Call for Participation: 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)

The IEEE/ACM Automated Software Engineering (ASE) conference series is the premier research forum for automated software engineering. Each year, it brings together researchers and to discuss foundations, techniques, and tools for automating the analysis, design, implementation, testing, and maintenance of large software systems. ASE 2016 covers topics including, but not limited to:

- Automated reasoning techniques
- Component-based systems
- Computer-supported cooperative work
- Configuration management
- Data mining for software engineering
- Domain modeling and meta-modeling
- Empirical software engineering
- Human-computer interaction
- Knowledge acquisition and management
- Maintenance and evolution
- Model-driven development
- Model transformations
- Program synthesis & transformations
- Modeling language semantics
- Open systems development
- Program comprehension
- Re-engineering
- Requirements engineering
- Specification languages
- Software analysis
- Software architecture and design
- Software product line engineering
- Software visualization
- Testing, verification, and validation

Technical Program:

53 technical research papers • 14 new idea papers • 4 experience reports • 20 tool demos
See website for list of all papers.

4 Workshops:

- Workshop on Specification, Comprehension, Testing and Debugging of Concurrent Programs
- First International Workshop on Software Refactoring
- Fifth International Workshop on Software Mining
- Workshop on Formal Methods for Analysis of Business System

9 Tutorials:

Build Your Own Model Checker with PAT • Product Line Variability • Software Analytics: Challenges and Opportunities • Automated GUI Testing of Android Apps: Challenges, Approaches, Tools, and Best Practices • Search Based Software Engineering: Foundations, Challenges and Recent Advances • Learn to Build Automated Software Analysis Tools with Graph Paradigm and Interactive Visual Framework • Testing Stochastic Software • Mining and Modelling Unstructured Data • Using Docker Containers to Improve Reproducibility in Software Engineering Research

Sponsors:

[List of sponsors, including Microsoft Research, Fujitsu, LEE Foundation, IEEE, LARC Research Centre, etc.]

Organizing Committee

General Chair
David Lo, Singapore Management University

Program Co-Chairs
Sven Apel, University of Passau
Sarfraz Khurshid, Uni. of Texas at Austin

Tool Demo Co-Chairs
Yuanfeng Cai, Drexel University
Yang Liu, Nanyang Technological University

Workshop Co-Chairs
Leonardo Mariani, Uni. of Milano Bicocca
Zhencang Xing, Nanyang Technological U.

Tutorial Co-Chairs
Tien Nguyen, Iowa State University
Rui Abreu, Palo Alto Research Center

Doctoral Symposium Co-Chairs
Massimiliano Di Penta, University of Sannio
Tien N. Nguyen, Iowa State University

Review Process Chairs
Yuriy Brun, University of Massachusetts
Claire Le Goues, Carnegie Mellon University

Local/Finance Chair
Lingxiao Jiang, Singapore Management U.

Sponsorship Chair
Siau-Cheng Khoo, National U. of Singapore

Proceedings Chair
Jun Sun, Singapore U. of Tech. and Design

Publicity Co-Chairs
Christian Kästner, Carnegie Mellon University
Shahar Maoz, Tel Aviv University
Xin Xia, Zhejiang University

Social Media Co-Chairs
Lingming Zhang, University of Texas at Dallas
Lucia, University of Luxembourg

Local Advisory Committee
Jin Song Dong, National U. of Singapore
Steven Miller, Singapore Management U.

Webmasters
Pavneet S. Kochhar, Singapore Manage. U.
Ferdian Thung, Singapore Management U.

September 3–7, Singapore • www.ase2016.org • Early registration ends July 22
Keynotes:

**Program Generation for Performance**

It has become extraordinarily difficult to write software that performs close to optimally on complex modern microarchitectures. Particularly plagued are domains that require complex mathematical computations such as multimedia processing, communication, control, graphics, and machine learning. In these domains, performance-critical components are usually written in C (with possible extensions) and often even in assembly, carefully "tuned" to the platform's architecture and microarchitecture. The result is usually long, rather unreadable code that needs to be re-written or re-tuned with every platform upgrade. On the other hand, the performance penalty for relying on straightforward, non-tuned, "more elegant" implementations can be often a factor of 10, 100, or even more.

The overall problem is one of productivity, maintainability, and quality (namely performance), i.e., software engineering. However, even though a large set of sophisticated software engineering theory and tools exist, it appears that to date this community has not focused much on mathematical computations nor performance in the detailed, close-to-optimal sense above. The reason for the latter may be that performance, unlike various aspects of correctness, is not syntactic in nature (and in reality is often even unpredictable and, well, messy).

The aim of this talk is to draw attention to the performance/productivity problem for mathematical applications and to make the case for a more interdisciplinary attack. As a set of thoughts in this direction we offer some of the lessons we have learned in the last decade in our own research on Spiral (www.spiral.net), a program generation framework for numerical kernels. Key techniques used in Spiral include staged declarative domain-specific languages to express algorithm knowledge and algorithm transformations, the use of platform-cognizant rewriting systems for parallelism and locality optimizations, and the use of search and machine learning techniques to navigate possible spaces of choices. Experimental results show that the code generated by Spiral competes with, and sometimes outperforms, the best available human-written code. Spiral has been used to generate part of Intel's commercial libraries IPP and MKL.

**Changing Microsoft's Build: Revolution or Evolution**

Tens of thousands of Microsoft engineers build and test hundreds of software products several times a day. It is essential that this continuous integration scales, guarantees short feedback cycles, and functions reliably with minimal human intervention. During the past three years TSE's charter has been to shorten this cycle time. We went after this goal in two ways: Evolution via CloudBuild and Revolution via Concord.

CloudBuild is a build service infrastructure, now being used by all major product groups in Microsoft, like Azure, Bing, Office, SQL except for Windows. CloudBuild addresses all aspects of a continuous integration workflow, like builds, test and code analysis, but also drops, package and symbol creation and storage. CloudBuild supports multiple build languages as long as they fulfill a coarse grained IO based contract. CloudBuild uses content based caching to run build-related tasks only when needed. Lastly, it builds on many machines in parallel. The speed ups of build and testing range from 1.2x to 10x. CloudBuild aims to rapidly onboard teams and hence has to support non-deterministic build tools and specification languages that under-declare dependencies. CloudBuild, being a reliable build service in the presence of unreliable components, currently achieves service availability better than 99%.

Windows went a different path. Their past build exhaust was so massive that building Windows in the cloud and bringing the build results back for testing on corp.-net. was considered infeasible. So they decided to move to a new build language, codename Concord. By construction, Concord guarantees reliable builds, no over-build, and allows for efficient distribution. Adopting Concord has led to immense performance improvements, we have seen up to 100X speedup for Windows builds. But the path has been long and rocky, since it not only requires a substantial rewrite of existing build logic, but also all related developer and build lab processes have to change. Whether evolution or revolution is the right path forward -- the verdict is still out.

**The Power of Probabilistic Thinking**

Traditionally, software engineering has dealt in absolutes. For instance, we talk about a system being "correct" or "incorrect", with the shades of grey in between occasionally acknowledged but rarely dealt with explicitly. And we typically employ logical, algebraic, relational and other representations and techniques that help us reason about software in such absolute terms. There of course have been notable exceptions to this, such as the use of statistical techniques in testing and debugging. But by and large, both researchers and practitioners have favored the relative comfort of an absolutist viewpoint in all aspects of development.

In this talk, I will argue the benefits of taking a more thoroughly probabilistic approach in software engineering. Software engineering is rife with stochastic phenomena, and the vast majority of software systems operate in an environment of uncertain, random behavior, which suits an explicit probabilistic characterization. Furthermore, this uncertainty is becoming ever more pronounced in new software systems and platforms, such as the Internet of Things and autonomous vehicles, with their frequent imprecise outputs and heavy reliance on machine learning. To illustrate more deeply some of the considerations involved in taking a probabilistic approach, I will talk about some recent research I have been doing in probabilistic verification.

See you in Singapore!

About Singapore

Singapore is a city-state in Southeast Asia, a global transportation hub and leading commerce and financial center. It is often referred to as the Garden City owing to an extensive greening policy across the island. Its location is one degree north of the equator and it enjoys a tropical climate throughout the year. It is one of the top tourist destinations in Southeast. For details see: [http://www.yoursingapore.com/en.html](http://www.yoursingapore.com/en.html).