchronization is stronger than what is required by real RCU implementations, it is nonetheless significantly weaker than the assumption of sequential consistency made in prior work on RCU verification. Ours is the first formal proof of correctness for an implementation of RCU under a weak memory model

3.34 Software verification under weak memory consistency

Viktor Vafeiadis (MPI-SWS – Kaiserslautern, DE)

Weak memory consistency makes reasoning about concurrent programs rather challenging as it invalidates many of the traditional reasoning techniques that are sound under sequential consistency. The talk demonstrates some of the challenges involved and possible solutions.

3.35 Open Problems and State-of-Art of Session Types

Nobuko Yoshida (Imperial College London, GB)

Main reference R. Demangeon, K. Honda, R. Hu, R. Neykova, N. Yoshida, "Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python," Formal Methods in System Design, 46(3):197–225, 2015.
URL http://dx.doi.org/10.1007/s10703-014-0218-8

We give a summary of our recent research developments on multiparty session types for verifying distributed and concurrent programs, and our collaborations with industry partners and a major, long-term, NSF-funded project (Ocean Observatories Initiatives) to provide an ultra large-scale cyberinfrustracture (OOI CI) for 25-30 years of sustained ocean measurements to study climate variability, ocean circulation and ecosystem dynamics. We shall first talk how Robin Milner, Kohei Honda and Yoshida started collaborations with industry to develop a web service protocol description language called Scribble and discovered the theory of multiparty session types through the collaborations. We then talk about the recent developments in Scribble and the runtime session monitoring framework currently used in the OOI CI.

Participants

- Lennart Beringer
Princeton University, US
- Lars Birkedal
Aarhus University, DK
- Andrea Cerone
IMDEA Software – Madrid, ES
- Adam Chlipala
MIT – Cambridge, US
- Karl Crary
Carnegie Mellon University, US
- Pedro Da Rocha Pinto
Imperial College London, GB
- Thomas Dinsdale-Young
Aarhus University, DK
- Mike Dodds
University of York, GB
- Alastair F. Donaldson
Imperial College London, GB
- Cezara Dragoi
IST Austria – Klosterneuburg, AT
- Derek Dreyer
MPI-SWS – Saarbrücken, DE
- Xinyu Feng
Univ. of Science & Technology of China – Suzhou, CN
- Philippa Gardner
Imperial College London, GB
- Alexey Gotsman
IMDEA Software – Madrid, ES
- Aquinas Hobor
National Univ. of Singapore, SG
- Jan Hoffmann
Yale University, US
- Chung-Kil Hur
Seoul National University, KR
- Bart Jacobs
KU Leuven, BE
- Cliff B. Jones
Newcastle University, GB
- Ralf Jung
MPI-SWS – Saarbrücken, DE
- Eric Koskinen
IBM TJ Watson Research Center – Yorktown Heights, US
- Neel Krishnaswami
University of Birmingham, GB
- Ori Lahav
MPI-SWS – Kaiserslautern, DE
- Hongjin Liang
Univ. of Science & Technology of China – Suzhou, CN
- Paul McKenney
IBM – Beaverton, US
- Maged M. Michael
IBM TJ Watson Res. Center – Yorktown Heights, US
- Peter Müller
ETH Zürich, CH
- Aleksandar Nanevski
IMDEA Software – Madrid, ES
- Kyndylan Nienhuis
University of Cambridge, GB
- Scott Owens
University of Kent, GB
- Frank Pfenning
Carnegie Mellon University, US
- Jean Pichon-Pharabod
University of Cambridge, GB
- Francois Pottier
INRIA – Le Chesnay, FR
- Shaz Qadeer
Microsoft Corporation – Redmond, US
- Azalea Raad
Imperial College London, GB
- John Reppy
University of Chicago, US
- Noam Rinetzky
Tel Aviv University, IL
- Claudio Russo
Microsoft Research UK – Cambridge, GB
- Ilya Sergey
IMDEA Software – Madrid, ES
- Peter Sewell
University of Cambridge, GB
- Zhong Shao
Yale University, US
- Kasper Svendsen
Aarhus University, DK
- Joseph Tassarotti
Carnegie Mellon University, US
- Viktor Vafeiadis
MPI-SWS – Kaiserslautern, DE
- Martin T. Vechev
ETH Zürich, CH
- Nobuko Yoshida
Imperial College London, GB