THE FUTURE OF ELECTRICITY IS HERE. ARE YOU READY?

PRESENTING THE FIRST OF ITS KIND PLATFORM PRESENTING THE INTELLIGENT ELECTRICITY ECOSYSTEM

The 2015 IEEE-IEEMA INTELECT Conference and Exposition will include the first-ever 3-in-1 global platform that brings together a $10 billion business opportunity across industry verticals. It will feature:

- Interactive Display Pavilions
- A World Class Expo
- Global Conference: Smart Electricity for Emerging Markets

PLUG INTO THE FUTURE OF ELECTRICITY TODAY

www.ii-intelect.org
CALL FOR PAPERS
July 1-5, 2015
Tunghai University
Taichung, Taiwan

COMPSAC 2015 will be organized as a tightly integrated union of several symposia, each of which will be focusing on a particular technical segment. Please visit www.compsac.org for full information on symposia organization.

- Symposium on Embedded & Cyber-Physical Environments
- Symposium on Software Engineering Technologies & Applications
- Symposium on Technologies and Applications of the Internet
- Symposium on Security, Privacy and Trust Computing
- Symposium on Mobile, Wearable and Ubiquitous Computing
- Symposium on Web Technologies & Data Analytics
- Symposium on Human-Machine and Aware Computing
- Symposium on Novel Applications and Technology Advances in Computing
- Symposium on Computer Education and Learning Technologies
- Symposium on IT in Practice

Authors are invited to submit original, unpublished research work and novel computer applications in full-paper format. Simultaneous submission to other publication venues is not permitted. The review and selection process for submissions is designed to identify papers that break new ground and provide substantial support for their results and conclusions as significant contributions to the field. Submissions will be selected that represent a major advancement in the subject of the symposia to which they are submitted. Authors of submissions with a limited contribution or scope may be asked to revise their submissions into a more succinct camera-ready format; e.g., a short paper, workshop paper, fast abstract, or poster.

COMPSAC 2015 will also feature a workshops program for topics closely related to the conference theme, Mobile and Cloud Systems - Challenges and Applications. Special sessions such as Fast Abstract and Industry Papers will be applicable especially for researchers and engineers who would like to present a new, early and work-in-progress ideas, method, and analysis. The Doctoral Symposium will provide a forum for doctoral students to interact with other students, faculty mentors, industry and government. Students will have the opportunity to present and discuss their research goals, methodology, and preliminary results within a constructive and international atmosphere.

Important Dates for Authors:
January 17, 2015: Paper submissions due
March 15, 2015: Paper notifications
April 28, 2015: Camera ready and registration due

Contact COMPSAC
For full information and CFP please visit www.compsac.org
Contact COMPSAC organizers at cs_compsac@iastate.edu

COMPSAC 2015 Local Host:
Tunghai University
Taichung, Taiwan

COMPSAC 2015 Sponsors:
IEEE Computer Society
IEEE Cloud Computing
Korean Institute of Information Scientists and Engineers

COMPSAC Technical Co-Sponsors:
Information Processing Society of Japan

THE WORLD'S NEWSSTAND®
Previous Page | Contents | Zoom in | Zoom out | Front Cover | Search Issue | Next Page

THE WORLD'S NEWSSTAND®
Previous Page | Contents | Zoom in | Zoom out | Front Cover | Search Issue | Next Page

THE WORLD'S NEWSSTAND®
Previous Page | Contents | Zoom in | Zoom out | Front Cover | Search Issue | Next Page

THE WORLD'S NEWSSTAND®
Previous Page | Contents | Zoom in | Zoom out | Front Cover | Search Issue | Next Page

THE WORLD'S NEWSSTAND®
Previous Page | Contents | Zoom in | Zoom out | Front Cover | Search Issue | Next Page
FOCUS

INTERNETWARE AND BEYOND

35 Guest Editors’ Introduction
Software Engineering for Internet Computing: Internetware and Beyond
Antonia Bertolino, M. Brian Blake, Pankaj Mehra, Hong Mei, and Tao Xie

38 Debugging the Internet of Things: The Case of Wireless Sensor Networks
Patrick Eugster, Vinaitheerthan Sundaram, and Xiangyu Zhang

50 Automated Synthesis of Service Choreographies
Marco Autili, Paola Inverardi, and Massimo Tivoli

58 Stigmergy-Based Construction of Internetware Artifacts
Wei Zhang, Haiyan Zhao, Yi Jiang, and Zhi Jin

67 Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications
Yepang Liu, Chang Xu, and Shing-Chi Cheung

76 A Tail-Tolerant Cloud API Wrapper
Qinghua Lu, Xiwei Xu, Len Bass, Liming Zhu, and Weishan Zhang

83 Multitier Diversification in Web-Based Software Applications
Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire

91 Roundtable
The Future of Software Engineering for Internet Computing
Jian Lü, David S. Rosenblum, Tevﬁk Bultan, Valerie Issarny, Schahram Dustdar, Margaret-Anne Storey, and Dongmei Zhang

FEATURE

98 The Road Ahead for Architectural Languages
Patricia Lago, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Antony Tang

See www.computer.org/software-multimedia for multimedia content related to the features in this issue.
DEPARTMENTS

4 From the Editor
Fewer Dead Trees, More Engagement
Diomidis Spinellis

11 On Computing
Of Boilers, Bit, and Bots
Grady Booch

14 Reliable Code
To Code Is Human
Gerard J. Holzmann

18 Requirements
Toward Meaningful Industrial–Academic Partnerships
Jane Cleland-Huang

22 Software Technology
What Next? Advances in Software-Driven Industries
Christof Ebert, Gerd Hoefner, and Mani V.S.

29 Impact
Mobile Money’s Impact on Tanzanian Agriculture
Balachandran Seetharam and Drew Johnson

106 Voice of Evidence
How Do I Know Whether to Trust a Research Result?
Martin Shepperd

110 The Pragmatic Architect
The Five Properties of Successful Architectural Oversight
Nick Rozanski

116 Software Engineering
Microservices
Johannes Thönes

MISCELLANEOUS

5 How to Reach Us
8 2014 Reviewers
82 IEEE Computer Society Information
90 Advertiser Information

www.computer.org/software
Fewer Dead Trees, More Engagement

Diomidis Spinellis

**Editor in Chief:** Diomidis Spinellis
*Athens University of Economics and Business, dds@computer.org*

**FROM THE EDITOR**

**Switch to Digital**

In 2013 it became evident that our readers were switching en masse from print subscriptions to digital delivery. In response to that trend and to take advantage of digital delivery options while saving paper, printing, and distribution costs, IEEE Computer Society approved the move to digital as the primary delivery mechanism starting in 2015. This change is now reflected in our membership renewal forms, which list digital delivery as the default; the print issue is available only at a considerable additional cost. Also, through the new “Software and Systems” Computer Society membership option (www.computer.org/portal/web/membership/Software-and-Systems-Join), you can get a digital subscription to Software (and 12 more Digital Library articles) without paying extra.

Don’t let the two nonpaper versions of the magazines that the Computer Society now offers confuse you. The “digital” format, which is the complete magazine, is available to individual members through an app or as one PDF download. The “electronic” format, which consists of either the entire issue or individual articles in PDF format, is distributed through the Computer Society Digital (yes) Library. I understand that the app is available only to individual subscribers, not to members of institutions that subscribe to the Digital Library.

I must admit that, initially, I was skeptical when I heard about the switch to digital. I love printed paper, and I have been slow to adjust to the brave new world of e-reading. I like looking at the books lining my bookshelves, I enjoy the speed with which I can flip a magazine’s pages, I value the serendipity brought by grazing printed material, and I have fond memories of browsing through library stacks. (In the past, when I visited a university, I made a point to spend some time in its library, photocopy articles, and look for new and interesting stuff. I can still remember the neatly bound, typewritten volumes of a USSR journal, which was translated into English at the height of the Cold War, that I once chanced upon at the University of British Columbia library.) Also, I find it difficult to read articles on my computer screen, and I can’t stop worrying I might one day lose digital access to the material I enjoy.

However, what happens in practice differs a lot from the romanticized picture I’ve presented. My desk is literally crumpling under the load of three stacks of partly read maga-
azines. Having run out of shelf space, I stack old books behind the new ones, and I’m looking for three more shelf-meters (10 ft.) to organize the eyesore on my desk. When I visit a library, I look around with an empty feeling, which sinks in when I reflect that I can find vastly more, and more current, material on the Internet. I fall behind in reading the beloved paper copies on my desk, probably because I spend excessive time reading stuff on the Web. And magazines often arrive weeks late at my home address and then weigh down my backpack when I take them on the road.

I therefore decided to experiment with Software’s digital format and was pleasantly surprised. I read the magazine through its Qmags app on an iPad and an $80 no-frills Android tablet, and as a PDF on my laptop. All worked well. Some problems mentioned in reviews of the Android app seemed to have been addressed in the version I downloaded. The text was crisp (even on the Android tablet’s lackluster screen), the hyperlinks from the table of contents navigated fine, and I could zoom in to read small figures. The PDF wasn’t protected by any insidious digital-rights-management scheme, which means I can archive it to ensure its long-term availability. (My insecurity regarding availability is probably unfounded because over the past 15 years, the material I can access online has always been increasing.)

For me, the advantages of the switch to digital were eye-opening. First, I can say goodbye to the accumulating stacks of magazines on my desk and the space required to shelf them: out of sight, out of mind. Then, with a tablet at hand, I can now tap to read Software articles—stuff that really matters, instead of another rehash of the day’s events on the Web. Surprisingly, I found I can browse faster through a magazine’s online version than through the paper one. So, I’ll go over a lot more material. Finally, I can have all the articles the day they appear, wherever I am, and carry them always with me.

**Future Perfect**

The switch to digital can also enable us to do many more things in the future. Currently, magazines are delivered in batch as a facsimile of what was the print edition. Instead, we could deliver them in a flowed-text format, which allows nice zooming and is more suitable for reading online on diverse devices. We can also switch to gradual delivery of the articles, keeping our community more engaged through comments and “likes.” Adopting the existing tablet newsstand applications will let readers view all our magazines through the same interface. With digital delivery’s low marginal cost, we can easily experiment with alternative pricing models, such as limited-time free introductory subscriptions, affordable student offers, and free access to a specific number of articles. At the same time, we must find a way to provide digital (rather than electronic) access to all institutional-subscriber members.

Getting enhanced (anonymized) analytics from the magazine’s digital readers can help us guide the editorial content toward what our community actually reads. We can gain valuable insight from knowing the articles being read, the engagement per article, and the engagement of subscriber groups. More important, given that paper, binding, and weight no longer constrain the magazine, we could more easily increase our page count and issue frequency.
Crowdsourced editing and composition might help in this direction.

**Building a Community of Software Practitioners**
You might have noticed the new editor’s name on this column’s masthead. The strength of *Software* is its community of practitioners and researchers. The editorial and advisory boards and the editor in chief are only helping the community express its ideas. Having said that, I’m excited to be taking over the magazine’s helm from this issue onward. I’m extremely grateful to Forrest Shull for the terrific state in which he’s leaving the magazine, and to our two boards for their contagious enthusiasm, amazing hard work, inspirational ideas, and indispensable level-headedness. *Software* enjoys excellent health. It’s the most popular choice for a Computer Society magazine and generates about 180,000 digital article downloads per year. The magazine’s people-ware is its core asset: the strong and widely respected editorial and advisory boards, the keen Software Engineering Radio team churning out podcasts with more than 40 thousand downloads per episode, and its supportive reader community.

Yet there are challenges in staying true to our mission, “building a community of leading software practitioners,” because the Internet is raising the bar on how technical communities are served. The emergence of technology blogs, Q&A sites, massive open online courses, open-access publishing, and career-oriented social networking sites means that yesterday’s offerings no longer cut it. We must adapt or risk becoming irrelevant.

My vision for *Software* to be the leading voice for software engineers, competing in authoritativeness, accessibility, field reach, and engagement with publications such as *Nature, Science*, the *Harvard Business Review*, the *Economist*, and the *MIT Sloan Management Review*. This will be a tough nut to crack, but there’s already a plan in place involving the department columns and regular articles and the magazine’s design, online presence, promotion, and governance.

On the department front, key ideas include a column offering advice on software engineering topics, a department that will host our field’s experts and luminaries, summaries of important conferences, and a retrospective of influential articles. We can also enhance our regular articles by having important conference or journal papers rewritten for practitioners, commissioned articles on hot topics, short articles by practitioners for practitioners, and a service that pairs developers with an interesting story with academics who want to help its presentation.

We can do a lot to enhance our online presence: publish regular social network updates, run online debates, host author-provided infographics and rich presentations, establish collaborations with online communities, run blogs, and fine-tune our digital delivery. Initiating a collaboration with existing online communities, such as those behind GitHub and the StackExchange sites (Stack Overflow, Programmers, Software Quality Assurance & Testing, Code Review, and so on) can be a win-win situation in which *Software* can act as a neutral hub.

Sadly, no matter how attractive our content becomes, the mountain won’t come to Muhammad, so we must work harder on promotion, visibility, and marketing. Our goals here include expanding the awards program, publishing translated summary pages, employing mailing lists, publishing a newsletter, and partnering with initiatives targeting software developers, such as the Hour of Code and the Google Summer of Code. We can also explore cross-selling opportunities with software vendors and leveraging our multimedia offerings.

To deliver on all these fronts, we must change how the magazine is run. The most important factor of *Software*’s success is its volunteer board members. Through larger and wider volunteer participation in our boards, we can establish the bandwidth required to deliver the changes I outlined, and more. This will entail appointing associate editors in chief for new editorial content domains (for example, conference reports and articles), and advisory board members for engaging with our communities.

I’m certain that by issuing open calls for new board members, many of you will step in to help. Through a position on one of the *Software* boards, you can serve your profession, interact closely with leaders in our field, affect the field’s direction, and gain additional recognition and visibility (if you haven’t more than enough already). You can apply for the available board positions at [http://goo.gl/forms/piw5yF6H1X](http://goo.gl/forms/piw5yF6H1X). Although the form will remain open until the end of January 2015, please visit the form now, and volunteer generously to become part of the team that will create the new *Software*.
REVIEWER THANKS

2014 Reviewers

The articles appearing in IEEE Software are the result of hard work by many people. We deeply appreciate the efforts of everyone who reviewed over 200 articles submitted to Software last year. The peer review process helps maintain the magazine’s revered quality. All of us in the software development community owe gratitude to the people who participate in this crucial service. Readers who would like to contribute as reviewers can visit www.computer.org/software/reviewers to find out how they can get involved. —Software’s editorial board and staff

Ivan Aaen
Alain Abran
Bram Adams
Gojko Adzic
Osama Al-Baik
Mohamed Almorsy
Pierre America
Apostolos Ampatzoglou
Chintan Amrit
Saswat Anand
Samuil Angelov
Giuliano Antoniol
Brad Appleton
Alain April
Mark Ardis
Luca Ardito
Phil Armour
Martin Ashton
Ali Ataya
George Athanasopoulos
Chris Atlee
Anna Averbakh
Jeffry Babb
Jon Bach
Ebrahim Bagheri
Hamid Bagheri
Xiaoying Bai
Jim Baker
Luciano Baresi
David Barnes
Scott Barnett
Edd Barrett
Olga Baysal
Steffen Becker
Earl Beede
Andrew Begel
Fevzi Belli
Stephany Bellomo
Fouad ben Nasr Omri
Brian Berenbach
Egon Berghout
Daniel Berry
Stefanie Betz
Domenico Bianculli
Stamatia Bibi
M. Brian Blake
Kelly Blincoe
Robert Boccino Jr.
Michael Bolton
Carl Friedrich Bolz
Daniele Bonetta
Matthias Book
Eric Bouwers
Ivar Bratberg
Kevin Brown
Stefan Brunthaler
Dennis Brylow
Luigi Buglione
Bora Caglayan
Yuanfang Cai
David Callele
Russell Campbell
Rafael Capilla
Marcelo Cataldo
W.K. Chan
Anis Charfi
Robert Chatley
Stephen Chenoweth
Jürgen Cito
Gilles Civario
James Clause
Annie Combelles
Tayana Conte
Oliver Creighton
Daniela Soares Cruzes
Bojan Cukic
Didem Culha
Xiaoling Dai
Daniela Damian
Lars-Ola Damm
Maya Daneva
Eric Dashofy
Boris Debic
Berend de Boer
Jose Luis de la Vara
Serge Demeyer
Rodrigo de Toledo
Lukas Diekmann
Torger Dingsöyr
Massimiliano Di Penta
Joerg Doerr
John Dooley
Juan Duenas
Tore Dybå
Cyntrica Eaton
Christof Ebert
Jutta Eckstein
Jonathan Edwards
Vyacheslav Egorov
Veli-Pekka Eloranta
Neil Ernst
Tor Fægri
Davide Falessi
Guisheng Fan
Xianwen Fang
John Favaro
Michael Felderer
Ludger Fiege
Fabien Fleutot
Marios Fokaefs
Xavier Franch
Akos Frohner
Ilenia Fronza
Qi Gao
Kevin Gary
Massimo Gervasini
Arbi Ghazarian
Robert Glass
Karl Goeschka
Jorge Gómez
Alessandra Gorla
Sarah Gregory
Greg Grossmeier
David Grove
Sujay Gupta
William Halfond
Geir Kjetil Hanssen
Zhong Hao
Ryan Hardt
Ahmed Hassan
Orit Hazan
Peter Henderson
Stephan Herbut
Thomas Herrmann
Dirk Heuseroth
Robert Hierons
Mike Hinchev
Reid Holmes
Leonard Hoon
Joost Houwen
Changjin Huang
Jiwei Huang
Harvey Hyman
Shams Imam
Christian Inzinger
Tuomas Jaanu
Slinger Jansen
Zhen Ming Jiang
Qun Jin
Andrew Johnston
Vikrant Kaulgud
Rick Kazman
Noureddine Kerzazi
Petri Kettunen
Atabak Kheirkhah
Foutse Khomh
Seon Wook Kim
Pat Kirwan
Eric Knauss
Dwayne Knirk
Stefan Koch
Carsten Kolassa
Dimitris Kolovos
Philippe Kruchten
Marco Kuhrmann
Craig Kuziemsky
Jo Ann Lane
Claire Le Goues
Timo Lehtinen
Benjamin Lerner
Timothy Lethbridge
Jean-Louis Letouzey
Grace Lewis
Juho Lindman
Mikael Lindvall
Lin Liu
Xuanzhe Liu
Yu Liu
David Lo
Wolfgang Lohmann
Cauahtemoc Lopez-Martin
Jian-Guang Lou
Jianguo Lu
Mircea Lungu
Etuo Luoma
James Lyndsay
Neil Maiden
Somayeh Malakuti
Ivano Malavolta
Tomí Männistö
Mika Mäntylä
David Maplesden
Edu Marchetti
Sabrina Marczak
Hidehiko Masuhara
Florian Matthes
Stephen McCamant
Shane McIntosh
Collin McMillan
Nancy Mead
Grigori Melnik
Ettore Merlo
Harald Merten
Marija Mikic-Rakic
Tommi Mikkonen
Tim Miller
Raffaela Mirandola
Andriy Miransky
Dustin Mitchell
Audris Mockus
Nils Moe
Kim Moir
Martin Monperrus
Michal Moskal
Henry Muccini
Hausi Muller
Neel Mullick
Jürgen Münch
Ethan Munson
Talfile Murnane
Brendan Murphy
Sarah Nadi
Meiyappan Nagappan
Iulian Neamtiu
Jason Newblanc
Chris Newburn
Tien Nguyen
Tung Nguyen
Marc Novakouski
Jukka Nurminen
Nate Nystrom
Helena Olsson
Zachary Oster
Meriem Ouederni
Ipek Ozkaya
Maria Paasivaara
Barbara Paech
Krzysztof Palacz
Neeraj Parolia
Pedro Pastor
Nattakarn Phaphoom
Matthew Plavcan
Robert Plummer
Andrea Polini
Lori Pollock
Macario Polo
German Poo
Eltjo Poort
Karl Popp
Christopher Preschern
Davy Preuveneers
Rafael Prikladnicki
Giuseppe Procaccianti
Cynthia Putnam
Alcides Quispe
Hridesh Rajan
Balasubramaniam Ramesh
Annette Reilly
Patrick Rempel
Ita Richardson
Jan Richling
Dirk Riehle
Stan Rifkin
Peter Rigby
Armin Rigo
Martin Robillard
Pilar Rodriguez
Elco Rommes
Tiark Rompf
Marc Roper
Louis Rose
Chuck Rossi
Romain Rouvoy
Guenther Ruhe
Sukyoung Ryu
Mehrdad Sabetzadeh
Iman Saleh Moustafa
Norsaremah Salleh
Raul Santelices
Ganesh Santhanam
Walt Scacchi
Giuseppe Scanniello
Josef Schieber
Thomas Schilling
Christoph Schmidt
Jean-Guy Schneider
Kurt Schneider
Carolyn Seaman
Lionel Seinturier
Alper Sen
Alexander Serebrenik
Basma Shahadat

JANUARY/FEbruary 2015 | IEEE SOFTWARE | 9
Watts S. Humphrey Software Process Achievement Award

Nomination Deadline: January 15, 2015

Do you know a person or team that deserves recognition for their process improvement activities?

The IEEE Computer Society/Software Engineering Institute Watts S. Humphrey Software Process Achievement Award is presented to recognize outstanding achievements in improving the ability of a target organization to create and evolve software.

The award may be presented to an individual or a group, and the achievements can be the result of any type of process improvement activity.

To nominate an individual or group for a Humphrey SPA Award, please visit http://www.computer.org/portal/web/awards/spa
Of Boilers, Bit, and Bots

Grady Booch

ONCE UPON A TIME, boilers used to blow up quite often, killing many people. This was generally not seen as a good thing.

Despite the comforts that boilers brought to life—trains and steamships that could transport people and goods faster than a horse, central heating for buildings previously warmed by individual fireplaces, power for innumerable manufacturing industries from milling to weaving to machining—the social transformation and human cost couldn’t be neglected. In the second half of the 1800s, professional engineering organizations formed, best practices were codified into law, and the ethos of living with these noisy, smelly, and sometimes explosive boilers slowly entered the daily life of everyone dwelling in an industrialized city. It took much longer to regulate the pollution associated with coal-burning boilers, but that too mostly came back into balance (only to be subsumed by other kinds of energy-related pollution).

But, by going back in time, I get ahead of myself. Let’s begin again with three contemporary stories.

On Privacy, Regulation, and Encryption

There’s a site that lets you watch tens of thousands of unsecured Internet connected cameras (www.insecam.com). The public outrage over this site has been rather vociferous, especially by those who have found their home or office on the list. Their reaction is quite understandable. For the most part, we all have some expectation of privacy in our personal spaces, and we rightfully assume that what happens behind closed doors will remain hidden from public view.

On the other hand, the reaction by the technical community has been far less sympathetic. As many have noted, putting an unsecured IP camera on the Internet is akin to placing your home or office on a busy public street, installing large glass windows, and leaving the drapes open. Of course—as the digerati point out—you have abrogated your right to privacy by being so careless.

Amazon just announced the Echo, an intentionally unobtrusive physical artifact that embodies Alexa, an ever-present digital personal assistant. Alexa joins a growing family of personal digital assistants, including Apple’s Siri, Microsoft’s Cortana, and Google’s Now.

We aren’t by any means even close to a really personal digital assistant such as Samantha, as found in the universe of Spike Jonze’s Her, but it’s not hard to trace the trajectory of such technology. Human as we are, we often desire an uncomplicated and compliant companion to...
assist us with the tedium and cruft of life. Even today, the wealthy and the famous might have the means to hire a flesh-and-blood personal assistant; technologies such as Alexa, Siri, Cortana, and Now bring some of that ease to the masses.

That being said, there are certainly issues worth debating publicly. As one observer wryly noted regarding Echo’s product features, the “NSA, CIA, and FBI would like to personally thank Amazon for installing spycams in every home” (http://techcrunch.com/2014/11/06/amazon-echo/#comments). While I don’t exactly share that level of cynicism or paranoia, the observer does make a valid point: In a digital world, what’s the meaning of trust?

There are any number of legal battles going on whose front lines are at the confluence of the digital and the social. The FCC is at the center of the debate about network neutrality. Many ISPs are against it, driven by economic needs that prefer an unregulated Internet (just as long as the playing field still tilts unnoticeably in their direction), whereas many of those representing consumer rights are against it. FBI Director James Comey has come out against the efforts by Apple and Google to encrypt their phones in ways that increase consumer privacy at the expense of law enforcement having easy access the data behind such encryption. Taylor Swift’s latest album 1989 went platinum—the first such album of 2014, in a year in which music sales were deeply reduced—in spite of explicitly withdrawing her tracks from the digital streaming service Spotify.

What can we make of these contemporary stories?

Now and Then
Let’s return to the age of boilers, the age of the great Industrial Revolution that transformed nations and lives. This was also the age of Charles Babbage and Charles Dickens (who, by the way, died only about a year apart). In his marvelous book Charles Dickens in Cyberspace (Oxford Univ. Press, 2006), Jay Clayton speaks of the societal transformation that took place in that era, a time that I also spoke of in my earlier column, “To Code or Not to Code” (IEEE Software, vol. 31, no. 5, 2014, pp. 9–11). Referring to Jane Austen’s novel Mansfield Park, Clayton observes that part of the plot hinges on “the consequence of complex interactions among varied communications media”—in short, the juxtaposition of the pen and paper against the telegraph. Clayton notes, “we live in an undisciplined culture again,” so our challenges in adapting to a digital culture are born out of the same human dynamics that people in the time of Babbage and Dickens faced in adapting to an industrialized culture.

Boilers used to blow up, but science and industry were compelled to mitigate the underlying causes, and society and the law eventually metabolized their use. Cameras, digital assistants, and encryption sometimes yield terrible results, yet the art and science of computing is similarly compelled to mitigate their negative aspects. Furthermore, we have a generation being born who doesn’t know that these things once didn’t exist.

Let’s return to each of my contemporary stories one last time.

Evolving Doors
The problem with unprotected cameras is just one visible manifestation of a much larger issue: the growth of the Internet of Things. Who in his or her right mind would have imagined a world in which doorknobs and light bulbs would have unique IP addresses, much less each of us carrying around mobile network device (our phones) and driving around in our cars (which themselves are networked)? Well, actually, a few prescient folks did predict such a thing, but the basic undercarriage of the original Internet was never crafted for such circumstances (thus the recent move from the IPV4 address to IPV6).

In the early days of the Industrial Revolution, who would have thought of putting a boiler in every home? Well, we do now, in the form of water heaters. These things are still capable of explosion and damage, but we’ve largely engineered the risk out of them. Also, their user interface is so simple that we mostly don’t even think about them, they have so faded so completely out of sight.

So it will be with these cameras and other connected devices, but with one big difference. By their nature, they’re connected in ways beyond our choice—their meaning is materially different from the water heaters we place in our homes. How then do we cope? The general public must adjust; they must accept the additional risk that these things might blow up in unintended digital ways.

At the same time, we who develop, deploy, and deliver such technology must engineer the risk out of them and make them so simple we don’t even think about them. As we engineers know, making things simple is terribly hard, but the human use of our software-intensive artifacts demands it.

The rise of cognitive assistants brings us to exactly the same perspective. Whereas the Internet of Things connects us to one another, Siri, Cor-
tana, Now, and Alexa have the potential to give us something much more intimate: ways to connect to ourselves, to reduce the friction of life. We engineers still must drive the risk out of these artifacts and make them part of the atmosphere. This too is very hard, made even more so because we’re talking about software-intensive systems that can become extremely personal. Not only do we need good software engineers and cognitive scientists building such things, we also need psychologists, cultural anthropologists, and artists to be a part of the journey.

As for the vibrant legal morass that we now see at the intersection of computing and humanity, let me offer the positive spin on it that the presence of such noise might actually be a good thing. Science and technology always precede the rule of law. The fact that we see public debate suggests that the time has finally come for society to codify best practices into law, just as in the age of boilers. The thing we nonlawyer software types must get used to is that the business of making laws is incredibly messy, often muddled and inconsistent, and rarely ever stable or settled. To that end, as computer scientists we must both accept the rule of law and not be silent in its making.

The story is often told of Steve Job’s maniacal focus to shave milliseconds from seemingly trivial parts of the Mac OS. Under his point of view, even milliseconds—when multiplied by trillions of times of execution—add up to human cost. Early boilers were incredibly sloppy and wasteful, but we engineered efficiencies into them. Most of our Web-centric systems are equally sloppy and wasteful, but because we live in an abundance of computational resources, the energy costs of a simple RESTful interaction are often ignored. And yet, when multiplied by trillions of times of execution, they do add up to cycles spent on a server, electricity consumed, and some fuel—fossil, hydroelectric, nuclear, or solar—expended.

We, who have the privilege to develop, deploy, and deliver software-intensive systems that matter, shouldn’t expect the general public to grok the intimate technology with which we live and breathe and in which we delight. We must design these artifacts as if our children’s lives depended on them (and, in many cases, they do). We’re the builders of these modern-day digital boilers that entertain us, feed us, care for us, and serve us. It’s our responsibility to ensure that we do our work with the utmost professionalism and care for the human spirit, to which our work is ultimately directed.

GRADY BOOCH is an IBM Fellow and one of UML’s original authors. He’s currently developing Computing: The Human Experience, a major transmedia project for public broadcast. Contact him at grady@computingthehumanexperience.com.
When I first learned to code, in what now seems more like the dark ages, the first step in program development was to take a coding sheet and write out the program’s design. Then, someone had to sign off on that design, to make sure it wasn’t too obviously wrong. Next, I had to wait my turn at a punch-card machine to type the program line by line onto punch cards (see Figure 1). When that was complete, I placed the card deck in a little bin with my name on it. In due time, a computer operator would pick up the deck and feed it into a large mainframe computer. The computer center proudly offered a rapid 24-hour turnaround on jobs. So, the very next day I could collect the printed output and discover all the things that had gone wrong. This process then had to be repeated, day after day, until the program was finally right.

As painful as this process might seem today, it taught me to be very careful with my code. Just getting the program to execute was a time-consuming battle with syntax errors or with mistakes in the also-required “job control” cards that told the behemoth machine how to execute a program. And then the real frustration would begin. No matter how certain I was that my program was correct, the machine would manage to find something to complain about. The delays were never the machine’s fault. It was just very good at confronting me with my own fallibility.

How are things different now? Thank goodness, the long wait to find out how we’re messing things up is gone, now that we have more compute power in our watch than a roomful of hardware in the dark ages. But we’re still human, and we still make mistakes. We can just make and discover our mistakes much faster than before.

Most compilers have built-in checks that warn against common types of coding flaws. Static source code analysis tools also excel at finding bugs and revealing risky coding patterns. Modern coding standards similarly include a lot of rules that

Figure 1. An 80-column punch card, from the days when writing a program took time—a lot of time.
can prevent common mistakes. But now that our tools have gotten so efficient in pointing out where we’re taking unnecessary risks, we come up against some other very human characteristics: programmers, and perhaps humans in general, don’t like following rules. In fact, we take pride in finding creative ways around them.

Dodging rules, or finding clever ways around them, happens in every field. For instance, in 2013, the National Hockey League forbade players from taking off their helmets when a fight breaks out during a game. So, players simply took off each other’s helmets before enthusiastically engaging in a fight. Similarly, after a town issued an ordinance forbidding vendors to sell drinks during street festivals, a quick-thinking vendor figured out that he could still sell a peanut for a dollar and offer a free bottle of water with each purchase. Even the US Congress isn’t immune to this type of creative subversion. In 2011, it declared that pizza could count as a vegetable in school lunches, thus letting students get around the stricter rules for lunches that were about to go into effect. We all love to outsmart the system, and that’s just as true when it comes to writing code.

### Outsmarting the System

Near the top of the list of the most common types of mistakes programmers make is the unintentional use of uninitialized data. In C, if you declare a global variable and forget to initialize it, its initial value is guaranteed to be zero. That might be the right value, but the compiler can’t really tell, so it won’t issue a warning. If, however, you omit the initialization in the declaration of a function local variable, your variable’s value will be undefined. It will be whatever happens to be in memory at the stack location where the automatic variable is stored. If the first access to that variable tries to retrieve its value, the result will likely be wrong (see Figure 2). A good compiler will therefore issue a warning for such cases.

The root cause for the accidental use of an uninitialized variable is often the existence of an execution path the developer didn’t realize existed. To fix it generally requires rewriting the algorithm to ensure the unsuspected execution path is safe. That requires thought.

But there’s also another, quicker way to remove the warning. You can explicitly initialize the local variable to any value at all in its declaration, and the warning will go away. The code, though, isn’t necessarily any better than it was before, and it might actually be worse. You’ve only succeeded in hiding it better from the tools that are trying to protect you from yourself. In Figure 2, the result of initializing the local variable \( x \) to the value 1 would be a run-time error when you later try to divide the sum of \( x \) and \( y \) by their difference.

### The Gentle Art of Sidestepping

There is, of course, no shortage of examples of programmers sidestepping rules and of the risks this can create. For instance, safety-critical applications often have a rule limiting the use of conditional compilation directives. The one exception that’s generally allowed is to use a conditional directive to prevent the repeated processing of header files. The reason for frowning on conditional compilation directives is that their unrestricted use can dramatically increase the number of ways the code can be compiled and thus the number of ways it should be tested. With just 10 such directives, the code can be compiled in up to \( 2^{10} \) (1,024) ways, but it would most likely be tested thoroughly for only one. This leaves all other possible variants at risk, including the one that might be called into action the very last moment before a public release.

Finding violations of this coding rule isn’t hard: a scan of the source files with a simple script that looks for the telltale `#ifdef` and `#ifndef` keywords suffices. But it’s easy to defeat this type of check if the resulting warnings become too annoying. For instance, a developer can first define a macro that’s statically defined to be either true or false. Then, he or she can employ a Boolean test of that value in a standard `if-then-else` construct in the code. This achieves the same effect of the conditional compilation directive, but with the benefit that the checker can’t detect it. But the vulnerability is still there.

Another trick is remarkably popular to defeat compiler warnings about unused parameters to
functions. You could define a parameter that’s meant to be used when the code for a function is complete, but that isn’t used when earlier versions of the code are tested. It isn’t too hard to fool the compiler into thinking that the parameter is used, to avoid those pesky warnings that the compiler otherwise generates.

Figure 3 shows a common method in which the programmer simply adds a line of code that evaluates the parameter’s value and then discards it explicitly with a cast to (void). Most compilers accept this quietly and don’t generate code for the added statement, and you score a quiet victory over the machine. The danger is that you might forget to add the originally intended code that properly uses the parameter, because all now seems to be well with the function as written.

**Manipulating Metrics**

According to Goodhart’s law (named after the economist Charles Goodhart), “When a measure becomes a target, it ceases to be a good measure.” Code metrics are particularly easy to manipulate in your favor. Consider, for instance, what would happen if you reward software developers solely on the number of lines of code they write per day. “Productivity” will likely increase dramatically in a short amount of time. Similarly, if you evaluate a test suite only on the basis of a blind measurement of code coverage, you’re likely to inspire the creation of test suites that excel at executing lines of code, without actually testing them.

Another metric that program managers like is comment density. Large projects often require every function to have a comment header based on a predefined template that includes entries for defining the parameters to the function and the return value it’s meant to compute. More often than not, though, the template is just copied into the code unchanged. This satisfies the formal rule that the template is present but doesn’t actually provide any useful information about the function’s purpose or use. And who hasn’t seen those eminently superfluous comments of the type “assign a new value to x” or “subtract one from the value of y.” It’s just like the kids in high school when they figure out that using words such as “magnificent” or “inordinate” increases the writing score they get from automatic grading tools in their English class. Before you know it, just about everything becomes “inordinately magnificent.”

Finding creative ways around rules is almost an art. But the tools we use to check those rules can sometimes also play games with us and trip us up when we least expect it.

**Ducking Compliance**

Figure 4 shows a remarkable piece of code that illustrates what can happen if you move perilously beyond a language definition’s boundaries. (Xi Wang and his colleagues described a similar example in 2013.1) The C language standard explicitly enumerates all implementation-defined and formally undefined language features. Nonetheless, exploiting some of those features is for some the ulti-
mate test of cool. Be warned—if you do so, you’ll be at the mercy of the authors of the specific version of the compiler you’re using, running on a specific hardware platform.

If you compile and execute the code from Figure 4 without optimization, the result will be \( x < 0 \) and \( -x < 0 \), which says that the integer variable \( x \) is negative and the negation of \( x \) is also negative. That would surprise most math majors, but it’s the result of having initialized \( x \) to the minimum value that can be represented in an integer.

So, what’s going on? For 32-bit signed integers, the value of \( \text{INT\_MIN} \) is \(-2^{31}\) (\(-2,147,483,648\)). The two’s-complement binary representation encodes this as a single one followed by 31 zeros. On the other end of the spectrum, the maximum value representable in a signed integer is \( 2^{31}-1 \) (\(2,147,483,647\)), which is encoded as a single zero followed by 31 ones. If you try to take the negation of \( \text{INT\_MIN} \), like the code in Figure 4 does, you might expect to get \(2,147,483,648\), but that’s one more than \( \text{INT\_MAX} \). And, through the wonders of the two’s complement, one more than \( \text{INT\_MAX} \) gets the same encoding as \( \text{INT\_MIN} \). So, in two’s complement, the negation of \( \text{INT\_MIN} \) is equal to \( \text{INT\_MIN} \) itself, which explains how the program executes. But does it?

If you compile the code again, but this time enable compiler optimization (for instance, using gcc compiler version 4.6.3), the printed result can change to “everything is OK.”

What gives? Clearly, the program didn’t change, and most likely neither did the representation of integer numbers that the compiler used for storing integer numbers. This is where the smarts of a compiler writer can outsmart those of the programmer. The optimizer can reason that because a true integer number can’t be both negative and positive, it can replace the conditional \( (x < 0 \&\& -x < 0) \) with \( \text{false} \). This optimization forces the program’s execution into the \text{else} branch and gives a new result. Whether or not you use optimization, the gcc compiler issues no warnings, so you aren’t alerted to the code’s hidden ambiguity.

The C language standard warns clearly about the use of integer values that can’t be represented in the available bit-width of the machine, noting that the result of such operations is implementation-defined. This means that not every compiler will necessarily produce the same result and that even a single compiler can validly produce different results depending on how the code is compiled. When something is implementation-defined, it’s simply up to the compiler writer to decide what to do, and, indeed, compiler writers are humans too.

Writing reliable code is hard. What’s so interesting about software development is that even the smallest mistake can have the largest consequences. As a comedian once said, “When you dial a wrong number, and you’re off by only one digit, I don’t think you should get a whole different person!” But this is precisely what can happen in software systems. A single misplaced break statement can paralyze the entire long-distance calling network, as it did in the US on 15 January 1990. A single uninitialized variable can make a spacecraft crash, as it did on 3 December 1999 when the Mars Polar Lander failed its landing.

What’s so fascinating is that these types of failure seem completely out of proportion with the flaws that caused them. This is also what makes designing and building reliable software systems so different from designing and building reliable hardware. The best remedy? Keep your code clean and simple and heed the warnings from the tools that are designed to protect you. Either do that or risk having a dumb machine show you it can continue to outsmart you.

References

Gerald J. Holzmenn works at the Jet Propulsion Laboratory (JPL) on developing stronger methods for software analysis, code review, and testing. He’s an ACM and a JPL fellow and a member of the National Academy of Engineering. Contact him at gholzmenn@acm.org.
Toward Meaningful Industrial–Academic Partnerships

Jane Cleland-Huang

OVER THE PAST few years, there has been much discussion in the requirements engineering (RE) research community about researchers’ responsibility to deliver practical, usable solutions to industry. Practitioners complain that researchers often work on theoretical problems that never seem to see the light of day, while the day-to-day problems remain unsolved. Nevertheless, many researchers collaborate closely with industrial partners and care deeply about technology transfer. Unfortunately, we face many hurdles in moving a viable research idea from inception to deployment.

To encourage and foster technology transfer in the RE community, Daniela Damian and I initiated Ready-Set-Transfer, a game-show-style panel, at the 2011 IEEE International Requirements Engineering Conference. We designed it to encourage an honest exchange of ideas between researchers and practitioners. Instead of just talking about technology transfer, researchers present their ideas to a team of industrial judges, explaining the industrial motivation and describing how they’ve evaluated their idea’s adoption readiness.

We’ve been running Ready-Set-Transfer each year since 2011. During that time, several interesting projects have emerged, representing ideas as varied as crowdsourcing, adaptive privacy, and traceability of safety-critical systems. This year, Jane Huffman Hayes and Didar Zowghi hosted the panel.1 The three research projects they selected represented solutions from the diverse areas of requirements elicitation, modeling, and traceability. You can decide for yourself whether these solutions are successful in terms of technology transfer. Each one is at a different level of adoption readiness and carries a different degree of risk. I hope that talking about them will help us start thinking seriously about the issues, challenges, and responsibilities of technology transfer in RE.

This year, our panelists included Mike Panis from Teredyne (US), Juha Savolainen from Danfoss Power Electronics (Denmark), and Erik Bjernulf from Tolpagorni Product Management AB (Sweden) (see Figure 1). As you’ll see, they made insightful comments to the contestants. Now, let’s take a quick look at the three projects.

FlexiSketch
FlexiSketch (see Figures 2a and 3) is a unique collaborative sketching tool that University of Zurich researchers developed.2 FlexiSketch runs across multiple tablets mimicking a distributed whiteboard. It supports free-form drawings and arbitrary node-and-edge diagrams. In addition, it aims to fill the gap between formal modeling and free-form drawing tools. Users can incrementally transform informal whiteboard sketches into formal models by retroactively creating new types using
a lightweight metamodeling feature. For example, they can select a shape on the whiteboard and promote it to a formal, reusable type in the metamodel. So, FlexiSketch supports a seamless transition from sketching to modeling. Whiteboard sketches become more than pretty pictures; users can evolve them into semiformal models that they can open in modeling-tool environments.

The FlexiSketch team have made a great effort to overcome the technology transfer hurdle, including uploading an early version to Google Play and encouraging requirements engineers from various companies to assist them in the early prototyping. They’ve also presented FlexiSketch at conferences attended by potential industrial adopters. Finally, they conducted three in-depth workshops at select companies to evaluate their approach and investigate people’s sketching and notation-defining behavior when using FlexiSketch.

University of Zurich researcher Dustin Wüest said this experience has taught them that platforms and technologies matter. Several companies were willing to try FlexiSketch only if an iOS version was available. Dustin also pointed out that research and industry needs often diverge. Whereas researchers tend to focus on proposing and evaluating novel ideas and concepts, industrial adopters want rich, stable feature sets, which can be costly to deliver. In addition, Dustin mentioned that taking a research idea to market requires a team effort that involves not just scientific discovery but also outreach and marketing skills.

The industrial panel asked how the team could be certain that early adopters were using the novel features of the semiformal metamodeling and not just adopting FlexiSketch for its basic collaborative-whiteboard features. The team answered that they already had been addressing this through the face-to-face industrial workshops they had conducted, which gave them deep insights into the features people were using. Indeed, FlexiSketch is much more than a collaborative-whiteboard sketching tool.

**Creativity for Requirements Discovery**

Martin Mahaux from Namur University has developed a slew of creativity solutions for requirements
One such solution is the Collaborative Creativity Canvas (see Figures 2b and 4). Facilitators can use it to foster creativity and replace the often frustrating requirements negotiation process with a lively cocreation process. It aims to turn stakeholder conflicts into opportunities for innovation.

Although Martin’s research included traditional literature reviews, expert opinion, and practitioner surveys, it was driven largely by industrial collaboration. Ideas conceived in industry were iteratively and incrementally improved as they moved back and forth between the lab and practice. As such, Martin’s creativity solutions resulted from industrial partnerships and not through the more traditional model in which an idea emerges from research and then incubates in a lab for five years before the finished product is offered to industry.

Martin explained that this project revealed the benefits of industrial co-design, especially through the ongoing guidance and feedback he obtained. He pointed out that working with industry didn’t produce short-sighted research because he had the time and freedom to consider, and explore, innovative ideas throughout the process.

**Archie**

Architectural knowledge and related quality concerns are often undocumented and tacit. So, developers often lose track of early design decisions. For example, system-level qualities representing “non-functional” requirements tend to become eroded during refactoring, bug fixing, and other maintenance activities.

To address this problem, my research group at DePaul University developed Archie (see Figures 2c and 5), an Eclipse plug-in. Archie focuses on requirements’ role in a project’s downstream design and maintenance phases. It parses source code and then automatically detects and visualizes a range of architectural tactics such as heartbeats, resource pooling, and role-based access control.

Archie was funded by grants from the US National Science Foundation and Department of Homeland Security and developed by Ahmed Fahkry (see Figure 6) and other students under Mehdi Mirakhorli’s supervision. To place Archie into practitioners’ hands, we released it on GitHub under Archie-Smart-IDE and on the Department of Homeland Security’s SWAMP (Software Assurance Marketplace).

Mirakhorli explained that one of the greatest challenges for technology transfer was in understanding the real users’ actual usage patterns. We addressed this through frequent iterations of prototyping, coding, and testing. However, the real test will come as industrial users adopt Archie in their development environments. As such, Archie is less advanced along the technology-transfer scale than FlexiSketch or Martin’s creative collaboration activities.
The panelists questioned whether developers on real projects would find the prepackaged set of trained classifiers fit for their purpose. The Archie presenters explained that Archie was highly flexible and that teams could construct new tactic templates and either train new classifiers or use Archie’s click-and-point features to map code sections manually to the new tactics.

What’s Next?

The Ready-Set-Transfer panel always raises interesting technology-transfer issues—some of which we have answers for and some of which we don’t. Mike Panis saw benefits to both sides and explained that the panel “helps researchers—regardless of whether they are contestants or in the audience, to step back from the potential, future value of their research and consider what would be needed for it to provide immediate benefits.” He also observed that “it helps practitioners consider whether they can apply research results to their current work.”

So, challenges and opportunities abound. In my research, I’ve found that the biggest adoption barriers are the cost and effort of bringing viable research prototypes to industrial standards. A typical research grant doesn’t normally include funding for this kind of technology transfer, so researchers must proactively seek additional funding to jump the readiness hurdle.

One thing is clear. Successful technology transfer needs both sides of the partnership. We can’t succeed unless researchers and practitioners work together to address important problems that a typical software development project can’t accommodate. Neither can we be confident that we’re addressing the right problems at the appropriate scale and complexity without industry’s feedback and willingness to share data and expose its challenges.

As always, I’d love to hear from you. In a future column, I’d like to give more voice to practitioners. So, I invite you to email me and tell me about problems you’re experiencing that you wish researchers would address or about your success or failure stories regarding technology transfer. Let’s engage in an ongoing, fruitful discussion so that we can see innovative requirements projects—possibly seeded from industry—make their way to industry as viable, effective solutions.

References


JANE CLELAND-HUANG is a professor of software engineering at DePaul University. Contact her at jhuang@cs.depaul.edu.

FIGURE 5. A snapshot of Archie showing the detected heartbeat tactic highlighted in code and visualized in graphical form. For a demo of Archie, visit http://re.cs.depaul.edu/mehdi/Archie.mp4.

FIGURE 6. Ahmed Fahkry, Archie’s lead developer, explained that working on Archie gave him the opportunity to engage in a challenging research project as a graduate student.


JANE CLELAND-HUANG is a professor of software engineering at DePaul University. Contact her at jhuang@cs.depaul.edu.

See www.computer.org/software-multimedia for multimedia content related to this article.
What Next? Advances in Software-Driven Industries

Christof Ebert, Gerd Hoefner, and Mani V.S.

Which software technologies will be relevant in the near future? Where is development of innovative products and solutions heading? Companies and engineers need to flexibly respond to markets and changes and adapt their competences. But which of the many new technologies and hypes deserve focus? Further progress in software technologies, coupled with the growing capability to reliably develop complex systems, will impact every engineer. In this installment of Software Technology, Siemens experts Gerd Hoefner and Mani V.S. and I outline major software trends and offer recommendations for practitioners. I look forward to hearing from both readers and prospective column authors about this column and the technologies you want to know more about. —Christof Ebert

COMPLEXITY SCALES. With society’s increasing dependence on software, its complexity and scale continue to expand. Furthermore, while becoming more pervasive, software is also becoming more transparent, with increasingly invisible interfaces. Software is transforming entire industries such as healthcare and the automotive industry. Electronics, software, and IT are major drivers of innovation across all industries.

This article provides an overview on trends in software-driven industries. We have spoken with clients from technology companies worldwide in various industries to identify where they are heading and what topics are relevant for 2015 and beyond. We will outline major trends and supplemented these trends with concrete recommendations.

Products and solutions must meet more and more requirements but also must be designed for cost effectiveness, easy adaptability, and the ability to exploit the advantages of emerging and dominant industry platforms. New competitors are entering markets with new solutions, which in some cases circumvent the dependence on legacy systems. Software-driven systems are characterized by rapidly growing complexity coupled with an unprecedented increase in scale, based on a fast-changing technology landscape.

Software Advances
The Billion-Dollar Startup Club of the Wall Street Journal and Dow Jones...
VentureSource tracks companies that venture capital firms have valued at more than US$1 billion (http://graphics.wsj.com/billion-dollar-club). It’s an interesting source to compare technologies and their expected impact on worldwide needs. Currently, the growing list has 60-odd companies.

The Five Dimensions of Software Advancement
Looking at these companies along with our own wealth of industry contacts, we cluster software advancement along five dimensions (see Figure 1):

- **collaboration**—consumer Internet, social network interaction, single-customer segmentation, configurators for products and services, digital money, computer-assisted collaboration tools, and crowdsourcing;
- **comprehension**—semantic search, big-data handling, smart data, data analytics, the data economy, online data validation, and data quality;
- **connectivity**—ubiquitous mobile computing, mobile services, cyber-physical systems, Industry 4.0, machine-to-machine communication, sensor networks, and multisensor fusion;
- **cloud**—applications and services in the cloud, location-based networks, new license models for software and applications, sustainability, and energy efficiency; and
- **convergence**—mobile enterprises, bioinformatics, the Internet of Things, pervasive sensing, and autonomous systems.

These dimensions, coupled with the underlying complexity and scale, demand new software solutions based on new computing paradigms and infrastructure. Examples include IT architectures that facilitate seamless connectivity, robust infrastructures for cyber-physical systems in safety-critical environments, and data analytics to predict choices and behaviors to improve the overall customer experience. Such software-driven solutions can create nontraditional market entry points and consequently entirely new mechanisms to address a single customer with time- and location-specific services.

Security, Robustness, and Usability
Such solutions require new technologies that will not only create numerous opportunities but also introduce complexity. So, these solutions introduce new challenges—for instance, regarding information security, robustness, and usability.

Security and robustness have a tremendous impact on business decisions. The more we share and network, the more we’re exposed to attacks of all kinds. The exploding need for secure software and protection schemes for our business processes, end-to-end, indicate this impact. Imagine automotive suppliers working on multisensor fusion connected to GPS and vehicle-to-vehicle communication to predict critical situations and foresee appropriate measures in situations in which even the driver might not be aware of what will happen. Other examples are service companies that leverage their sales channels to flexibly provide related services such as door-to-door transportation, or firms that offer a single service card for identification, payment, and access to services of various providers, both physical and in the cloud.

Complexity and scale demand a focus on usability. We already face situations in which inadequately trained
users are forced to operate systems they don’t understand sufficiently to meaningfully assess risks and stay in control across normal day-to-day scenarios. Insufficient usability is a major source of critical failures caused by humans in healthcare, transportation, and production plants.

Creating Value
Software and IT move on a fast highway. Global software development requires managing software projects that span geographic and organizational boundaries, which adds to the challenge of developing software. But we see many companies and endeavors that failed because they overemphasized technology and didn’t sufficiently implement a sound business strategy.1

Consider Netscape. For many of us, Netscape was our first experience of the Internet. In 1995 it had a market share of 80 percent. But by 1997, it slowed down and lost market share, and in 2003 it went bankrupt. What went wrong? One manager put it simply, admitting that instead of using sound processes, they just dumped features and software technologies into their products.1 Insufficient product management still hampers companies. Research has shown that roughly half of delivered features don’t create value (see Figure 2).1,2

The Complexity Trap
While working with clients on product strategy and requirements engineering, one of the first questions we ask is, what value will a potential feature add to a product? The vast majority of responses we receive can be reduced to “We don’t know; the spec says this feature is required.” Although this might be true, it’s important to recognize that this will lead to an unsustainable increase in complexity and cost. This is what we call the complexity trap.

As complexity expands, it must be balanced. Companies that make the wrong decisions during a period of fast technology evolution and change will fall back or fail. Most selections involving human choices follow a “long tail” or 1/1 distribution. The Pareto principle states that, for many events, roughly 80 percent of the effects come from 20 percent of the causes. Today, economists have put more energy into analyzing the long-tail distribution; they’ve found that, for instance, going from 20 to 100 features in a system adds only 10 percent of value.3 Steve Jobs was one of the few taking concrete lessons from this principle, demanding simplicity in his products. Yet managers are unsure whether the measures taken are sustainable and how to manage the increasing innovation pressure. Specifically, they expect technology and IT proposals to both reduce costs and set the right priorities for technology innovation and new products and solutions.

Impacts for the Leading Practitioner
Simplicity secures. Too often software engineers are overly fascinated by new technologies. In addition, their companies obviously demand rapid transition to new products and services, to create sales opportunities and growth. However, in the interest of balance, you need to combine evolving software technologies with sound engineering and management practices.

In talking with clients in many different projects and industries worldwide, we identified four major levers to avoid an overly narrow focus on technology:

- Connect architecture and functionality.
- Master the entire life cycle.
- Strengthen globally distributed teams.
- Streamline development.

We now examine these levers and offer recommendations for practitioners. Most of the recommendations are further detailed in Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing.2

FIGURE 2. Only a small percentage of features actually create value in a software-driven system.
Connect Architecture and Functionality

Software and IT systems need a close connection of architecture and functionality. Any system's requirements and architecture are highly interdependent. Projects without early architecture evaluation will likely overlook critical requirements, especially nonfunctional ones. In addition, these projects are more likely to fail in prioritizing requirements. They typically face budget overruns and rework due to inappropriate and late design changes.

The following example illustrates this disconnect. A project team introduced a service-driven approach but did not adequately align the architecture and requirements. The team developed and modeled the requirements without looking to solution models and architecture constraints. They modeled business processes with a top-class modeling tool but didn't map the processes to the requirements. The intended service-oriented architecture copied previous technology-driven silos because this was the language everybody understood. The result was a mixture of individual components that didn't perform according to business expectations. Getting the project back on track required restarting the entire requirements-modeling and architecture concept—at a high cost of rework.

Modeling functions and architecture will help avoid such failures. Functions, architecture, implementation, and dependencies must be modeled and connected suitably. It's essential to evaluate nonfunctional requirements such as safety, security, performance, reusability, maintainability, and system cost—and their impact on architecture—at an early stage. Product line development with efficient variant management has a significant economic impact. The recent evolution toward modular concepts is a positive step. Working at a higher abstraction level and automating activities can improve productivity and quality. The trend is “from field to system to lab to math.” Development is getting increasingly virtualized, so that the code runs on multiple platforms.

The greatest obstacle is the learning curve, thus not achieving consistency across organization units. Developers are tasked with optimizing the code without getting the time or training to understand its function. Systems engineering remains in the background and is isolated from application development. So, roadmaps are created for only the subsystems, and new features and variants introduce overwhelming complexity. The business case is clear: Consistent modeling of the product's critical software components significantly reduces the defect rate and development costs. We typically find a share of 30 to 50 percent as adequate; that is, rarely used uncritical parts should not be modeled.

Recommendations: First, establish a strong focus on systems engineering and modeling of functions and architecture. Define quality requirements and measures early and consistently from a system point of view. Break the system down to its components and functions and analyze the impact of requirements on architecture. Integrate suppliers and customers into your overall quality and lifecycle concept. Reduce the nonconformance cost through integrated modeling, early defect detection, and reuse. Ensure consistency through traceability, automated consistency checking, and automated code generation from models of functionality.

Move stepwise toward model-driven development, and focus on critical components and consistency of requirements throughout architecture, design, and test cases. Measure the migration and its effects. In each project, try to improve by 10 to 20 percent in areas on which you want to focus—for example, 20 percent fewer cost variations or 10 percent lower cost in the test phase.

Master the Entire Life Cycle

Rising cost pressure is forcing companies and their suppliers to jointly and consistently master product development. Product life-cycle management and application life-cycle management (PLM and ALM) are the primary mechanisms for integrating engineering processes, tools, and people across the domains of system, software, hardware, and mechanical engineering. Unfortunately PLM and ALM often aren't well introduced. Companies believe that with a tool and the necessary IT interfaces, all issues will be resolved. This isn't the case; the high percentage of abandoned PLM and ALM projects indicates the criticality of professional change management.

The following example illustrates the significance. A supplier introduced model-driven engineering (MDE) based on a modern tool environment that enabled seamless collaboration across development centers and with partners and customers. Cost-effectiveness was evident up front because the system was going to provide faster data access, while the improved change and configuration control was expected to produce fewer defects and reduce budget overruns.

The engineers used the tool for modeling during design, but not during the test, and without much modeling methodology. Soon, models became inconsistent and were...
discarded in further product evolution. What had happened? The tool was designed to support the development and was integrated into the company-wide product data management system. However, not only developers but also product managers and project leaders couldn’t work with the tool and created parallel systems for their documents, which they exchanged using traditional methods. The solution would have been effective if, before the tool was introduced, it had been made clear which processes had to be supported along the life cycle with which methodology, and how these processes had to be first improved and then automated.

**Recommendations:** First improve the process and then the tools, on the basis of concrete improvement objectives that are set, measured, and used to correct deviations. Ensure consistency of features and products with a strong systems-engineering approach. Specifically, in distributed collaborative environments, we see huge benefits from a single repository for consistent requirements, specifications, and models across all versions. Use tools to appropriately model the different abstraction levels, from functions to logic and from architecture to implementation. Evaluate tools on the basis of your own requirements under realistic conditions. Support the interfaces to the various components and processes through traceability, automatic consistency checks, and test automation.

Manage the transformation across the entire organization. Pilot the changes in process and tools, coach and train engineers, and recognize the power users who will set the pace. Introduce model-based development intelligently; step by step, focus on critical components, continuity of requirements throughout code and test cases, and improving processes in parallel. Support developers and ensure continuous improvement.

**Strengthen Globally Distributed Teams**

The pressure to continuously innovate and reduce cost, the lack of the right skills, and the need for global presence will further boost distributed development. Figure 3 shows the four major drivers of global software engineering:

- **presence**—in local markets, both for visibility and access but also for learning;
- **talent**—to succeed in the race for the best engineers;
- **flexibility**—with just-in-time networks and emerging ecosystems and crowdsourcing; and
- **efficiency**—to balance the complexity and simplicity in reducing overheads and being more agile.

Distributed development teams require new forms of collaboration for teams, projects, and people. The diverse network of components, applications, devices, and users is demanding completely new ways of working. However, communication difficulties, cultural differences, and management overhead lead to numerous challenges. Eighty percent of companies that outsource their development or maintenance have problems. Twenty percent of sourcing contracts are canceled within the first year. Fifty percent of the contracts don’t achieve the intended objectives and are then terminated.²

So, what’s needed is **smartshoring**, which replaces traditional labor-cost-based location decisions with a systematic improvement of business processes in a distributed context. The benefits are tangible; the most often reported ones are multisite collaboration, clean-variant management, and transparent workflows. Merely considering labor costs must be replaced by a holistic strategy taking into account onsite presence and customer proximity as well as reducing friction losses.

**Recommendations:** Prepare distributed teams and smartshoring as a competence and business process before going operational. This involves risk management, vendor se-
Streamline Development

In uncertain economic times, technologies and processes that make a company more efficient and powerful gain attention. Software development managers must evaluate costs and productivity and implement goal-oriented improvements. In our experience, you can reduce cost and cycle times by continually optimizing development processes. The application of lean-software-development principles and agile practices can effectively streamline interfaces and reduce rework and inefficiencies. Agile practices such as Scrum or Extreme Programming are increasingly applied but often seem a mere slogan instead of a sustainable way to work.

Our experience shows that far too often, teams reinvent the wheel. Earned value, value stream mapping, and Scrum are proven techniques that shouldn’t be developed in-house with a lot of energy and cost. Our clients at Vector claim that nearly 90 percent of companies want to improve their efficiency in 2015 but that only a third of them are satisfied with their previous results. Unprofessional change management is a common reason for failing efficiency projects.

Recommendations: Check your project performance: what creates pressure in projects, what demotivates your teams, and where do work products need rework? Streamline workflows and related tools stepwise, with an overarching strategy, incremental goals, and a future-oriented IT architecture. Set concrete improvement targets every quarter. Train employees in lean principles. Have each team develop its own action plan for reducing waste, rework, and interface conflicts—with reference to your company-wide efficiency targets. Evaluate your performance—for example, by revenue per developer, lead time, fault detection rates, and cost drivers. Ensure that agile practices don’t lead to arbitrariness. Apply professional change management.

Complexity Scales, Simplicity Secures

2015 calls for more efficiency and competitiveness, because the business climate is more volatile. We talked to our clients from various industries to identify where they are and what topics will be important for product development in 2015 and beyond. The response: companies will continue to invest in growth through innovation by developing new products and solutions because this determines their market position. At the same time, they’re aware of the volatile market situation and want their development teams across the world to be as lean and efficient as possible.

These days, companies must be doubly innovative—in both technology and efficiency. The ability to successfully implement innovations in a short time is the all-important competitive factor. Innovations are not only new products and optimized processes but also entirely new basic technologies, such as we see in electric vehicles, communication networks, or even intelligent energy use.

ICGSE

The annual International Conference on Global Software Engineering (ICGSE) brings together worldwide industry and research leaders in distributed software development. With participants from more than 20 countries and one-third of the papers from industry, it’s the preeminent forum on global software development. ICGSE 2015 will take place from 13 to 16 July in Ciudad Real, Spain. Join the conference and learn how to succeed with distributed software projects. Meet the authors of this article, and learn from their keynote speeches how to make distributed teams more effective, the benefits of smartshoring, and how to cope with the challenges, such as heterogeneous methods and tools. More information is at www.icgse.org.

Participate in our technologies 2015 survey and have the chance to win a free copy of our IEEE book Global Software and IT with many practical case studies. Directly go to the survey: www.vector.com/trends-survey.
Insufficient use of road maps, unmanaged complexity, and cost-cutting in the wrong areas lead to a situation in which a significant proportion of R&D expenditure doesn’t lead to successful innovation. Customers in various industries complain about overly long cycle times from idea to market. Techniques such as lean innovation can address 20 to 40 percent of customers’ total cost structure. This is an enormous savings potential that should be effectively and sustainably captured.

What determines a product’s success isn’t the number of features; it’s the few that differentiate it from others. Complexity scales but must be mastered with product strategy, sound engineering processes, and technology management to achieve the necessary simplicity that secures your growth and sustains your markets.

**References**


**CHRISTOF EBERT** is the managing director of Vector Consulting Services. He serves on the *IEEE Software* editorial board. Contact him at christof.ebert@vector.com.

**GERD HOEFNER** is the managing director and CEO of Siemens Technology and Services in Bangalore. He’s also the head of Siemens’ Corporate Development Center. Contact him at gerd.hoefner@siemens.com.

**MANI V.S.** works on the marketing team of Siemens Technology and Services in Bangalore. Contact him at vs.mani@siemens.com.
Mobile Money’s Impact on Tanzanian Agriculture

Balachandran Seetharam and Drew Johnson

The 25th Impact column comes from Africa and shows how software enables the use of mobile money by farmers and has significantly benefited agriculture in Tanzania. —Michiel van Genuchten and Les Hatton

TANZANIA, with a population of 48 million, has one of the fastest-growing economies in the world. Situated on Africa’s east coast, the country has sustained an annual growth in GDP (gross domestic product) of 5 to 8 percent since 2000.1 The World Bank predicts a continued annual GDP growth of 7 percent.2 Agriculture, the backbone of the Tanzanian economy, constitutes 27 percent of the GDP and employs 80 percent of the workforce.3 A typical Tanzanian farmer cultivates a small parcel of land (two to four acres) and grows a combination of food and cash crops. The entire family performs farm work, using hand tools or draft animals. Although the country is rapidly developing, most rural Tanzanians still live on just a few dollars per day.

Tanzanian smallholder farmers face a myriad of challenges. Poor education, lack of technology access, and limited infrastructure continually curb their potential for success. When farmers grow high-quality crops, they struggle to get a fair price owing to the long supply chain from farm to fork. A series of middlemen traders who aggregate and transport produce from rural areas exploit information asymmetries and market inefficiencies to negotiate low prices from farmers. Farmers depend on these middlemen to purchase their goods, but the farmers have limited bargaining power owing to their low crop volumes (from small acreage) and lack of market information.

The dominant smallholder farming environment is contrasted by a nascent commercial agribusiness industry. Investments in commercial agricultural production often integrate an outgrower scheme—a contractual agreement between smallholder farmers and the agribusiness. Farmers produce crops of a certain type and quality, which they sell to the agribusiness. The agribusiness provides services, such as supplying seeds and fertilizer, to the farmers. The nucleus commercial farm serves as a central point for services, aggregation, and processing. Outgrower schemes can create symbiotic relationships between smallholder farmers and agribusiness. The farmers benefit from access to improved services, such as equipment and loans, and a reliable market. The agribusiness benefits by accessing produce from thousands of farmers.

To support and improve the relationship between farmers and agribusiness, Vodafone, with TechnoServe’s support, developed an application that helps
agribusiness improve their efficiency, process payments faster, and purchase more from farmers. Such technology has the potential to greatly improve the livelihoods of farmers in countries such as Tanzania.

**Tanzania and Mobile Networks**

Tanzania is one of the fastest-growing mobile markets, with an annual subscriber growth of 20 percent. More than 55 percent of the population own SIM (Subscriber Identity Module) cards, for a total of 26.6 million mobile connections. Eight mobile network operators (MNOs), of which four are major players, compete for consumers. The competition has led to falling voice tariffs. Now, the MNOs are focusing on Internet and other services. Tanzania has 6.1 million Internet users, with over 90 percent accessing it via mobile phones.

The hope is that information and communications technology will play a significant role in the country’s economic development. Studies show that a 10 percent increase in household Internet penetration yields up to a 1.4 percent increase in a country’s GDP. Access to information and knowledge improves productivity and unlocks opportunity. Mobile phone networks are being used to deliver social services, increase access to market opportunities, and transfer money.

**Mobile Money**

Although services such as Google Wallet and Apple Pay are considered new innovations in the US, mobile-money services were actually pioneered in East Africa. In 2007, Safaricom launched M-Pesa, and Vodacom Tanzania followed in 2008. Mobile-money services let users send and receive money on even the simplest mobile phone. The users store money in a digital wallet and can withdraw it at any of the thousands of small kiosks throughout Tanzania (see Figure 1). The system is protected by a two-factor authentication system: the SIM card plus a PIN. If the user loses his or her phone, any balance is safely stored until the user recovers the account at a kiosk.

Mobile-money services now reach 90 percent of mobile subscribers in Tanzania. Of the adult population, 44 percent have used mobile money, with at least one family member holding an active mobile-money account in 35 percent of Tanzanian families. During December 2013, customers made 100 million transactions totaling US$1.8 billion.

The arrival of mobile money has a significant social impact in a country where 78 percent of the people have no bank account. Mobile money facilitates the safe storage and transfer of money for people who previously lacked access to formal financial services. By providing safe storage, mobile money encourages households to save cash for important bills such as school fees. Because long-distance transfers are easy and cheap, households can safely send money to family members hundreds of kilometers away. The benefits are far reaching. One research study, comparing the household characteristics of users and nonusers of mobile money with similar earnings, showed that the users had a higher wealth index and more than double the assets.

**Case Study: Multiflower**

Multiflower is an agricultural exporter founded in 1995 and headquartered in Arusha. The company has two divisions: flower seeds and flower cuttings. In the flower seeds division, Multiflower employs an outgrower scheme involving 3,300 farmers.

At the beginning of the growing season, farmers enter into a contract to grow specific varieties of flower seeds on the basis of weather, agronomy factors, and land size. Multiflower aggregates the resulting seeds and sells them to Multiflower. Multiflower aggregates the seeds and exports them to prearranged buyers in Europe. After pack-
aging and transportation to a local store, the seeds are sold to consumers, who plant them in their gardens. Figure 2 shows the basic process.

Multiflower provides a range of services to its contracted farmers. For example, a field team advises them on proper agronomic practices. Farmers can request loans in the form of goods, such as fertilizer and pesticides, and cash. The farmers repay the loans when they sell their seeds to Multiflower. By buying seeds, Multiflower provides a reliable market for the farmers.

**Challenges**

Managing 3,500 smallholders who grow more than two dozen varieties of flower seeds poses significant challenges for Multiflower. We focused on the following challenges.

**Challenge 1: Uncoordinated Data Management**

Tracking the data for all Multiflower farmers was time-consuming. Field staff regularly collected information from farmers using paper forms. At the office, the information was compiled so that management and clients could see the harvest’s status.

**Challenge 2: Manual Loan Processing**

As we mentioned before, farmers required loans of goods or cash. Farmers derived value only when they could see the harvest’s status.

**Challenge 3: Farmer Payments**

Multiflower paid farmers in cash for their seeds. During the season, farmers received several payments for each delivery, resulting in complex operations for Multiflower. Additionally, handling the cash was a risky, manual process involving up to seven employees.

**Challenge 4: Communication**

Multiflower staff struggled to disseminate messages and reminders to farmers spread across a wide area with poor infrastructure. Messages had to be relayed through field staff or sent individually by mobile phone.

**Addressing the Challenges**

To address these challenges, we devised a two-phase solution. Phase 1 addressed data management and loan advances; phase 2 addressed farmer payments and communication.

This solution had to work within four constraints. First, both farmers and Multiflower used simple phones. Second, Multiflower was reluctant to invest in an unproven system. Third, network connectivity was sparse in rural areas. Finally, the M-Pesa agent network in rural areas was underdeveloped.

**Phase 1**

We mapped our system to the manual processes that Multiflower followed; Figure 3 diagrams the resulting process. All system users, including farmers, farmer representatives, and field officers, are registered by the solution provider’s master administrator through a Web portal on a PC. This process also captures the field officers’ mobile phone numbers.

When they access this application, the system authenticates their mobile phone number to prevent fraud. The request goes to the Multiflower production manager, who evaluates it and modifies, rejects, or approves it. On approval, the request goes to the account assistant and account manager for further processing and payment via mobile money (M-Pesa). During this process, farmers receive continual updates through SMS (short message service).

Our solution uses gateways and servers hosted by the solution provider in the mobile operators’ hosting environment (see Figure 4). The required USSD multimodal number is allocated with a mapped URL pointing to the Multiflower application server. When a field officer dials the USSD number, the gateway directs the request to the application server hosting the Multiflower application. The application authenticates the field officer’s mobile number (MSISDN; Mobile Station International Subscriber Directory Number) before providing access to the rest of the Multiflower application. The application server captures the request and validates all the data, including the farmer’s MSISDN. The application server, on the basis of the workflow, forwards the request to the production manager, account assistant, and account manager. The application generates regular SMSs on the loan status and delivers them to the farmer through the SMS center (SMSC).

**FIGURE 2. The agriculture workflow of Multiflower, an agricultural exporter headquartered in Arusha, Tanzania.**

![Diagram of Multiflower's agriculture workflow](image-url)
For cash loans via M-Pesa, Multiflower transfers the money to the farmer’s M-Pesa account. Once the production manager approves the loan, the application server submits the farmer’s MSISDNs and the loan advance amount to the mobile-money aggregator platform. We used the mobile-money aggregator because it was integrated with the M-Pesa platform APIs. M-Pesa checks for the balance available in Multiflower’s M-Pesa account and transfers the amount to the farmer’s M-Pesa account. M-Pesa notifies the Multiflower application whether the transfer was a success or failure.

Phase 2
This phase addressed the challenges of making and receiving bulk payments, which were time intensive and risky. Previously, collecting bulk payments involved significant time and travel by the farmers to visit the Multiflower office. Multiflower also had significant challenges conveying regular information to farmers.

This phase required a process mapping that we translated into business requirements and technical specifications. The solution has a detailed workflow process, starting with the stores officer collecting the seeds, the quality inspector performing quality tests, and the account team processing payments in two steps (see Figure 5). Also, we developed a Web portal for relaying customized SMS messages to the farmers.

Building the Solution
We developed the Multiflower application on top of the Helium platform in a proprietary DSL (domain-specific language).

A USSD menu builder program created the menu for the Multiflower application. SMS was mobile terminated, meaning there were no incoming SMSs from farmers. The applications used the PostgreSQL database. M-Pesa was on a Vodafone universal platform with API interfaces for third-party applications.

The Helium platform used PostgreSQL, Java EE (Enterprise Edition) 7, and GlassFish. The Helium platform was a generic platform built for hosting applications across domains; it comprised more than 500,000 LOC. The mobile-money middleware was written on Java EE 7 to connect to the mobile-money aggregator platform. Vodacom Tanzania provided the short code, SMSC, and USSD gateway. A third-party menu-builder company in India built and hosted the USSD menu. Vodacom Tanzania hosted the application server on a 64-bit Ubuntu Linux server.

The entire Multiflower solution (the two development phases; integration with the SMSC and the USSD gateways; and integration with the mobile-money platform, the Helium platform, and the hosted USSD front end) required 2.5 calendar months, including quality assurance (QA) and testing. Three engineers coded the application with support from a technical architect, a project manager, and two QA test engineers. The project took 980 person hours, taking into account shared resources such as QA, the technical architect, and the project manager. The Multiflower application, SMS relaying platform, and USSD front-end development and integration comprised 48,000 LOC. Some of the 48,000 LOC were reused from similar applications.

FIGURE 3. The eight-step process to register users and create loans to farmers, using PCs and mobile phones. SMS stands for short message service, GSM stands for Global System for Mobile Communications, and USSD stands for Unstructured Supplementary Service Data.
Impact

From June 2013 to March 2014, Multiflower piloted the solution with 300 farmers in northern Tanzania. The solution benefited both the farmers and Multiflower.

For the company, the solution improved business operations. Previously, Multiflower lacked a central system for managing farmers’ information, which was scattered across paper forms, Excel files, and local databases. Following the solution implementation, Multiflower staff reported improved visibility into field operations, streamlined farmer payments, and better communication. The company could make bulk digital payments instead of individual cash payments. Also, Multiflower staff could quickly and easily send important bulletins to farmers over SMS. Automated messages during payment and loan processing gave farmers greater visibility of their payment status.

The farmers benefited most from the mobile payments. As we mentioned before, farmers previously traveled to the Multiflower office to collect cash. Many of them live more than 150 km from the office and would spend the entire day and $13 on bus fare to reach it. Then, at the office, they queued to wait for their cash. On the return trip to the farm, thieves would sometimes target the farmers owing to the significant amount of cash they carried.

With the introduction of the mobile payment system, farmers could receive loans and payments directly on their mobile phones. A farmer at home would receive an SMS notification about the deposit. To collect the payment, the farmer could then travel a few kilometers to a mobile-money agent in the farmer’s local village. Farmers paid a nominal fee to withdraw cash from their digital wallet. By using the new system,
the 300 farmers saved an estimated $8,000 and 6,000 hours.

Future Opportunities

This implementation’s success demonstrates opportunities for MNOs and agribusinesses like Multiflower.

MNOs tend to gravitate toward easily accessible urban markets. In the fierce competitive environment, these markets quickly become saturated. Mobile agriculture solutions give MNOs a significant opportunity for new revenue streams and the ability to improve their rural-market exposure, while strengthening core business operations (voice, messaging, and mobile money).

By using cost-effective mobile technology, agribusinesses can modernize their operations. The adoption of mobile-enabled digital workflows and mobile payments leads to efficiencies and cost savings. Agribusinesses that work with smallholder farmers, through outgrower schemes, have the most potential to benefit.

Those benefits don’t stop at the agribusiness level. As the Multiflower case study illustrates, farmers benefit from receiving timely communication and mobile payments. With 75 percent of the world’s poor living in rural areas, and with most dependent on agriculture, mobile technology has a huge potential for improving lives and reducing poverty.

The Impact department in IEEE Software has previously reported on mass- software systems of millions of LOC. The system we just described has a huge potential for improving lives and reducing poverty.

The Impact department in IEEE Software has previously reported on mass- software systems of millions of LOC. The system we just described has a huge potential for improving lives and reducing poverty.

FIGURE 5. Distributing payments to farmers for flower seeds. This phase required a process mapping that we translated into business requirements and technical specifications. GRN stands for goods received note.

References


BALACHANDRAN SEETHARAM is the head of products at Vodafone Solutions (Emerging Markets) India. Contact him at balus67@gmail.com.

DREW JOHNSON is a program manager at TechnoServe. Contact him at adj131@gmail.com.
Software Engineering for Internet Computing
Internetware and Beyond

Antonia Bertolino, Italian National Research Council
M. Brian Blake, University of Miami
Pankaj Mehra, SanDisk
Hong Mei, Peking University
Tao Xie, University of Illinois at Urbana-Champaign
THE INTERNET, once a network of networks, has become not just the platform of choice for delivering services to increasingly mobile users but also the connective tissue among people, information, and things. The newest and most popular computing and application paradigms have been born on the Internet, or at least motivated by it, such as Web 2.0, social networking, mobile Internet, cloud computing, the Internet of Things, and big data.

An Open, Dynamic, Evolving Environment
The open, dynamic, and evolving environment of Internet computing continues to demand new software technologies. Such technologies should be context aware, adaptable, and able to evolve to effectively deal with rapid changes in user requirements and run-time environments. The Internet-based software ecosystem increasingly impacts software engineers by redefining their roles and patterns for collaboration, innovation, and value creation, particularly in global distributed environments.

Software engineering for Internet computing involves the architecting, development, deployment, management, and quality assurance of software supporting Internet-based systems. It also addresses global-development issues such as communication complexity, distributed control, governance policies, and cultural differences. So, new programming and life-cycle paradigms, such as Internetware,1,2 are inevitable. Example research topics for this area include, but aren’t limited to,

- programming models and platforms for dominant and emerging Internet-based systems such as cloud computing,3 service computing,4 social computing,5 mobile Internet,6 the Internet of Things,7 and cyber-physical systems;
- platforms and application frameworks for Internet-based software, such as Web-based integration (for example, REST [representational state transfer] and JSON [JavaScript Object Notation]), infrastructure and JSON
- frameworks for Internet-based software, such as Web-based integration (for example, REST [representational state transfer] and JSON [JavaScript Object Notation]), infrastructure and JSON
- and Web-scale data analytics and content handling (for example, MongoDB and Hadoop);
- quality-assurance approaches and security-and-trust aspects8 in the engineering of Internet-based software;
- software design models for Internet-based software, such as UML, BPM (Business Process Management),9 and Petri nets;
- software development processes and tools for the Internet (for example, agile development for Internet-based software) or with the Internet (for example, cloud-based development environments);10
- technology and human-interaction models and techniques in the development of Internet-based software;
- migration or integration of legacy software into Internet-based software;11 and
- case studies and experience reports on one or more of the previous aspects in industry practices.

This Special Issue
This issue includes the following exciting and representative research.
In “Debugging the Internet of Things: The Case of Wireless Sensor Networks,” Patrick Eugster and his colleagues relate their experiences developing debugging tools for wireless sensor networks, enablers of perception in the Internet of Things.
In “Automated Synthesis of Service Choreographies,” Marco Autili and his colleagues describe a tool for creating service choreographies. A choreography is a form of decentralized service composition that describes peer-to-peer message exchanges among participant services from a global perspective.
In “Stigmergy-Based Construction of Internetware Artifacts,” Wei Zhang and his colleagues present an approach that enables the continual construction and evolution of model-based Internetware artifacts by the collective of software stakeholders connected by the Internet.
In “Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications,” Yepang Liu and his colleagues describe the characteristics of energy and performance bugs in smartphone applications. They discuss challenges...
and techniques in diagnosing them, and study tool usage for analyzing smartphone applications and software development kits.

In “A Tail-Tolerant Cloud API Wrapper,” Qinghua Lu and her colleagues explore the characteristics of cloud APIs, using Amazon EC2 (Elastic Compute Cloud) as a test bed. They also present mechanisms to improve cloud API performance.

In “Multitier Diversification in Web-Based Software Applications,” Simon Allier and his colleagues introduce an approach that extends software diversification beyond the OS level, as a step toward breaking Internet application monoculture.

Finally, to look at what’s beyond Internetware, seven outstanding researchers in the field (Jian Lu, David Rosenblum, Tevfic Bultan, Valerie Issarny, Schahram Dustdar, Margaret-Anne Storey, and Dong-mei Zhang) share their views on the future of software engineering for Internet computing.

Acknowledgments

We thank the authors and reviewers of this special issue for their great effort and contributions. We’re also grateful to the IEEE Software production team for their hard work in assembling and editing the issue.

References

Debugging the Internet of Things
The Case of Wireless Sensor Networks

Patrick Eugster and Vinaitheerthan Sundaram, SensorHound
Xiangyu Zhang, Purdue University

// Proposed debugging tools for wireless sensor networks gather run-time information on sensor node executions and interactions and then compress that information. //

THE INTERNET OF THINGS (IoT) could lead us into a new era in which humans interact with their physical environment more symbiotically. The miniaturization of sensing and computing devices supports many futuristic visions such as smart grids, smart buildings, or entire smart cities. Sensing devices coupled with actuators through networked computer systems will sense environmental cues including our own behaviors. They’ll then take actions to improve our physical context in real time. Improvements will range from enhanced perception, physical well-being, and health, to reduced energy footprints and pollution.

A core assumption underlying these visions is that computer systems behave reliably—that is, satisfy our expectations. Deviant behavior in these systems could have catastrophic environmental consequences. For example, the infamous 2003 blackout in northeast North America was due to a software defect in the electricity grid’s alarm system.1 Thus, a core tenet of the IoT is that computer systems have no intrinsic flaws. Complying with this tenet requires impeccable software development techniques and tools. This is no small feat, considering that even more traditional networked computer systems—which have a smaller degree of distribution, ampler resources, and less dynamic behavior—don’t fully live up to the challenge.

The situation is especially dire in the context of wireless sensor networks (WSNs)—tiny battery-powered sensing devices interacting by wireless communication. WSNs are the foundation of the IoT in that they’re its antennas, allowing for the perception upon which other devices react.

The challenges of programming for WSNs have prompted us to reconsider the best practices for developing reliable software systems. Specifically, owing to the inherent difficulty of predicting all execution conditions, we’ve investigated run-time debugging techniques for WSNs. Unfortunately, hardware and resource constraints make existing techniques inapplicable for this. Here we describe our discoveries and experiences.

The Challenges of Programming Reliable WSNs
You might be tempted to think that the small form factor of such sensing devices and the actuators makes them simpler to program than, say, desktop computers and thus more manageable regarding assurance. However, four characteristics inversely hamper these devices’ reliability.

First, many tiny devices employ specialized hardware. This calls for new software paradigms and tools
that will be less mature than the corresponding ones for traditional systems.

Second, tiny, specialized hardware often has resource constraints. Besides plaguing the application of existing tool chains, these make the application of existing fault-tolerance techniques challenging or even impossible, given that most such techniques employ redundancy (for example, in space and memory or time and processing cycles).

Third, many IoT scenarios require dynamic interaction because of the open nature of the real-life physical contexts that such applications target. This makes it hard to foresee all constraints and fully validate systems before fielding them.

Finally, dynamic environments further exacerbate the difficulty of exhaustively predicting execution contexts. Many IoT scenarios include precisely austere environments that rely on technology to protect humans.

**Debugging Resource-Constrained Nodes**

Wireless sensor nodes (commonly called *motes*) are extremely resource constrained. For example, the mica family of motes typically includes 8-bit processors with 4 Kbytes of data or program memory and 128 Kbytes of flash memory. This makes programming such devices challenging; these challenges carry over to designing solutions for debugging the corresponding software. In particular, full debugging that allows for precisely replaying entire executions is infeasible, calling for solutions that trace (keep track of) only certain aspects of execution on motes.

**Tracing Control Flow**

TinyTracer is a tool we conceived for tracing control flow on motes running TinyOS, on the basis of the following insights.

**Effectiveness.** Control-flow information is effective in fault diagnosis because many defects manifest themselves through abnormal control flow. Knowledge of the control flow can help considerably in diagnosing a large class of errors that change the control flow. Such errors include logic errors, high-level race conditions, node reboots, network failures (node or link failures), and memory defects.

Despite the available sophisticated techniques and tools (see the sidebar “Debugging Wireless Sensor Networks”), developers often use LED debugging. They manually annotate a program to switch LEDs on and off to notify the success or failure of events of interest. They can use this debugging to obtain the control-flow path.

Control-flow traces also adequately capture some effects of input values such as sensed values and network messages. For example, if a network message represents a command to the sensor to send the current sensed values to the base station, the control flow reflects the message content. Likewise, if the sensed value exceeds a threshold, the control flow captures the action the sensor took.

For defects such as routing-table corruption by malicious entities, recording input values could be useful. However, such an approach is impractical for WSNs owing to the large trace sizes and stringent constraints on memory. Control-flow traces yield a good compromise that allows partial replay of the execution.

**Efficiency.** WSN applications repeatedly execute the same sequences of actions, with occasional unusual events. So, if the control-flow path is suitably encoded, the repetitive sequences can be highly compressed, leading to resource-efficient recording of control-flow information.

**Exploiting TinyOS**

The TinyOS event-driven execution model poses unique programming challenges. However, it allows for efficient solutions for severely resource-constrained environments, compared to the execution model of general-purpose program environments, which allow arbitrary thread interleavings.

A TinyOS application often comprises a set of reusable components that are “wired” through configurations. Components communicate through interfaces consisting of commands and events. A component provides services to other components through commands. If a component requests a service, an event signals the request’s completion to the component. Events are also the channels through which the hardware communicates with components. In a layered
DEBUGGING WIRELESS SENSOR NETWORKS

Run-time debugging is a challenging problem even in “traditional” resource-rich wired networks. Online debugging, which allows for insights into programs as they execute, can reduce the latency of fault detection and diagnosis. However, it tends to incur high run-time overhead and is susceptible to Heisenbugs. In contrast, offline debugging collects information traces from the execution of programs for replaying the program or analyzing its execution later. Techniques for wired networks don’t apply to wireless sensor networks (WSNs) because these techniques require large amounts of data memory and nonvolatile memory to store megabytes or even gigabytes of traces generated for subsequent offline mining.

Researchers have proposed several debugging solutions for WSNs. Automated debugging tools6,8–11 mostly monitor the network, so they don’t provide much help for program faults because the causes and symptoms might be far apart. These tools might also impose expensive synchronization to achieve good resilience. Or, programmers might have to express desired program invariants or perform SQL-like queries to retrieve run-time debugging information.

Remote-control techniques provide insight into—and control of—remote sensor nodes.13,14 They incur substantial overhead and require developers to navigate through the program at run time. Finally, program-analysis-based tools provide higher-level understanding of programs.15,16 However, they fail to quickly pinpoint the exact causes of faults or are highly complex.

As a form of offline debugging, replay debugging is a powerful technique for diagnosing complex faults. It’s especially useful for debugging distributed applications because of its inherent nondeterminism. Replaying requires obtaining the trace of the ordered sequence of events or the control-flow path and the input values. This technique might not present insurmountable issues in a wired setting. However, it can be prohibitively expensive in a WSN in terms of the space required to store the trace and in the bandwidth required to transfer the trace from a node to the base station. So, an efficient tracing scheme is essential. The key problem is to decide what information to record that allows partial, but faithful WSN replay debugging while satisfying resource constraints.

References

architecture, commands can be viewed as downcalls and events as upcalls.

There’s no explicit thread abstraction because maintaining multiple threads needs precious RAM and because thread interleavings easily introduce subtle data race errors. Nonetheless, TinyOS applications need a mechanism for parallel operations to ensure responsiveness and to respect real-time constraints. TinyOS has two sources of concurrency: tasks and interrupt handlers.

Tasks are a deferred computation mechanism. In the absence of events, they run to completion and don’t preempt each other. They’re posted by components. The post request immediately returns, deferring the computation until the scheduler executes the task later. To ensure low task execution latency, individual tasks must be short. Lengthy operations should be spread across multiple tasks.

Interrupt handlers correspond to events and are thus often called events. Events also run to completion but, unlike tasks, might preempt the execution of a task or another event. An event signifies either completion of a lengthy (and thus split) operation or an event from the environment (for example, message reception or time passing). TinyOS execution is ultimately driven by events representing hardware interrupts.

The executions of tasks and events are sequential or nested. This yields the foundation for our efficient tracing in TinyTracer. TinyTracer leverages knowledge of possible executions to minimally encode actual executions, including intraprocedural and interprocedural control flow based on a specifically tailored variant of Ball-Larus path encoding. For more details of our approach, see the sidebar “Node-Level Tracing with TinyTracer.”

### Debugging across Nodes

Many WSN defects involve interaction between several nodes. Replaying for WSNs thus requires interactions to take place faithfully and in the right order. A correct replay maintains the causal ordering of messages observed in the original execution. This ordering is usually defined as follows:

- **FIFO order.** A message send causally precedes its receive and subsequent sends by the same process.
- **Local order.** A message \( m_1 \) received by a node before it sends message \( m_2 \) causally precedes \( m_2 \).
- **Transitivity.** Causal ordering is transitive; if \( m_i \) causally precedes \( m_j \) and \( m_j \) causally precedes \( m_k \), then \( m_i \) causally precedes \( m_k \).

To obtain the original execution’s ordering, the message dependencies must be recorded in the trace. To capture these dependences, trace-based replay can be used. However, for wired distributed systems uses program-managed **logical clocks.** Originally, these clocks were designed for WSNs and the TinyOS environment (for example, message reception). A correct replay is required to understand causal ordering when debugging.

- **Robustness.** Accuracy goes even further in many distributed systems. Communication is often unreliable; messages can get lost. Similarly, messages can get reordered by the network.

**In debugging and other scenarios, false positives are a strong impediment.**

Certain losses and reordering can be tolerated by many applications yet must be taken into account when you’re reasoning about them.

- **Efficiency.** Even if clocks are required to be fully accurate even in an unreliable network, they might not compromise on efficiency. That is, the time and space required to recreate it in the replay.

**Lamport clocks** are a simple form of logical clock that use a single-integer-value clock maintained by each node and appended to any message. Although scalable, they’re inaccurate; they classify some concurrent events as causally related. Nonetheless, tracing-based replay usually combines Lamport clocks with sender identifiers (to distinguish between concurrent events with identical time stamps) because of their ease of implementation and scalability.

This approach is inadequate in WSN debugging because it can’t meet these requirements:

- **Robustness.** Accuracy goes even further in many distributed systems. Communication is often unreliable; messages can get lost. Similarly, messages can get reordered by the network.

**In debugging and other scenarios, false positives are a strong impediment.**

Certain losses and reordering can be tolerated by many applications yet must be taken into account when you’re reasoning about them.

- **Efficiency.** Even if clocks are required to be fully accurate even in an unreliable network, they might not compromise on efficiency. That is, the time and space
FOCUS: INTERNETWARE AND BEYOND

**NODE-LEVEL TRACING WITH TINYTRACER**

Figure A shows how tasks and events execute in the TinyOS concurrency model. Boxes depict the task (T) or event (E) execution. Boxes with the same pattern represent the same execution instance of a task or an event, and their labels denote the task or event names. A task or an event can be preempted, giving rise to multiple boxes with the same pattern. The level of nesting involved in preemption is shown vertically for clarity.

Figure A1 presents a legal execution. Task T1 executes without preemption. Event E1 occurs sometime after T1 and is preempted by E2. Once E2 finishes, E1 resumes and runs to completion. T2, E3, and E4 represent a more complex case in which multiple event preemptions occur. When T2 is running, E3 occurs and preempts T2. E4 occurs, preempts E3, and runs to completion, upon which E3 resumes and runs to completion. T2 resumes and gets preempted by another invocation of E3, which might correspond to another instance of the same hardware interrupt received by the handler E3. Once E3 completes, T2 resumes and runs to completion without further preemption.

Figures A2 and A3 represent impossible execution sequences. In Figure A2, a task can’t preempt another one. In Figure A3, the executions of a task and an event can’t interleave because the preempting execution must complete before the preempted execution gains control.

We exploit the nesting structure of TinyOS application execution to describe traces with a context-free grammar:

![Figure A](image-url)

**FIGURE A.** Task and event execution in the TinyOS concurrency model. The y-axis represents the preemption level. (1) A legal execution. (2) An illegal execution in which a task can’t preempt another one. (3) An illegal execution in which the executions of a task and an event can’t interleave because the preempting execution must complete before the preempted execution gains control.

![Figure B](image-url)

**FIGURE B.** This string of grammar depicts the execution in Figure A1. Overbraces represent the nestings. Although this representation requires a number of end symbols, the symbol is universal, so encoding requires far fewer bits, compared to task or event identifiers.

overheads imposed on processes for clock management must scale well with the number of participants and communication rates.

The sidebar “Message Tracing with Logical Clocks” reviews Lamport clocks and related approaches and summarizes how they don’t fulfill the previous requirements. Approaches that use more complex clocks to avoid inaccuracies can overcome some of these shortcomings. However, they typically do this by using “multiclocks” of the size of the number of interacting nodes. This also leads to problems in open systems, in which the membership of nodes varies over time.

This motivated us to conceive a message-tracing approach that exploits these WSN application characteristics for efficiency:

1. Nodes usually communicate with only a few other nodes, usually their neighbors or special nodes such as cluster heads or a base station.
2. We can use nodes’ local control-flow traces to infer message contents such as type and local ordering.
3. Messages usually aren’t lost and arrive in order, although incidents of lost or out-of-order messages must be handled.

In short, with our bilateral clocks, every process \( p_i \) uses a different, independent, logical clock for each communication destination \( p_j \) (thus exploiting characteristic 1 in the previous list). For presentation simplicity, we consider a node to run only one process. As with any kind of logical clock, such a clock is incremented on every send. However, \( p_i \) attaches to a sent message \( m \) only \( p_j \)’s clock corresponding to the destination of \( m \) (besides its identity \( p_i \)), thus minimizing overhead. Inversely, \( p_i \) maintains a clock for every \( p_j \) from which it receives messages, denoting the time stamp of the last received message. So, these clocks resemble the monotonically increasing counters that many real-life protocols such as TCP/IP use to deal with retransmissions or ensure FIFO delivery.

The use of such clocks limits the amount of piggybacked information while reducing the amount of logged information. Assuming that successful in-order transmission of the raw tracing overhead with our approach. “TinyTracer” refers to a limited implementation of the compression technique we present in the main article, which takes into account the most common compression patterns (\( x = 4 \)). “Online” refers to a straightforward application of an out-of-the-box technique for run-time compression. “Hybrid simple” uses our compression technique without an optimized data structure. “Prius” refers to the combination of our techniques.

References
FOCUS: INTERNETWARE AND BEYOND

MESSAGE TRACING WITH LOGICAL CLOCKS

Leslie Lamport introduced logical clocks in his seminal paper on time and event ordering in distributed systems. With Lamport clocks, every node \( p \) has an integer clock \( \ell \), which is managed according to three simple rules:

- Every time \( p \) performs a local action, it increments \( \ell \) (including upon message send).
- Every time \( p \) sends a message, it appends its clock’s current value to the message.
- Every time \( p \) receives a message, it takes the larger of the appended clock value \( \ell \) and its current \( \ell \), increases it by one, and adopts it as the new value \( \ell = \max(\ell, \ell + 1) \).

This mechanism captures all causal dependencies. If event \( e \) occurs before event \( e' \), then \( t(e) < t(e') \), where \( t(e) \) is the value of the clock at the process at which event \( e \) occurs. However, this mechanism introduces inaccuracies, in that \( t(e) < t(e') \) doesn’t imply that \( e \) actually happened before \( e' \). Figure D shows some shortcomings of Lamport-clock-based message tracing.

To overcome inaccuracies, vector clocks\(^5\) can precisely capture concurrent and causally related events. You can use vector clocks to identify racing messages; by recording only those messages, you can reduce trace sizes considerably and replay the traces.\(^6\) Although vector clocks avoid false positives, they require \( O(N) \) space, where \( N \) is the number of nodes in the network. This leads to huge memory bloat when logging events with their time stamps and to corresponding overhead for every message sent over the network.

Several researchers have tried to balance accuracy and overhead. Plausible clocks are constant-length vector clocks that collapse several processes’ events onto the same components.\(^7\) Plausible clocks of size \( 1 \) are equivalent to Lamport clocks; plausible clocks of size \( N \) are equivalent to vector clocks. These clocks are useful when accuracy is required for only some processes that use dedicated clock components. Adaptive plausible clocks allow this set to change over time.\(^8\)

Hierarchical clocks presuppose a hierarchical system breakdown to reduce complexity.\(^9\) If two clusters of \( N \) processes interact only through an intermediate process \( p \), you can use clocks of \( N + 1 \) and \( N \) size in the clusters, respectively. \( p \) translates between the clusters’ clocks. When event \( e \) with time stamp \( t(e) \) is relayed from the first cluster to the second, \( p \) produces an event \( e' \) with a time stamp \( t(e') \) to the second cluster, keeping track of the association \( t(e') = t(e) \).

Plausible and hierarchical clocks can mitigate certain bottlenecks but still have clear limitations. The former collapse the clock spaces of several processes, thus decreasing accuracy, especially as a system scales up. The latter work only in a strict hierarchical system layout, restricting interaction.

messages is typical (characteristic 3), it suffices for a process \( p \) to log that it received a message from \( p' \). (The expected time stamp for the next message \( m' \) from a given process \( p \) is one larger than the last one from \( p \).) If the message doesn’t have the expected time stamp, the difference is also logged. We use a simple table to translate process identifiers \((p, p')\) to indices \((i, j)\), and storing the latter is sufficient. This inherent ability to compact traces is unique to bilateral clocks and in practice goes far beyond pure complexity improvements.

In addition, bilateral clocks reflect only send or receive events and not message contents (thus exploiting characteristic 2). Furthermore, processes only manage bilateral clocks for actual communication partners (thus exploiting characteristic 1). If this set changes over time, translation tables can be updated and the corresponding updates logged.

In contrast, using multicomponent clocks such as vector clocks to support sets of communicating processes that change over time is much harder. This is because those clocks require global agreement on mappings between processes and components.

For more on bilateral clocks’ benefits in debugging, see the related sidebar.

Trace Compression

In resource-constrained WSNs, every bit counts. We further exploit the inherent strong potential of bilateral clocks for compact representation of node interaction by using alias tables. Here, each node manages a small table of correspondences between unique identifiers \( i \) and the corresponding node or process identifiers \( p \) of interaction peers. This allows for receive and send events to be logged only with a small identifier \( i \). Changes in a given node’s communication peers can be easily reflected by updating the table (including index shifting) and logging the corresponding updates. Using individual clock spaces, as opposed to global clock spaces in other logical-clock approaches, lets us confine such changes to only the affected nodes, without needing global agreement.
REFERENCES


FIGURE D. Examples showing the shortcomings of Lamport-clock-based message tracing when pairing message receive events with the corresponding send events in the presence of unreliable channels or arbitrary local purging of traces. The traces from processes P0 and P1 appear below the space–time representation of processes. (1) A simple example with the same trace being generated in cases in which messages arrive in order (the left image) and out of order (the right image). It’s impossible to correctly identify out-of-order arrivals from the trace during postmortem analysis, implying that the receive events can’t be paired with the corresponding sends. (2) Examples in which it’s impossible to identify which message was lost from the trace during postmortem analysis.

We’ve also devised an effective technique to compress traces on individual nodes. Compression is a natural way to mitigate tracing overhead. However, WSNs’ extreme resource constraints pose novel challenges for compression.

Existing compression algorithms are either inapplicable or require adaptation to satisfy the limits on memory and CPU resources. Adaptations of these algorithms still perform poorly for WSN traces owing to inherently small input buffers. These buffers are only a few hundred bytes in WSNs, leading to few opportunities for learning and replacing the repeating patterns.

WSNs have inherently small input buffers for two reasons. First, because traces are constantly generated with the execution, they must be buffered in RAM before compression. Otherwise, computation-intensive compression might interfere with trace generation. After a trace buffer is compressed, the compressed output must be buffered in RAM as storage into nonvolatile flash, or wireless transmission will be slow. Owing to the differences in execution speed between trace generation, compression, and transmission or storage, multiple buffers are needed. The small RAM (4 to 10 Kbytes) and the requirement for multiple buffers limit the individual trace buffers to a few hundred bytes. Second, reliable delivery of large buffers over an unreliable wireless multihop network in WSNs is expensive.

Existing WSN compression algorithms also perform relatively poorly because they can’t exploit the rich repetitions in traces. This is because compression occurs independently on small buffers, as we explained earlier. Compression algorithms’ poor performance explains why early WSN tracing approaches either used simple, ad hoc compression or didn’t compress at all.

Our approach, dubbed Prius after the Toyota hybrid car,
relies on three key observations:

- WSN computations exhibit much repetition in a short time.
- The repetitive patterns in WSN
BILATERAL CLOCKS’ BENEFITS

Bilateral clocks fulfill the seemingly contradictory requirements mentioned in the section “Debugging across Nodes” in the main article by separating a global clock space into spaces between process pairs.

With bilateral clocks, every node $p_i$ has integer clocks $s_{ij}$ and $r_{ij}$ for its communication with $p_j$, which is managed according to two simple rules:

- Every time $p_i$ sends a message $m$ to $p_j$, it increments $s_{ij}$, logs the sending of $m$ to $p_j$, and appends the current value of $s_{ij}$ to $m$.
- Every time $p_i$ receives a message $m$ from $p_j$, it logs the reception of $m$ along with any difference between the clock $l$ piggybacked by $m$ and the identifier of the next expected message $r_j$ from $p_j$ ($l - r_j$). Then, it sets $r_{ij}$ to the value of $l$ augmented by $1$ ($r_{ij} = l + 1$).

Periodically, any process $p_i$ will log the absolute values of its counters $s_{ij}$ and $r_{ij}$ for all its communication partners $p_j$. From such a local “snapshot,” a process can recover the actual counters for all sent and received messages. This way, sends and receives can be paired accurately despite losses and reorderings. If all communications are successful and in order, the logged information is minimal. Local events are logged in between send and receive events in the same traces for dealing with the local ordering of events. There’s no need to include these events in communicated clocks.

Figure E demonstrates how bilateral clocks allow for dealing with out-of-order arrivals and message losses, using the scenarios in the sidebar “Message Tracing with Logical Clocks.”

We assume that when we reason about a set of processes (for example, when replaying their execution), traces of events on all these processes are available. Otherwise, we can’t reason about them. So, a process $p_i$ needn’t know, when receiving a message $m$ from $p_j$, which messages $p_i$ received (for example, from some $p_j$) before it sent $m$. As long as the send event on $p_j$ corresponding to the reception of $m$ on $p_i$ can be correctly identified, we can rely on information gathered at $p_i$ to determine which messages it received from others before sending $m$.

With vector clocks (see the sidebar “Message Tracing with Logical Clocks”), this information is readily available in the clock attached to $m$ by $p_i$ and is transferred to $p_i$’s time stamp of $m$’s reception. With bilateral clocks, this information can be computed when or if it matters, shifting overhead from run time to an as-needed basis, which for debug-

computations evolve only a little over time.

- WSN nodes use the Harvard architecture and thus have separate program memory (EEPROM) and data memory (SRAM). The program memory is extremely scarce, and the data memory has more generous constraints and is rarely a bottleneck.

So, Prius employs offline training to capture the repetitive patterns of WSNs that occur in the traces. It includes those patterns in the program memory using specially adapted data structures. The compression algorithm then uses these patterns to perform online compression. Although rather intuitive,
our approach is based on a careful balance, which our evaluation validated. Program memory can accommodate more patterns than data memory and thus potentially improve the compression ratio. However, accessing program memory is typically 1.5 times more costly than accessing data memory in both Atmel’s AVR and Texas Instruments’ MSP430 architectures.

Specialized data structures can counterbalance this increase by simplifying lookups considerably, but this doesn’t support run-time addition of patterns. Such missing patterns might reduce compression performance. The energy savings obtained through higher compression rates outweigh the additional CPU costs. Furthermore, missing patterns resulting and other applications occurs offline. The separation of clock spaces with respect to Lamport clocks—besides supporting accuracy—improves the compressibility of traces on processes. Table A presents an overview of complexities and other salient features of different logical-clock approaches.

Figure F empirically demonstrates bilateral clocks’ benefits through lower energy consumption overhead for tracing on sensor nodes, compared to Lamport clocks augmented with sender identifiers (to break ties). The overhead is for the Oscil TinyOS benchmark.

Reference
HYBRIDIZING SLZW

Here we explain our hybridization approach’s high-level design with respect to an abstract compression algorithm, A. First, we develop two modified variants of the algorithm: \( A_{\text{leaves}} \) and \( A_{\text{compressor}} \).

Given an uncompressed trace, \( A_{\text{leaves}} \) outputs the internal data structure used for compression to a file and compresses the input like \( A \). Depending on the compression algorithm, the internal data structure could be a table, dictionary, or grammar. We then design an efficient data structure that exploits the static nature of the stored patterns to reduce access time, storage space, or both. A data structure generator will take the output of \( A_{\text{leaves}} \) and encode the designed data structure in a header file.

\( A_{\text{compressor}} \) is the version of \( A \) adapted to run on motes. It includes the header file and an interface with the designed data structure instead of the one used in \( A \).

Consider LZW (Lempel–Ziv–Welch), a compression algorithm that builds a dictionary of repetitive patterns while scanning the input. If LZW doesn’t find patterns (or strings) in input in the dictionary, it adds them to the dictionary. It replaces (encodes) the input patterns found in the dictionary with indices to the dictionary. Because a pattern can be a prefix of other patterns, LZW continues the pattern search until it finds the longest pattern before encoding.

Implementing LZW in wireless sensor networks isn’t straightforward—especially maintaining a dictionary and looking up arbitrarily long patterns. SLZW (Sensor LZW) is an implementation with an array-based data structure. Each entry in the array is a tuple (value, next, miss):

- value stores the input character,
- next stores the pointer to the next entry in a pattern, and
- miss refers to a new entry to further look for a matching pattern when the current pattern doesn’t match.

The dictionary initially contains 256 entries, with each entry’s value corresponding to its index, and the next and miss pointers are initialized to 0.

The hybridization of LZW involves designing an efficient data structure for the dictionary to be stored in the program memory and then creating the hybrid. The latter is simple for LZW because the only change needed to it is to omit addition to the dictionary. The procedure for lookup in the dictionary must be rewritten to access the table from the program memory.

The SLZW data structure has two main drawbacks for hybrid compression. First, because the patterns are known, the next and miss pointers storing 0 are unnecessary because no more patterns will have to be stored. Second, when several patterns have common prefixes, a pattern’s lookup cost grows with the number of successors, which are patterns with a common prefix but different current entries.

To deal with these drawbacks, we employ a prefix tree, or trie. A trie is an ordered tree data structure that stores an associative array. The trie’s edges represent the input characters; its nodes represent the encoded dictionary values.

We can exploit the fact that the LZW dictionary is static for better performance. First, because successors are known beforehand, only pointers to those successors are stored at any given node. This avoids miss pointers or a next pointer with null values used in the array-based data structure we described earlier. Second, we store the successor edges in ascending order to enable faster lookups using binary search.

A trie can be compactly encoded (or tightly packed) in memory. Such compact tries allow faster lookup of successors than with array-based data structures by doing a binary search on the children at a given node. Binary search is possible because the children can be stored at fixed offsets from each other, allowing random access. There are several ways to tightly pack a read-only trie in memory; we used Ulrich Germann and his colleagues’ efficient encoding.3

The approach of hybridizing a compression algorithm is further elaborated in “Prius: Hybrid Trace Compression for Wireless Sensor Networks”4 and exemplified by the FSM (finite state machine) compression algorithm.5

References
are rare because WSN executions are repetitive and don’t evolve much. We can handle more substantial changes in execution patterns arising from reprogramming a WSN by uploading a new set of patterns for the software’s latest version. In the sidebar “Hybridizing SLZW,” we describe a popular WSN compression algorithm and illustrate Prius hybridization through it.

Our debugging technologies are being exploited by SensorHound (www.sensorhound.com), a startup company specializing in the IoT’s reliability and security.

The need for resource-efficient technologies will persist. The history of computing has shown that as soon as we can pack more resources in less space, one of two things happens. We either try to exploit this to make existing devices smaller and more ubiquitous, or we try to deploy more complex functionality and software on the now more powerful devices.

Acknowledgments
US National Science Foundation (NSF) Computer Systems Research grant 0834529 and NSF Small Business Innovations Research grant 1345940 supported this research.

References

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.
Automated Synthesis of Service Choreographies

Marco Autili, Paola Inverardi, and Massimo Tivoli, University of L’Aquila

// Service choreographies describe peer-to-peer message exchanges among participant services from a global perspective. A proposed tool automatically synthesizes choreographies and is effective in practical contexts. //

Today’s service composition mechanisms are based mostly on service orchestration, a centralized approach to composing multiple services into a larger application.¹,² Orchestration works well in rather static environments with predefined services and minimal environment changes. These assumptions are inadequate in the Future Internet vision, in which many diverse service providers and consumers keep changing and can’t be coordinated through a centralized approach.

In contrast, service choreography is a decentralized approach that provides a looser way to design service composition by specifying participants and message protocols between them. The need for service choreography was recognized in BPMN2 (Business Process Model and Notation Version 2.0; www.omg.org/spec/BPMN/2.0), which introduced choreography-modeling constructs. Service choreographies model peer-to-peer communication by defining a multiparty protocol that, when put in place by the cooperating parties, allows reaching the overall choreography objectives in a fully distributed way. In this sense, service choreographies differ significantly from service orchestrations, in which one stakeholder centrally determines how to reach an objective through cooperation with other services.

Future software systems won’t be realized by orchestration only; they’ll also require choreographies. Indeed, services will be increasingly active entities that, communicating peer-to-peer, proactively make decisions and autonomously perform tasks according to their own imminent needs and the emergent global collaboration.

In a distributed setting, obtaining the coordination logic required to realize a choreography is nontrivial and error prone. So, automatic support for realizing choreographies is needed. For this purpose, we developed CHOReOSynt (choreos.disim.univaq.it), a choreography synthesis tool. As you’ll see, CHOReOSynt advances the state of the art in automating choreography realizability enforcement.

Realizing Choreographies: The Problems

Choreography-based systems usually consider two problems:
Second, it supports the whole CD tem, thus avoiding state explosion. A centralized model of the whole sys-
duces CDs without generating a abilities. First, it automatically pro-
structure in a
nate the service interactions to real-
co-ordination delegates (CDs). When interposed among the participant
vices are active entities executing
butionary services and stakehold-
ables, and stakehold-
s, and stakehold-
vices, CDs “proxify” and coordi-
ate the service interactions to real-
Service choreography is a decentralized approach that provides a looser way to design service composition.

The Reference Scenario
We experimented with CHOReO-
Synt using a real-world reference
scenario that requires the complex
cooration and sales sub-
chooregaphy of a CRM system that
iters among the participant
vices, CDs “proxify” and coordi-
ate the service interactions to real-
CHOReOSynt has three unique
bies. First, it automatically pro-
udes CDs without generating a
alized way.

Our Solution
We developed CHOReOSynt as part of
es and sales sub-
chooregaphy of a CRM system that
i monitors the activity of a customer
the Client in Figure 1) in a store to
pose customized shopping offers
and advertisements. By exploiting
the In-Store Advertisement Totem
vice cooperates with the Market-
ing Manager service to manage the
offers of products in the store.

The choreography is triggered
when the Client enters the store. The
Shop Entrance service detects
the Client’s presence, assigning her
a cart and notifying the Marketing
Application about the new Cli-

Figure 1 shows a BPMN2 cho-
ography diagram drawn with the
Eclipse BPMN2 Modeler (eclipse.
org/bpmn2-modeler). It concerns
the in-store marketing and sales sub-
chooregaphy of a CRM system that
iters among the participant
vices, CDs “proxify” and coordi-
ate the service interactions to real-
CHOReOSynt has three unique
bies. First, it automatically pro-
udes CDs without generating a
alized way.

To address these problems, re-
sarchers have proposed many ap-
aches (for a look at some ap-
aches, see the sidebar). However, to put choreography into practice, we must consider realizing service choreographies by reusing third-
party services. This leads to a fur-
ther problem concerning automatic realizability enforcement. That is,
given a choreography specification
and a set of services (discovered or
istered as possible participants), we
must coordinate their interaction
so as to fulfill the global collabora-
tion that the choreography specifi-
cation prescribed.

A realizability check determines whether the choreography can be realized by implementing each participant so that it conforms to the choreography role specifying its expected behavior. A conformance check determines whether the overall global interaction of a set of services satisfies the choreography.

The Reference Scenario
We experimented with CHOReO-
Synt using a real-world reference
scenario that requires the complex
coordination of business services,
thing-based services, and stakehold-
ers from the customer relationship
management (CRM) domain.

Figure 1 shows a BPMN2 cho-
ography diagram drawn with the
Eclipse BPMN2 Modeler (eclipse.
org/bpmn2-modeler). It concerns
the in-store marketing and sales sub-
chooregaphy of a CRM system that
iters among the participant
vices, CDs “proxify” and coordi-
ate the service interactions to real-
CHOReOSynt has three unique
bies. First, it automatically pro-
udes CDs without generating a
alized way.

To address these problems, re-
sarchers have proposed many ap-
aches (for a look at some ap-
aches, see the sidebar). However, to put choreography into practice, we must consider realizing service choreographies by reusing third-
party services. This leads to a fur-
ther problem concerning automatic realizability enforcement. That is,
given a choreography specification
and a set of services (discovered or
istered as possible participants), we
must coordinate their interaction
so as to fulfill the global collabora-
tion that the choreography specifi-
cation prescribed.

A realizability check determines whether the choreography can be realized by implementing each participant so that it conforms to the choreography role specifying its expected behavior. A conformance check determines whether the overall global interaction of a set of services satisfies the choreography.

The Reference Scenario
We experimented with CHOReO-
Synt using a real-world reference
scenario that requires the complex
coordination of business services,
thing-based services, and stakehold-
ers from the customer relationship
management (CRM) domain.

Figure 1 shows a BPMN2 cho-
ography diagram drawn with the
Eclipse BPMN2 Modeler (eclipse.
org/bpmn2-modeler). It concerns
the in-store marketing and sales sub-
chooregaphy of a CRM system that
iters among the participant
vices, CDs “proxify” and coordi-
ate the service interactions to real-
CHOReOSynt has three unique
bies. First, it automatically pro-
udes CDs without generating a
alized way.

To address these problems, re-
sarchers have proposed many ap-
aches (for a look at some ap-
aches, see the sidebar). However, to put choreography into practice, we must consider realizing service choreographies by reusing third-
party services. This leads to a fur-
ther problem concerning automatic realizability enforcement. That is,
given a choreography specification
and a set of services (discovered or
istered as possible participants), we
must coordinate their interaction
so as to fulfill the global collabora-
tion that the choreography specifi-
cation prescribed.

A realizability check determines whether the choreography can be realized by implementing each participant so that it conforms to the choreography role specifying its expected behavior. A conformance check determines whether the overall global interaction of a set of services satisfies the choreography.
concurrently and might not synchronize as the choreography prescribes.

So, the global collaboration might exhibit undesired interactions. That is, interactions not allowed by the choreography specification can happen when the services collaborate in an uncontrolled way. For instance, the Client is allowed to perform the Add Product task to add products to the Smart Cart (see the top of the Figure 1). However, after paying and before check-out, the Client might try to add products (see the top-most tasks just before the End Event), thus avoiding paying for them.

This scenario reveals that implementing the required coordination logic is nontrivial and error prone in a distributed setting. So, automatic support for realizing correct choreographies is desirable.

**Choreography Realization**

Using the previous scenario, Figure 2 diagrams our automatic choreography synthesis.
**Step 1.** Software producers cooperate with domain experts and business managers to

- set the business goal (for example, assist travellers from arrival, to staying, to departure),
- identify the tasks and participants required to achieve the goal (for example, reserving a taxi from the local taxi company, purchasing digital tickets at the train station, and performing transactions through services based on near-field communication in a shop), and
- specify how participants must collaborate through a BPMN2 choreography diagram.

To support this step, CHOReOS provides a plug-in that allows importing the goal specification into the MagicDraw modeling tool (www.nomagic.com) and associates it with BPMN2 constructs and quality-of-service constraints. In particular, CHOReOS uses both the Q4BPMN notation—an extension to BPMN2—to specify nonfunctional properties and dedicated automated tools to assess the choreography specification’s quality.

**Step 2.** MagicDraw exports the modeled choreography to CHOReOSynt. CHOReOSynt supports the XML-based encoding of BPMN2 choreographies, such as the one of the BPMN2 Modeler.

**Step 3.** CHOReOSynt queries the registry to discover services suitable for playing the choreography’s roles. The registry contains services published by providers (for example, transportation companies and airport retailers) that have identified business opportunities in the domain of interest. To describe service interfaces, CHOReOSynt uses WSDL (Web Services Description Language; www.w3.org/TR/wSDL). To describe service interaction behavior, BPEL (Business Process Execution Language) specifies the flow of messages exchanged with the environment. The registry also contains the registration of users interested in exploiting the choreography through their mobile apps.

**Step 4.** Starting from the choreography diagram and the set of discovered services, CHOReOSynt synthesizes a set of CDs. The synthesis exploits model transformations. The transformations are implemented through ATL (www.eclipse.org/atl), a domain-specific language for realizing model-to-model (M2M) transformations. ATL transformations comprise a number of rules, each of which manages a specific BPMN2 modeling construct. The current implementation of these transformations in CHOReOSynt (available at...
FOCUS: INTERNETWARE AND BEYOND

THE STATE OF THE ART AND BEYOND

CHOReOSynt (see the main article) is related to several approaches developed for service-oriented engineering. For space reasons, we discuss only the approaches closest to automated choreography enforcement.

Matthias Güdemann and his colleagues’ approach enforces a choreography’s realizability by automatically generating monitors. Each monitor acts as a local controller for its peer. This approach obtains monitors by iterating equivalence-checking steps between two centralized models of the whole system. It produces one of the models by composing the peer labeled transition systems (LTSs) assuming synchronous communication. It produces the other by composing the peer LTSs assuming asynchronous communication.

Güdemann and his colleagues’ monitor is similar to our coordination delegate (CD). However, our approach synthesizes CDs without producing a centralized model of the whole system, hence preventing state explosion. Furthermore, Güdemann and his colleagues’ approach is more theoretical and generates only the model of the monitors. In contrast, our approach synthesizes both the actual code implementing the CDs and their deployment schema.

Raman Kazhamiakin and Marco Pistore’s approach checks the conformance between the choreography specification and the composition of participant implementations. Their framework can model and analyze compositions in which the interactions can also be asynchronous and the messages can be stored in unbounded queues and reordered if needed. Following this line of research, Kazhamiakin and Pistore provided a hierarchy of realizability notions that forms the basis for a more flexible analysis regarding classic realizability checks. These two approaches are novel in that they characterize relevant properties to check a certain degree of realizability. However, they statically check realizability and don’t automatically enforce it at run time.

The ASTRÖ toolset supports automated composition of Web services and the monitoring of their execution. It aims to compose a service starting from a business requirement and the description of the protocols defining available external services. More specifically, a planner component automatically synthesizes the code of a centralized process that achieves the business requirement by interacting with the available external services. Unlike our approach, ASTRÖ deals with centralized orchestration-based business processes rather than fully decentralized choreography-based ones.

The CIGAR (Concurrent and Interleaving Goal and Activity Recognition) framework aims for multigoal recognition. CIGAR decomposes an observed sequence of multigoal activities into a set of action sequences, one for each goal, specifying whether a goal is active in a specific action. Although such goal decomposition somewhat recalls CHOReOSynt’s choreography decentralization, goal recognition represents a fundamentally different problem regarding realizability en-

choeros.disim.univaq.it) extends and advances their preliminary version.

Step 5. The generated CDs, together with the description of the services, serve as input to the enactment engine for deployment. CD deployment descriptors are codified in XML.

Step 6. As we mentioned before, CDs are interposed among the participant services needing coordination. CDs coordinate the services’ interaction such that the resulting collaboration realizes the specified choreography. To achieve correct coordination, CDs exchange additional communication (codified in XML) at run time to prevent undesired interactions. The coordination logic embedded in CDs is obtained by a distributed coordination algorithm implemented in Java; each CD runs its own instance of the algorithm. Once deployed by the enactment engine, CDs support the correct execution of the choreography by realizing the required distributed coordination logic among the discovered services.

The Synthesis Processor
As Figure 3 shows, the CHOReOSynt architecture comprises four REST (Representational State Transfer) services that perform the synthesis-time activities in Figure 2. The figure also shows CHOReOSynt’s main functionalities.

The M2M transformer. The M2M transformer offers a REST operation bpmn2clts(), which implements an ATL-based transformation. It takes as input the BPMN2 choreography specification and automatically generates a choreography labeled transition system (CLTS). A CLTS is a finite-state automaton that, for coordination purposes, is suitably extended with fork and
forcement. That is, goal recognition concerns learning a goal-based model of an agent by observing the agent’s actions while interacting with the environment. In contrast, realizability enforcement produces a decentralized coordination logic out of a task-based specification of the choreography.

Given a set of candidate services offering the desired functionalities, the TCP-Compose* algorithm identifies the set of composite services that best fit the user-specified qualitative preferences over nonfunctional attributes.\(^6\) CHOReOSynt could exploit this research to extend the discovery process to enable more flexible selection of services from the registry.

The research we described in the main article is an advance over our previous research.\(^7,8\) Although the synthesis process described in our previous research treated most BPMN2 (Business Process Model and Notation Version 2.0) constructs, it considered a simplified version of their actual semantics. For instance, as in Güdemann and his colleagues’ research, the selection of conditional branches was simply abstracted as a nondeterministic choice, regardless of the run-time evaluation of their enabling conditions. Analogously, the synthesis process enforced parallel flows by nondeterministically choosing one of their linearizations obtained through interleaving, thus losing the actual degree of parallelism.

To overcome these limitations, CHOReOSynt relies on a choreography model that, being more expressive than the choreography model in CIGAR and TCP-Compose*, preserves the BPMN2 constructs’ actual semantics. Relying on a more expressive model lets us define a novel, more effective distributed coordination algorithm.

References

join constructs, conditional branching, and conditional loops (typical BPMN2 constructs). The CLTS provides CHOReOSynt with a formal choreography model independent from the specific choreography modeling notation. This means you can use CHOReOSynt in different practical contexts in which different modeling notations might be adopted, provided you’ve implemented a dedicated M2M transformation.

Starting from the CLTS, the M2M transformer extracts the participant list, using extractParticipants(), and derives the CLTSs modeling each participant’s expected behavior. To produce the participant-specific CLTS, the transformer projects the choreography onto the participant’s role, thus filtering out the transitions and related states not belonging to the role description. The participant-specific CLTS models the interaction behavior that a candidate service (to be discovered out of the registry) must support to play the participant’s role.

The discovery manager. The synthesis process and the discovery process interact to retrieve, from the registry, candidate services suitable for playing the required participant roles—those services whose behaviors are compatible with the participant-specific CLTSs. For each participant, the discovery manager interacts with the extensible service discovery (XSD) system by invoking discoverServices(), which takes the participant-specific CLTS as input.

The behavior simulator. Using simulation[], the behavior simulator selects the candidate services that can behaviorally simulate a participant-specific CLTS. It takes as input the participant-specific CLTSs and the BPEL description of the service retrieved by the XSD service. It
Focusing on INTERNETWARE AND BEYOND

CHOREOSynt implements a form of LTS simulation extended to treat CLTSs and BPEL. That is, a BPEL description simulates a participant-specific CLTS if the former specifies—at least—all the message flows modeled by the latter. This means that the discovered service’s behavior must cover the participant’s expected behavior.

The CD generator. After the services have been selected, the CD generator generates the CDs, using `generateCD()`. This produces an executable description of the choreography to pass to the enactment engine through `createChorSpec()`. The enactment engine takes as input the selected services and the CDs generated for them. In particular, the choreography executable description is an XML-based declarative description specifying the selected services’ locations, the generated CDs’ locations, and the service–CD and CD–CD interdependencies.

The CHOREOSynt Eclipse Plug-in

CHOREOSynt has two modalities. The user might choose whether to execute the automatic synthesis processor or interactive synthesis processor. The former produces all the artifacts in one step. The latter produces them step by step and visualizes them by using a graphical editor we developed that’s based on GMF (Graphical Modeling Project; www.eclipse.org/modeling/gmp). The synthesis plug-in can be automatically installed into the Eclipse platform through the update site, http://choreos.disim.univaq.it/updatesite/site.xml. A video demonstrating CHOREOSynt at work on the reference scenario is at choreos.disim.univaq.it/downloads.

Our experiments demonstrated that CHOREOSynt can be effectively applied in practical contexts. In particular, they show that considering domain-specific interaction patterns mitigates the complexity of coordination enforceability when recurrent business protocols must be enforced. Generally, choreography synthesis is difficult in that not all possible collaborations can be automatically realized. This suggests we could improve CHOREOSynt with a combination of domain-specific choreography patterns, as well as protocol interaction patterns that correspond to service collaborations that are tractable through exogenous coordination. This approach would also let us produce parameterized coordination patterns offline, which could then be instantiated at run time.

Currently, CHOREOSynt supports pure coordination. It doesn’t deal with heterogenous interaction protocol adaptation because it doesn’t account for mismatches at the level of service operations and related I/O parameter types. To support data-based coordination through the elicitation and application of complex data mappings, CHOREOSynt should be enhanced to automatically infer mappings to match the data types of messages sent or received by mismatching participant services. This
means effectively coping with heterogeneous service interfaces and dealing with as many enterprise integration patterns and protocol mediation patterns as possible, in a fully automatic way. Toward that end, we achieved promising results in automated synthesis of modular mediators.

We want to enable the market acceptance and further enhancement of CHOREosSynt by third-party developers, especially small and medium enterprises, including development of applications for commercialization. So, we released CHOREosSynt under the umbrella of the Future Internet Software and Services Initiative (FISSI: www.ow2.org/view/Future_Internet/). Using a market-oriented approach, FISSI aims to develop awareness of OW2 Future Internet software in both FISSI members and nonmembers and both open source vendors and proprietary vendors. Our primary objective, to be achieved in the near future, is to establish a community of developers and third-party market stakeholders (for example, users, application vendors, and policy makers) around CHOREosSynt.

Acknowledgments
This research is supported by the Italian Ministry of Education, Universities and Research, project 2012E47TM2 (Integrated Design and Evolution of Adaptive Systems), and by the European Community (Seventh Framework Program, 2007–2013) under grant agreement 257178 (the CHOREos project).

References
Stigmergy-Based Construction of Internetware Artifacts

Wei Zhang and Haiyan Zhao, Peking University
Yi Jiang, Nanjing University of Aeronautics and Astronautics
Zhi Jin, Peking University

// Employing incremental graph superimposition, a collective of Internet-connected individuals collaboratively and continually construct and evolve a collective-level graph by incrementally aggregating the individuals’ working results. //</p>

IN THE PAST DECADE, software development has witnessed two important trends. The first is model-driven development.1,2 More and more software artifacts are represented by models; software development can thus be understood as the process of building a set of related models. This trend reflects a more concise understanding of software’s internal elements.

The second trend is Internet-based development.3,4 Developers are organized not in traditional localized and hierarchical ways but increasingly in distributed and flat ways supported by the Internet. This trend reflects the exploration of new ways to deal with the increasing software complexity brought by the Internet.

The Internet’s influence on software involves not only new ways to deliver services but also, and more important, new requirements (for example, autonomy, situation awareness, evolution, emergence, and trustworthiness). Hong Mei and his colleagues called software that satisfies those Internet-imposed requirements Internetware (software suitable for Internet-based run-time environments).5 Regarding the evolution requirement, Internetware should continually evolve to adapt to the rapid changes in both Internet-based run-time environments and user requirements.

Generally, developers can enable Internetware evolution in two ways. The first is to give Internetware the intelligence to evolve itself. The second way is to create methods that support the continual evolution of Internetware artifacts (artifacts constituting Internetware or produced in the development of Internetware), so that Internetware can be evolved. Although the first way seems perfect, it’s currently not realistic, considering the state of the art of artificial intelligence. The second way isn’t as perfect as the first way, but its feasibility is gradually increasing with the growth of Internet-based development. In the context of the Internet, it’s much easier to form large, dynamic, and continually evolving communities of stakeholders. If appropriately managed, such a community’s aggregated efforts will lead to the emergence of continually evolving artifacts. The success of open source software6 and software crowdsourcing7 points to the second way’s feasibility.

We’ve developed a stigmergy-based approach that combines model-driven and Internet-based development to continually construct and evolve Internetware artifacts.
Stigmergy is the process that produces collective intelligence in social insects. Despite the success of open source software and software crowdsourcing, those methods provide little support for model-driven development. One important reason is that artifact management mechanisms such as CVS (Concurrent Versions System), SVN (Apache Subversion), or Git aim to manage artifacts with a streaming structure (a typical example being source code). In contrast, model-based artifacts usually have a more complex conceptual structure. Although any artifacts are certainly represented by streaming documents at the storage level, the management of model-based artifacts at this level lacks the necessary conciseness, explicitness, and effectiveness. Our approach aims to remedy that situation.

Software Models
Model-driven development builds a set of related models in different phases. Usually, it builds conceptual models in the requirements phase and transforms them into implementation-dependent models in subsequent phases.

Although different models have different concerns, most of them have a graph structure, including a set of vertices and edges. Figure 1a shows a conceptual model represented as a class diagram, with the vertices indicating classes and the edges indicating the relationships between classes. Figure 1b shows a process model represented as an activity diagram, with the vertices indicating activities, decisions, and the initial and final nodes, and the edges indicating flows. This structure lets us treat different kinds of software models in a unified way.

Collective Intelligence and Stigmergy
Collective intelligence in social insects has been puzzling scientists for a long time. To explain its mechanisms, in 1959, Pierre-Paul Grassé introduced the concept of stigmergy (from the Greek stigma, which means mark, and ergon, which means work).

Stigmergy has two components: a collective of individuals and an environment (see Figure 2). The relationship between the individuals and environment is embodied in the individuals’ work process. This comprises exploring the environment and, on the basis of the perceived information (stimuli), taking actions to modify the environment (leaving new information in the environment). The relationship between individuals is embodied in the environment-mediated indirect interaction. Information that
one individual leaves in the environment could be perceived by another individual and stimulate that individual to leave new information. On the basis of these two relationships, collective intelligence emerges as the result of the positive feedback loop formed by massive numbers of individual behaviors.8

Stigmergy explicitly identifies the environment as a basic component of collective intelligence. The environment's function is twofold: to enable the environment-mediated indirect interaction between individuals and to contain the working results of the collective.

With the Internet’s rapid growth, Web-enabled collective intelligence is emerging.9,10 A large, loosely organized collective of people, connected by the Internet and mediated by computer-supported Web environments, does things collectively and intelligently, in either explicit or implicit ways. For example, in Wikipedia, thousands of editors are building an evolving encyclopedia, collectively and explicitly.11 In reCAPTCHA, over one billion people are helping to digitize traditional books, collectively and implicitly.12 Despite the differences in individuals and environments, stigmergy keeps its value in Web-enabled collective intelligence.

**Stigmergy-Based Artifact Construction**

Stigmergy provides a general way for a large collective of individuals to collaboratively construct and evolve certain artifacts in a shared environment. Assuming that most software models have a graph structure, we designed **incremental graph superimposition** (IGS) to support the construction and evolution of model-based Internetware artifacts.

### Incremental Graph Superimposition

As in stigmergy, IGS involves an environment and a collective of individuals (see Figure 3). Individuals can explore and leave information in the environment by taking actions. Specifically, the environment contains a collective graph produced by the collective, and each individual is associated with an individual graph.

The collective graph consists of a set of elements (vertices and edges); each element has an integer indicating the number of references (NOR) to that element (how many individuals referenced the element in their individual graphs). An individual graph consists of a set of references to elements in the collective graph and thus is a subgraph of that graph. This subgraph relationship forms as the result of individuals’ actions.

Individuals can take four kinds of actions:

- **View.** An individual views a set of elements in the collective graph (perceives stimuli), according to clues the individual provided. For example, when an individual tries to create a vertex with the name “student,” once he or she has inputted a substring (for example, “stu”), IGS displays to this individual a set of vertices with similar names.

- **Reference.** An individual references an element he or she hasn’t previously referenced. That element becomes an element of the

---

**FIGURE 2.** Components and relationships in stigmergy. Collective intelligence emerges as the result of the positive feedback loop formed by massive numbers of individual behaviors.
individual's individual graph, and the element's NOR increases by 1.

- **Create.** An individual adds a new element (with 0 as its initial NOR) into the collective graph and then references this element.

- **Unreference.** An individual cancels his or her previous reference to an element—for example, when the individual thinks this reference is a mistake or is no longer appropriate. IGS then removes this element from the individual's graph, and its NOR decreases by 1.

The collective graph is constructed incrementally in real time. When an individual takes an action, IGS immediately superimposes the effect on the collective graph.

Figure 4 illustrates an example of IGS. Figure 4a shows four individuals taking six sets of actions at six time points. Consider individual A. Before time t0, the collective graph is empty; at t0, A creates vertices O, P, and Q and edges <O, P> and <O, Q>. At t5, A references vertices R and S and edges <O, R> and <O, S>.

Figure 4b shows how the collective graph is constructed and evolved by incrementally superimposing the actions' effects on it. If we treat an individual's working result at a time point as a graph, the collective graph is the superimposition of all these graphs at previous time points. For example, between t4 and t5, the collective graph is the superimposition of the five graphs at the previous five time points.

As Figure 1 indicates, as long as individuals take create, reference, or unreference actions, the collective graph will evolve continually.

**An Example: The Conceptual Model**

To provide a more concrete understanding of IGS, let’s take the conceptual model (CM) as an example. A CM usually includes a set of concepts and relationships in a problem domain, represented by different notations (for example, an entity–relation diagram or class diagram).

In the context of CMs, an individual's individual graph is incarnated as the **individual CM (ICM)**, which reflects the individual's understanding of the problem domain. The collective graph is the superimposition of all individuals' ICMs—the **collective CM (CCM)**.

An ICM and the CCM differ in two ways. First, an ICM has some basic consistency properties; for example, a concept has exactly one name. These properties don’t hold in the CCM; for example, a concept in the CCM might have several names referenced by different sets of individuals.

**FIGURE 3. Concepts and relationships in incremental graph superimposition (IGS). The environment contains a collective graph produced by the collective, and each individual is associated with an individual graph. OCL stands for Object Constraint Language.**
Second, an ICM is easier to understand. Understanding an ICM is like listening to one individual; understanding the CCM is like listening to a collective of individuals speaking simultaneously.

**Case Studies**

To observe IGS’s practical effects, we developed a Web environment for IGS-based CM construction, deployed it on the Internet, and conducted case studies. These case studies are ongoing, and anyone can participate.

**The Web Environment**

In this environment, class diagrams represent CMs. (You can access this environment at http://115.28.38.84. After registering, you can participate in existing projects or create new projects.)

When entering a project for the first time, a user sees an empty page (see Figure 5a). On it, the user can maintain his or her ICM by taking actions. When the user wants to create a class and inputs a class name or just a substring of this name, a dropdown list recommends similar class names in the CCM (see Figure 5b). In this way, the user explores the CCM (perceives stimuli) implicitly and incrementally. After the user selects a name from the list, the interface lists classes with the same name in the CCM; the user can reference one of them (see Figure 5c). When

---

**FIGURE 4.** An example of IGS. (a) Four individuals taking six sets of actions at six time points. (b) The collective graph’s construction and evolution through incremental superimposition of the actions’ effects. As long as individuals take create, reference, or unreference actions, the collective graph will evolve continually.
the user wants to create a relationship between two classes, the interface recommends existing relationships between the two concepts (see Figure 5d). If the user selects a relationship, the interface lists its properties (see Figure 5e). After the user takes some actions, the interface displays the classes and relationships in the user’s ICM (see Figure 5f).

IGS is a basic mechanism that’s transparent to participants. For example, in this environment, users can’t directly see concepts in IGS; they see only concepts related to CMs (for example, classes, relationships between classes, and class or relationship attributes).

Current Results
We created three projects for three problem domains: a CD store, course management, and train ticketing. Fifteen people (two associate professors, two PhD students, nine master’s students, one undergraduate student, and one enterprise developer) constructed ICMs for the projects. (Six ICMs were completed in about one month; nine were completed in one week.)

Take the course management project as an example. In it, the 15 participants created more than 1,700 elements. Figure 6 shows some results.

Figure 6a shows the statistics for each ICM, including the number of created and referenced elements and the number of times other individuals referenced those created elements.

Figure 6b shows a global view of the classes, their names, and their relationships. Red nodes are classes, blue nodes are class names, and squares connect class names to a class. A line connecting two classes denotes a relationship between them. The elements’ opacity is proportional to their NOR.

Figure 6c shows the concepts, attributes, and relationships that at least five individuals referenced. Rectangles with italic text denote classes; normal text denotes attributes. Numbers in brackets denote elements’ NOR.

Incentives for Participation
To be successful, IGS must attract a large collective of stakeholders. So, appropriate incentives should be offered to participants. Generally, IGS enables most participants to get high-quality artifacts that might be impossible for them to construct otherwise. That is, most participants benefit from participating. However, if this kind of benefit isn’t enough,
other incentives should be considered—for example, prizes in crowdsourcing or the pleasure from games with a purpose.13

### The Environment’s Dynamics

The environment’s dynamics decides how individuals’ working results aggregate into collective-level artifacts and how these artifacts evolve. More efficient dynamics could be explored, including more intelligent mechanisms for element recommendation,
duplicate-element detection and resolution, and useless-element detection and elimination.

**Measuring IGS**
Research could focus on three kinds of measurement:

- **efficiency measurement** of the collective, indicating the collaboration’s health;
- **quality measurement** of the collective artifacts, indicating the trustworthiness of the collective’s working results; and
- **contribution measurement** of individuals, indicating their contribution to the collective.

A more basic problem concerns the relationships between these kinds of measurement.

**Relevance to Practitioners**
IGS could support collaboration-based model or view merging (see the sidebar). It also could improve the state of the practice of open source software and software crowdsourcing.

Currently, in open source software development, the requirements are often scattered over emails or online-forum discussions, making it difficult for newcomers to easily understand the requirements. This is because the shared environments that open source software provides don’t have the appropriate dynamics to aggregate different concerns or views into a unified perspective. In this situation, if requirements can be represented as easy-to-understand models, IGS can support collaboration-based construction and evolution of unified requirements models.

IGS can also support competitive collaboration in software crowdsourcing. Usually, the prize in software crowdsourcing goes to only the one or two top-performing participants. So, software crowdsourcing is competition-intensive and lacks collaboration. Combined with reasonable contribution measurement, IGS could make software crowdsourcing both competition- and collaboration-intensive. A set of individuals could collaboratively construct the winning solution, and prizes would be awarded to each individual in proportion to his or her contribution.

**Model merging and view merging**

Model merging in model-driven development and view merging in conceptual modeling are closely related to incremental graph superimposition (IGS; see the main article). Like IGS, they both focus on constructing a complex model by merging a set of individual models reflecting stakeholders’ different views or concerns.

However, model merging and view merging differ fundamentally from IGS in three ways. First, they’re algorithm oriented, whereas IGS is human oriented. They aim to develop algorithms to identify the correspondences between individual models and merge them into a unified collective model. IGS aims to explore collaboration mechanisms so that the stakeholders’ mass collaboration produces a collective model.

Second, in model or view merging, the construction of individual models and of the collective model are two clearly separate sequential activities. After stakeholders complete the individual models, those models still need an additional activity to merge them. In IGS, the two activities are inseparable. When stakeholders complete their individual models, the collective model emerges simultaneously.

Finally, in model or view merging, stakeholders work in private environments and construct individual models without interaction or collaboration. In IGS, stakeholders work in a shared environment and construct individual models through environment-mediated indirect interaction or collaboration. That is, when constructing individual models, stakeholders can reference elements created by others from the environment. A side effect of these reference actions is clearly defined correspondences between individual models, which remain a challenge for model or view merging.

Despite these differences, we think that the two approaches are complementary. IGS could support collaboration-based model or view merging. Also, you could use algorithms that are effective in model or view merging to detect duplicate elements in the IGS collective model.

**References**

n 1987, Frederick Brooks made the “no silver bullet” assertion about software development.14 This assertion embodies deep insights but is based on two implicit assumptions that might not still hold:

• The development team is organized as a strict hierarchy.
• The viewpoint of human individuals is used to observe the essential difficulties of software.

If these two assumptions are violated, will this pessimistic assertion change? The answer has a high probability of being yes. For example, in Internet-based development, the stakeholders are organized in a flat manner, and complex software artifacts continually emerge. It seems the so-called essential difficulties of software aren’t a problem anymore from the viewpoint of a collective of Internet-connected stakeholders.

We believe that IGS can integrate model-driven and Internet-based development. As a result of this integration, model-based artifacts will be constructed and evolved by the Web-enabled collective intelligence that emerges from mass collaboration among Internet-connected stakeholders.

Acknowledgments
We thank the editors and anonymous reviewers for their valuable comments. This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201 and the National Natural Science Foundation of China under grants 91318301, 61432020, and 61272162.

References
Diagnosing Energy Efficiency and Performance for Mobile Internetware Applications

Yepang Liu, The Hong Kong University of Science and Technology
Chang Xu, Nanjing University
Shing-Chi Cheung, The Hong Kong University of Science and Technology

MOBILE INTERNETWARE applications seamlessly connect physical and cyber environments and provide smart services to users. However, unlike their desktop counterparts, smartphone applications run on resource-constrained platforms, so even small implementation inefficiencies can lead to a bad user experience. Ensuring a satisfactory user experience is a nontrivial task for developers, especially because many smartphone applications have these requirements:

- **Seamless communication.** Smartphone applications (such as email clients or shopping applications) often use the latest cloud data. Inefficient communication with Internet cloud services can easily waste valuable battery power and network bandwidth.
- **Frequent sensing.** Smartphone applications often use various sensors to detect users’ physical and cyber environments and thereby provide context-aware services (such as navigation). However, sensing operations consume considerable energy, and problematic use of sensors can easily waste energy.
- **Intensive computation.** Most smartphones have fully fledged operating systems, enabling computationally intensive applications (such as games). However, such computation places a big burden on smartphone GPUs and CPUs, and inefficient computation easily hurts application performance.

The smartphone application market is extremely competitive. Developers rarely have sufficient time or resources to carefully optimize their
applications’ energy efficiency and performance before pushing them to market. So, many applications suffer from energy and performance bugs.¹ The former can seriously waste battery power; the latter can significantly reduce smartphones’ responsiveness and available computational resources (such as memory and network bandwidth). These bugs severely impact the user experience and cause significant user frustration.²,³ Despite such bugs’ pervasiveness, both the research and industry communities haven’t sufficiently understood them. The result is a lack of mature tools to effectively help developers locate energy black holes and performance threats in their smartphone applications.

Here, we discuss the challenges in diagnosing energy and performance bugs in real-life Android applications. We hope to inspire further efforts to adequately address these challenges. We also review state-of-the-art diagnostic techniques and tools. In particular, we offer results from our case study, which applied a representative tool to popular commercial Android applications and the Samsung Mobile Software Development Kit (SDK). Our study results demonstrate that such tools are useful to and necessary for smartphone application developers; their feedback further suggests our research’s practical value.

### Activities

Our article focuses on the Android OS, which is Linux-based. Android applications are written primarily in Java, although, for performance reasons, developers might write critical parts using native languages (such as C).

Android applications typically comprise four types of components: *activity, service, broadcast receiver,* and *content provider.*¹ Each component follows a prescribed life cycle defining how it’s created, used, and destroyed. We describe the activity component here to aid the understanding of our later discussion. (For more on application components, see the official Android developer website, [http://developer.android.com](http://developer.android.com).)

**Application GUIs** are defined in activities. Figure 1 shows an activity’s life cycle. It starts after the `onCreate()` handler is called and ends after the `onDestroy()` handler is called. An activity’s foreground lifetime (its “running” state) starts after the `onResume()` handler is called and lasts until the `onPause()` handler is called. At that point, the activity goes to the background (its “stopped” state) and becomes invisible. An activity can interact with users only when it’s in the foreground. When it goes to the background, its `onStop()` handler is called. When users navigate back to a paused or stopped activity, the activity’s `onResume()` or `onRestart()` handler is called, and it returns to the foreground. In exceptional cases, a paused or stopped activity might be killed to release memory to higher-priority applications.

### Energy and Performance Bugs

Our recent studies identified that the following three common bug types seriously affect smartphone applications’ energy efficiency and performance:¹,⁴

- **Energy leak bugs** can quickly exhaust batteries.
- **GUI-lagging bugs** can significantly reduce application responsiveness.
- **Memory bloat bugs** can consume too much memory.

Here, we discuss representative examples of these bugs from real-world Android applications. For a more comprehensive discussion, along with additional bug patterns and examples, see our empirical studies.¹,⁴

---

**FIGURE 1.** The life cycle of an Android application’s activity component. Application GUIs are defined in activities. The activity’s life cycle starts when the `onCreate()` handler is called and ends after the `onDestroy()` handler is called.
Energy Leak Bugs

Many applications that communicate with the physical or cyber environment through sensors or network interfaces suffer from subtle energy leak bugs. One common cause is cost-ineffective use of sensory or network data.\(^4\)\(^5\)

Figure 2a shows a simplified version of a real-world bug.\(^4\) The associated application uses GPS data for navigation and location tracking (which can be disabled). When users launch the application, `MapActivity` (lines 1–25) starts, creating a live map for user interactions. For navigation, the application maintains a long-running

1. public class MapActivity extends Activity {
2. private Intent gpsIntent;
3. private BroadcastReceiver myReceiver;
4. public void onCreate() {
5. gpsIntent = new Intent(GPSService.class);
6. startService(gpsIntent); //start GPSService
7. myReceiver = new BroadcastReceiver() {
8. public void onReceive(Intent intent) {
9. LocData loc = intent.getExtra();
10. updateMap(loc);
11. if (trackingModeOn) {
12. persistToDatabase(loc);
13. }
14. }
15. }
16. //register receiver for handling location changes
17. IntentFilter filter = new IntentFilter("loc_change");
18. registerReceiver(myReceiver, filter);
19. }
20. public void onDestroy() {
21. //stop GPSService and unregister broadcast receiver
22. stopService(gpsIntent);
23. unregisterReceiver(myReceiver);
24. }
25. }

(a)

(b) Old item goes off screen and gets recycled

ViewTab 1

ViewTab 2

ViewTab 3

System recycler

//constructs new list items
getView(int pos, View recycledView, ...)

(c)

FIGURE 2. Examples of energy and performance bugs. (a) An energy leak bug. The code implements a callback that uses sensory data in an energy-inefficient way. (b) A list view (Firefox’s tab tray). (c) A GUI-lagging and memory bloat bug. The code implements an inefficient list view callback.
GPSService in the background for location sensing (lines 31–55). On receiving new location data, GPSService checks whether the data satisfy certain precision criterion (line 39). If that’s the case, it processes and broadcasts the data (lines 40–44) so that MapActivity can update its navigation map (line 10). MapActivity also stores the data on a database if location tracking is enabled (lines 11–13). Background location sensing is disabled only when MapActivity is destroyed (lines 20–24 and 51–54), which happens when users exit the application.

This design works well in many situations but can cause problems. For example, when users enter an area with weak GPS signals, the application might continue discarding noisy location data. This continual but useless location sensing can quickly drain the phone battery. Another problematic situation arises when users switch MapActivity to the background without enabling location tracking. In such cases, even if GPSService obtains precise location data, the data will be used only to render the navigation map, which is completely invisible. Again, this wastes battery energy.

To fix the bug, developers can tune down the application's location-sensing frequency or temporarily disable location sensing in problematic scenarios.

GUI-Lagging and Memory Bloat Bugs
This example, simplified from a Firefox bug, relates to frequently invoked callbacks.

Android applications are event-driven programs; a set of callbacks defines their major functionalities. The list view, for example, is a widely used GUI widget for displaying scrollable data items (such as an email list). Figure 2b shows an abstraction of a list view (Firefox's tab tray). Each list item represents a browser tab and contains two elements: a webpage icon and a webpage title label.

To render the list items, developers define a callback getView(). At run time, when users scroll the list view, getView() is continually invoked for constructing new items. The callback typically conducts two operations:

- In item inflation, the callback parses the list item's layout configuration files and constructs its GUI element tree.
- In inner view update, the callback traverses the list item's GUI element tree to retrieve specific elements for content updating.

However, file parsing and tree traversing can be time-consuming, especially when list items have hierarchical inner structures. Frequently conducting such operations can significantly reduce a list view's scrolling smoothness.

To improve performance, the Android OS recycles list items that go off the screen during user scrolling. Because list items often have an identical layout, Android applications can reuse the recycled items to render new ones, avoiding two heavy operations on each invocation of getView(). This approach is called the view holder pattern (see http://developer.android.com/training/improving-layouts/smooth-scrolling.html). Unfortunately, many real-world applications don’t adopt this good practice.

For example, Figure 2c shows how Firefox developers implemented the tab tray’s getView() callback. In this inefficient version, item inflation (line 3) and inner view update (lines 5–9) occur each time getView() is invoked. This hurts the tab tray’s scrolling performance. This implementation also fails to reuse items and consumes much memory by continuously inflating list items. Owing to the resulting memory pressure, the garbage collector will run frequently, further degrading the whole system’s performance.

**Diagnosis Challenges**

Diagnosing energy and performance bugs in smartphone applications is time-consuming and painful, and poses three main challenges.

**Triggering Energy and Performance Bugs**

Triggering bugs is a critical step before diagnosing and fixing them.

Triggering Energy and Performance Bugs

Triggering bugs is a critical step before diagnosing and fixing them. Unfortunately, triggering energy or performance bugs isn’t easy.1

First, the bugs often occur only in certain usage scenarios, and exposing them requires complex user interactions. Consider our first bug example. To expose the energy waste, we had to launch the application and switch on GPS,
• configure the application to disable location tracking, and
• run the application for a while and then switch it to the background.

Such interaction is nontrivial. In reality, users interact with applications in many ways, so it’s hard to predict which user interactions might trigger energy inefficiency or performance degradation.

In addition, triggering these bugs could require external stimulus. In the energy bug example, exposing the energy waste might require simulating the physical environment, such as poor GPS signals, which is a nontrivial endeavor.

Finally, these bugs might be triggered only under a sufficient workload. For example, to trigger Firefox’s GUI-lagging and memory bloat bug, we must open several browser tabs before scrolling the tab tray.

Judging Energy Inefficiency or Performance Degradation
Energy bugs might waste energy silently, and performance bugs might degrade performance gradually. Such bugs rarely cause immediate fail-stop consequences (such as a crash). This makes judging their existence and extent difficult.

Developers often adopt three judgment strategies:

• Make the judgments manually by running an application and observing its energy consumption and performance.

• Compare an application with similar applications to check whether its energy efficiency and performance are comparatively satisfactory.

• Rely on engineering experience. For example, many developers assume an application suffers from performance bugs if it can’t handle a user event within 200 milliseconds. However, these strategies either require nontrivial manual effort or haven’t been clearly defined. This makes energy and performance diagnosis less systematic and difficult to automate.

Diagnosis Adequacy and Effort
Diagnosing energy and performance bugs often requires considerable effort. For example, to understand the root cause of our example energy bug, developers must analyze how the application uses GPS data in many scenarios, including these:

1. The GPS data are precise, and the application is running in the background with location tracking enabled.
2. The GPS data are precise, and the application is running in the background with location tracking disabled.
3. The GPS data are continuously noisy, and the application is running in the background with location tracking enabled.
4. The GPS data are continuously noisy, and the application is running in the background with location tracking disabled.

Following such analyses, developers might realize that

• battery energy is completely wasted in scenarios 2 through 4 because all the collected GPS data are either discarded or used to render an invisible map; and
• in scenario 1, the application has slightly better data utilization because it stores data for future use, but battery energy is still wasted in rendering an invisible map.

Similarly, to diagnose Firefox’s GUI-lagging and memory bloat bug, developers must test how quickly Firefox responds and how much memory it consumes under different workloads.

Our examples are simplified. Diagnosing real-world bugs might require analyzing many more scenarios. So, it’s important to study how to improve diagnosis efficiency and effectiveness. For example, developers might want to analyze a minimal set of critical application usage scenarios to quickly understand the energy and performance bugs’ root causes. This would definitely boost their productivity.

State-of-the-Art Diagnosis
Here we review some of the most important energy and performance diagnosis techniques.

Energy bugs might waste energy silently, and performance bugs might degrade performance gradually.

Measurement and Estimation Techniques
Researchers have designed many techniques to measure or estimate smartphone applications’ energy consumption and performance. For example, vLens, eProf, and PowerTutor estimate...
Researchers have expended great effort to identify common patterns of energy and performance bugs.

Event Profilers

Developers have long used profilers to diagnose software energy and performance bugs; researchers have tailored these techniques for smartphone platforms.

For example, ARO (Application Resource Optimizer) monitors cross-layer interactions—such as user events at the application layer and network packets at the system layer—to disclose inefficient radio resource usage, which commonly causes energy waste and performance degradation.12 AppInsight helps instrument smartphone applications’ binaries to identify long latency execution paths.13 Panappticon identifies performance issues arising from inefficient platform code or problem-driven programs is nontrivial. State-of-the-art techniques often scan an application’s source code or binary to locate energy and performance problems. For example, Abhinav Pathak and his colleagues’ technique detects energy bugs caused by the forgotten release of wake locks (which keep smartphones awake).14 Lint, a popular static analyzer in the Android Studio SDK, detects a range of energy and performance bugs. Our PerfChecker can detect eight patterns of energy and performance bugs and provide actionable diagnostic information.1 (Android Studio developers integrated an enhanced version of our view holder violation checker into Lint; see https://developer.android.com/sdk for more information.)

End-User-Oriented Diagnosis

The techniques we’ve discussed so far are primarily for developers, but end-user-oriented techniques also exist. For example, eDoctor can correlate system and user events (such as configuration changes) to energy-heavy execution phases.15 It can thus help end users troubleshoot abnormal battery drains and suggest repairs (such as a configuration rollback).

Carat shares the same goal as eDoctor but adopts a collaborative, big-data-driven approach.2 It collects run-time data (for example, the active apps and device model) from a large community of smartphones to infer energy usage models. It thereby provides users with actionable advice on improving smartphone battery life. It can provide useful feedback without necessarily needing to profile much of a user’s smartphone data.

Such user-oriented techniques can also give developers useful diagnostic information. Carat can tell...
developers whether their applications would cause energy waste on certain smartphone models. eDoctor can tell them whether their applications would suffer from energy bugs under some configurations or whether new versions have energy and performance regression.

**Discussion**

These techniques have certain limitations. Developers often use estimation and measurement techniques to identify energy and performance hot spots. However, simply knowing an application’s energy cost or response time might be inadequate for effective optimization. The key diagnostic information developers need is whether the consumed energy or performed computation is necessary. For example, one energy measurement technique might identify an application component using GPS for navigation as an energy hot spot, even though it’s efficiently using the consumed energy to provide a smart service. Further research might study how to analyze such cost–benefit relations.

In addition, profilers can generate large profiles that contain considerable redundant and useless information. Effective profile aggregation, simplification, and visualization techniques are highly desirable to improve developer productivity. Besides, another open question is which information is critical to collect during profiling to effectively diagnose energy and performance bugs.

Finally, although code-pattern-based analyzers support a range of bug patterns, the root causes of many complex real-world energy and performance bugs are unclear. In addition, analyzers such as ADEL require test cases, but we don’t yet know how to effectively and efficiently construct test cases to manifest energy or performance bugs.

**A Case Study**

We applied the view holder checker in PerfChecker to 10 popular commercial Android applications to see whether PerfChecker can provide useful diagnostic information to developers. We chose our own tool because it’s easier for us to preprocess reported issues before communicating with developers. (Static analyzers inevitably generate false warnings; pruning them helps ensure that developers aren’t overwhelmed with useless information.)

Table 1 shows basic application information. These applications frequently fetch the latest cloud data to interact with users. We obtained their installation files (.apk) from the Google Play store and transformed them to Java bytecode for analysis. (PerfChecker doesn’t require an application’s source code for analysis, but if the source code is available, it can highlight code that might cause energy inefficiency or performance degradation.)

Table 1 shows basic application information. These applications frequently fetch the latest cloud data to interact with users. We obtained their installation files (.apk) from the Google Play store and transformed them to Java bytecode for analysis. (PerfChecker doesn’t require an application’s source code for analysis, but if the source code is available, it can highlight code that might cause energy inefficiency or performance degradation.)

**TABLE 1**

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>Version</th>
<th>Downloads* (in millions)</th>
<th>Reported warnings</th>
<th>True violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reddit is fun</td>
<td>News &amp; Magazines</td>
<td>3.1.13</td>
<td>1 – 5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Wechat</td>
<td>Communication</td>
<td>5.1</td>
<td>100 – 500</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>BBC News</td>
<td>News &amp; Magazines</td>
<td>2.5.2</td>
<td>5 – 10</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Sina Weibo</td>
<td>Social</td>
<td>4.2.6</td>
<td>5 – 10</td>
<td>43</td>
<td>10</td>
</tr>
<tr>
<td>Flipboard</td>
<td>News &amp; Magazines</td>
<td>2.2.7</td>
<td>100 – 500</td>
<td>18</td>
<td>10</td>
</tr>
<tr>
<td>Facebook</td>
<td>Social</td>
<td>6.0.0.28.28</td>
<td>500 – 1,000</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>LINE</td>
<td>Communication</td>
<td>4.0.1</td>
<td>100 – 500</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Skype</td>
<td>Communication</td>
<td>4.6.0.42007</td>
<td>100 – 500</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>Dropbox</td>
<td>Productivity</td>
<td>2.3.12.10</td>
<td>100 – 500</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Twitter</td>
<td>Social</td>
<td>5.2.2</td>
<td>100 – 500</td>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>

*We counted downloads from the Google Play store only.
**FOCUS: INTERNETWARE AND BEYOND**

**YEPANG LIU** is a PhD student in the Hong Kong University of Science and Technology’s Department of Computer Science and Engineering. His research interests include software engineering, software testing and analysis, and mobile computing. Liu received a BSc in computer science and technology from Nanjing University. Contact him at andrewust@cse.ust.hk.

**CHANG XU** is an associate professor in Nanjing University’s State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology. His research interests include software engineering, software testing and analysis, and pervasive computing. Xu received a PhD in computer science and engineering from the Hong Kong University of Science and Technology. Contact him at changxu@nju.edu.cn.

**SHING-CHI CHEUNG** is a professor of computer science and engineering at the Hong Kong University of Science and Technology and a director at the Hong Kong R&D Centre for Logistics and Supply Chain Management Enabling Technologies. His research interests include program analysis, testing and debugging, big-data software, cloud computing, the Internet of Things, and mining software repositories. Cheung received a PhD in computing from Imperial College London. Contact him at scc@cse.ust.hk.

Chang Xu and Shing-Chi Cheung are the contact authors for this article.

---

We really appreciate your valuable comments and suggestions on improving LINE. We would like to pass your comments to the relevant departments, where they may be used for future versions of LINE. —LINE Customer Support

It feels great to be tested by your tool, and it is awesome to connect with bright developers this way. We want to explore more interesting stuff around Android development with you. —Flipboard Customer Support

When analyzing Twitter, we found violations of the view holder pattern in the latest version of the Samsung Mobile SDK (1.5 Beta1), which Twitter uses. Our manual examination later verified these violations. We reported our findings to Samsung developers. They were quite interested and generally agreed that improving this SDK’s performance would benefit both application developers and end users (see [http://developer.samsung.com/forum/board/thread/view.do?boardName=SDK&messageId=256618](http://developer.samsung.com/forum/board/thread/view.do?boardName=SDK&messageId=256618) for details):

> With such powerful hardware, developers got lazy and started employing bad habits. … If we have better SDK [for example, improving their performance], we have happier developers and happier end users. —Samsung developer

These findings confirm that diagnosis tools such as PerfChecker can help developers find energy or performance optimization opportunities.

---

We hope that, in the future, research communities and industries will design useful techniques to help developers combat energy and performance bugs in their smartphone applications. Energy and performance diagnosis will surely become increasingly important as mobile Internetware continues to integrate itself into people’s daily lives.

**Acknowledgments**

This research was funded by the Research Grants Council (General Research Fund 611813) of Hong Kong and the National Basic Research 973 Program (grant 2015CB352202) and National Natural Science Foundation (grants 61472174, 91318301, 61321491, and 61361120097) of China.
References


IEEE Software offers pioneering ideas, expert analyses, and thoughtful insights for software professionals who need to keep up with rapid technology change. It’s the authority on translating software theory into practice.

www.computer.org/software/subscribe
A Tail-Tolerant Cloud API Wrapper

Qinghua Lu, China University of Petroleum
Xiwei Xu, Len Bass, and Liming Zhu, NICTA
Weishan Zhang, China University of Petroleum

Cloud APIs have a long-tail distribution. A proposed cloud API wrapper implements mechanisms to avoid this condition.

APPLICATIONS RUNNING in the cloud are rarely static in terms of their versions, deployment, or configurations. Internet companies report upgrading dozens of times or more a day.1 Performing these system operations, whether through Web interfaces, command-line interfaces, or specialized tools, relies heavily on cloud platform APIs. Regardless of the interface type, upgrading, redeployment, and configuration all use these APIs behind the scenes to complete the operations.

Research on system operations has focused on reducing errors and repair time rather than investigating latency issues.2,3 However, cloud consumers have limited visibility and control of the cloud infrastructure. Cloud platform API calls are the only interaction points between the cloud infrastructure and system operations. So, operations’ completion time and reliability depend on the calls’ reliability and performance.

We previously published an early version of our cloud API mechanisms.4 Here, we explore cloud API performance. We believe that cloud API calls are unresponsive for a significant percentage of invocations because they suffer from a long-tail distribution. To deal with this, we propose a cloud API wrapper with tail-tolerant mechanisms (such as hedging and alternative requests).

An Empirical Study of Cloud API Issues
We first observed cloud API issues while developing Yuruware Bolt (www.yuruware.com), a disaster recovery product that relies heavily on APIs. In exploring these issues, we extracted 2,087 API failure cases from a range of sources (a broader empirical study than our initial study, which covered only 922 cases5). These cases weren’t errors that were manifested in an error code or a crash-stop fashion or that you could significantly reduce after locating the causes. Many cases were unavoidable latency or timing failures (stuck API calls and slow response to API calls) that couldn’t be reduced in a large-scale system and often exhibited a crash–recovery behavior.

We then conducted experiments on the provisioning services’ timing behavior and observed long-tail characteristics. The API calls’ timing failures were a major cause of the long tails.

API Issues Reported in Public Sources
We extracted failure reports from a variety of public sources, including the Amazon EC2 (Elastic Compute Cloud) discussion forum (https://forums.aws.amazon.com/forum.jspa?forumID=30). We also examined technical analyses of API issues during outages from reputable sources such as Amazon outage reports (http://aws.amazon.com/message/65648), Netflix technical blogs (http://techblog.netflix.com), and the Availability Digest (www.availabilitydigest.com). To supplement the main study, we explored...
32 cloud platforms that provide APIs similar to those of Amazon EC2. However, Amazon accounted for 88.8 percent of the items in this study. We classified the 2,087 API failures into two subtypes: content failures and timing failures.

**Content failures.** A content failure means that “the content of the information delivered at the service interface deviates from implementing the system function.” Fifty-seven percent of the failures fell in this category. We classified these failures into four subtypes:

- failed calls with error messages (49 percent),
- missing content (4 percent),
- wrong content (2 percent), and
- unexpected content (2 percent).

Those failures were due mainly to API bugs. For most API bugs reported by consumers, cloud vendors fix them and release a new API version. However, this process normally takes several weeks.

**Timing failures.** A timing failure means that the delivered information's arrival time at the interface deviates from implementing the system function. Forty-three percent of the reported cases complained about slowly responding or stuck API calls. Sometimes, the API response time was slow but a response eventually arrived. For example, in one failure, the instance took 16 minutes to stop (posted on 27 Aug. 2012 at 11:57 a.m. on the EC2 discussion forum). The EC2 engineer advised to try a force stop twice if that happened again. Many users experienced API calls that took longer to complete (for example, a half hour) or got stuck. However, the API document doesn't specify the normal time to complete each API call. Sometimes, API calls didn't return at all, causing silent failures. In one case, the instance was stuck at stopping (posted on 27 June 2012 at 12:04 a.m. on the EC2 discussion forum). The EC2 engineer stopped the instance for the user on the EC2 side. However, the user complained that the volume was decommissioned, whereas the user had hoped to reuse it.

**Experiments**

We measured the five most frequent EC2 API calls (launch-instance, start-instance, stop-instance, attach-volume, and detach-volume) by calling the API 1,000 times and recording the return time. Figure 1 gives the measurement for the launch-instance API. Because this article focuses on long tails, we removed the calls that failed with error messages. We still based the calculation of the percentage on 1,000 calls. We determined that the long tail constituted 0.5 percent of the area under the distribution curve. This cutoff corresponds to 36 s (see the red dot in Figure 1). So, all the data points that are equal to or greater than 36 s represent the long tail. This translates into approximately 4.5 percent of the launch-instance calls. The other four types of calls had similar timing profiles.

**Our API Wrapper**

Figure 2 shows our API wrapper’s architecture. A cloud user requests an API through the wrapper. The wrapper interacts with the original cloud infrastructure API (the EC2 API). We maintain a repository of API timing profiles, which stores each API's wait time. When the API wrapper's API issues an original cloud API, a timer is set to specify the time-out of original-API calls. System administrators manually configure the initial time-out settings on the basis of their knowledge. At run time, the settings change dynamically according to the percentile of the historical...
return time and other factors including the process context, environment, and load.

The API wrapper processes the API calls on the basis of the state chart in Figure 2. When the wrapper triggers a tail-tolerant mechanism, the cost estimation component analyzes the selected mechanism’s cost and prints the cost in the log.

Figure 2 superimposes two state charts: the original EC2 workflow and the amended workflow that corrects for a long-tail delay. The solid arrows depict the original EC2 API’s transitions between these states during normal execution. The dashed arrows depict the transitions when a timing failure occurs when the wrapper is a particular state.

The Requested State

When the API wrapper intercepts an API call request, the wrapper enters the Requested state. This state might choose to make a normal request, a hedged request, or an alternative request if the first request fails. The hedge-request mechanism is similar to Jeffrey Dean and Luiz Barroso’s hedged request. For certain operations, we issue more requests than we need (for example, launching 12 instead of the 10 we need) and then cancel the excess requests immediately after reaching the required number. The alternative-request mechanism requests an alternative API call at the same time the original API is requested.

The Allocated State

API calls that need to allocate resources, such as attaching volumes, then transfer to the Allocated state. Other API calls transfer to the Started state (see the next section).

In the Allocated state, the API wrapper can use the continue-allocate, reallocate, cancel-allocate, or force-complete-a mechanisms. The continue-allocate mechanism schedules the request to be sent to the same instance at a future time if the API request fails or there’s no response from the cloud infrastructure within a certain time. For example, in Yuruware Bolt, we need to move data from one region to another for backup. One step creates an EBS (Elastic Block Store) volume from a snapshot. If the first ec2-create-volume fails or gets stuck, the application sends another ec2-create-volume request when a time-out occurs.

The reallocate mechanism resends the request to other instances. For example, in Yuruware Bolt, we need two data mover instances in two regions for backup. The EBS volume created from the snapshot must be attached to an instance. If the
The initial version of our API wrapper.

<table>
<thead>
<tr>
<th>API call</th>
<th>Mechanism</th>
<th>Implementation details</th>
</tr>
</thead>
<tbody>
<tr>
<td>launch-instance</td>
<td>hedge-request and continue-allocate</td>
<td>The wrapper launches one or more redundant instances by making multiple launch-instance API calls simultaneously when it receives a request to launch instances. If enough instances launch within the time specified in launch-instance’s time profile, the wrapper kills the redundant ones when they launch. If not enough instances launch successfully, the wrapper launches more.</td>
</tr>
<tr>
<td>start-instance</td>
<td>alternative-request</td>
<td>The wrapper starts an instance and simultaneously launches another instance using the same image. It cancels the instance with longer return time.</td>
</tr>
<tr>
<td>stop-instance</td>
<td>force-complete-a</td>
<td>The wrapper launches a call to the API and waits for the time specified in stop-instance’s time profile. If the call doesn’t complete, the wrapper forces the instance to stop, using the force-stop API.</td>
</tr>
<tr>
<td>attach-volume</td>
<td>alternative-request</td>
<td>The wrapper attaches the volume to an instance and simultaneously launches a new instance. The wrapper waits for the time specified in attach-volume’s time profile. If the call doesn’t complete, the wrapper reattaches the volume to the newly launched instance.</td>
</tr>
<tr>
<td>detach-volume</td>
<td>force-complete-a</td>
<td>The wrapper waits for the time specified in detach-volume’s time profile. If the call doesn’t complete, the wrapper force-detaches the volume</td>
</tr>
</tbody>
</table>

The initial version of our API wrapper transfers the API call to the Canceled state.

The Started State

When the API call is in the Started state, the API wrapper can use the force-fail-, reallocate-s, or cancel-start mechanism. The force-fail-s mechanism is for when an API call has been tried several times and continues to fail. It tries to keep time-outs short and have them fail quickly to avoid cascading time-outs. If a call is stuck in this state for a certain time, it’s regarded as a failed call, and no subsequent calls are triggered. The force-fail-s mechanism transfers the call to the Failed state.

Using the reallocate-s mechanism, the API wrapper gives up the current request and restarts the API request in another instance. For example, if the application can’t attach an EBS volume to one instance, it can try to attach it to a different instance in the same availability zone. Or, the application can ignore the current request and resend the API request to the cloud infrastructure. The reallocate-s mechanism transfers the API call back to the Allocated state.

The cancel-start mechanism is for when a started API call is stuck in a state. It transfers the call to the Cancelled state.

The Implementation

The initial version of our API wrapper wraps around Amazon EC2 APIs. To support multiple cloud infrastructures, we’ll have separate wrappers for different cloud providers. Our solution isn’t about a standardized API across clouds that requires users to change their code. However, our mechanisms could be across clouds behind the scenes.

Our initial API wrapper (https://sites.google.com/site/cloudapiwrapper) provides a tail-tolerant version of launch-instance, start-instance, stop-instance, attach-volume, and detach-volume. As we mentioned before, these calls are the most frequent. They also have significant latency issues according to both our experience and our empirical study of the EC2 discussion forum.5

Table 1 shows our wrapper’s implementation details. We built a timing profile for each API call. After the wait time reaches a configurable 90th percentile of the historical return time, we immediately resort to other mechanisms.
In our wrapper’s current version, hedge-request and alternative-request will lead to extra costs when the API call creates a wrapper for launch-instance, start-instance, or attach-volume. The extra costs are based on the rented resource’s price and usage duration. In our API wrapper, hedge-request issues one extra API call, so the extra cost of launching a t1.micro instance is US$0.02. Larger instances will cost more. Users need to be aware of these costs.

An organization must decide whether to incur these extra costs. For example, if an upgrade fixes a significant bug, an organization might decide to assume the additional costs. If the upgrade has a lower priority, the organization might decide not to.

**The API Wrapper’s Effectiveness**

To evaluate our long-tail-tolerant mechanisms, we compared our API wrapper’s return time to that of the original API.

**Setup**

Our experiments ran on the AWS (Amazon Web Services) EC2 us-east-1d availability zone. We used an AMI (Amazon Machine Image) that installed an AMP (Apache 2.2.22, MySQL 5.5.35, and PHP 5.3.10) software stack on a t1.micro-type instance.

For each API we wrapped, we measured the return time 1,000 times. The 1,000 API calls were issued in sequence within AWS EC2. As before, we removed the calls that failed with error messages, and we still based the percentage on 1,000 calls.

**Results**

Figure 3 compares the original EC2 API and our API wrapper in terms of some basic return-time statistics, including the mean, median, standard deviation, and 95th and 99th percentiles. The 99th percentile for the wrapper decreased substantially.

Figure 4. Results for the launch-instance API call. Our wrapper avoids 3.2 percent of the original EC2 API calls categorized as having a long tail (longer than 51 s).
compared to the 99th percentile for the original EC2 API. That is, introducing tolerant mechanisms in our API wrapper significantly reduced the long tails.

Figure 4 reports the results for launch-instance. (Owing to length constraints, we omit the measurements for the other four calls.) The original EC2 API used the same dataset as in Figure 1. Our wrapper’s longest return time was 51 s. The original EC2 API and our wrapper have a similar distribution of the return time when that time is less than 51 s. However, the EC2 API’s return time has a long tail to as much as 185 s. Our wrapper avoids 3.2 percent of the EC2 API calls categorized as having a long tail (longer than 51 s).

Although the probability of long-tail return time is low, a long tail’s time is very long. Sometimes, it could be 10 times the mean of the return times. Our experiments showed that our API wrapper with tail-tolerant mechanisms substantially reduced the long tail of the original EC2 API. Our wrapper could significantly reduce the impact of API issues on operations’ long tails and improve the reliability of operations in the cloud.

Because our solution provides the same API to the user through the API wrapper, users don’t have to change their scripts or code calling the API.

One limitation is that we evaluated the current version of our API wrapper, which implements only four of our nine proposed mechanisms. We have confidence in the other five mechanisms because they’re alternative mechanisms covering the same types of issues and because we’ve implemented some of

QINGHUA LU is a lecturer in the China University of Petroleum’s Department of Software Engineering. Her research interests include software architecture, the dependability of cloud computing, and service engineering. Lu received a PhD in computer science and engineering from the University of New South Wales. Contact her at dr.qinghua.lu@gmail.com.

XIWEI XU is a researcher at NICTA (National Information and Communications Technology Australia). Her research interests are software architecture, business processes, and cloud computing. Xu received a PhD in computer science and engineering from the University of New South Wales. Contact her at xiwei.xu@nicta.com.au.

LEN BASS is a senior principal researcher at NICTA (National Information and Communications Technology Australia). His research interests include software architecture, software engineering, and development operations. Bass received a PhD in computer science from Purdue University. Contact him at len.bass@nicta.com.au.

LIMING ZHU is a principal researcher at NICTA (National Information and Communications Technology Australia). He holds conjoint academic positions and teaches software architecture courses at the University of New South Wales and University of Sydney. His research interests include software architecture, service engineering, and system development methodologies. Zhu received a PhD in computer science and engineering from the University of New South Wales. Contact him at liming.zhu@nicta.com.au.

WEISHAN ZHANG is a full professor and the deputy head for research in the China University of Petroleum’s Department of Software Engineering. His research interests are big-data platforms, pervasive cloud computing, and service-oriented computing. Zhang received a PhD in mechanical manufacturing and automation from Northwestern Polytechnical University. Contact him at zhangws@upc.edu.cn.
them in Yuruware Bolt commercial workflow systems.

Acknowledgments
This project is supported by the National Natural Science Foundation of China (grant 61402533), Fundamental Research Funds for the Central Universities (grant 14CX02140A), and the Scientific Research Foundation of China University of Petroleum (grant Y1307021). NICTA (National Information and Communications Technology Australia) is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

2013/08/02/infrastructure-upgrades-with-chef.


Multitier Diversification in Web-Based Software Applications

Simon Allier, INRIA
Olivier Barais, University of Rennes 1
Benoit Baudry, INRIA
Johann Bourcier, University of Rennes 1
Erwan Daubert, INRIA
Franck Fleurey, SINTEF Information and Communication Technology
Martin Monperrus, University of Lille
Hui Song, SINTEF Information and Communication Technology
Maxime Tricoire, INRIA

Combining different software diversification strategies in multiple components can make Web applications more dependable. This article identifies key enablers for effective software diversification, especially at the application-code level.
WORDPRESS, A FRAGILE MONOCULTURE?

Wordpress is a Web content management system that supports massive customization through plug-ins. Although the vibrant Wordpress community keeps increasing the number and diversity of plug-ins for all possible tasks, we observe a paradoxical trend toward a monoculture of some popular plug-ins. This monoculture is a potential threat to the Wordpress ecosystem, particularly when the plug-ins have a defect. As an example of a massive threat, 21 percent of the 106,412 sites we sampled use the Jetpack plug-in, which has an SQL injection vulnerability.

However, you can exploit Wordpress plug-ins’ natural diversity to mitigate this threat. For example, you can replace Jetpack with WP Symposium or Disqus, which offer compatible functionalities with completely different implementations.

This example shows that, despite the emergence of monocultures in the Wordpress ecosystem, the community actively continues to develop diverse solutions.

This provides fertile ground to experiment with the automatic diversification of Wordpress sites by exploiting the natural diversity of functionally similar plug-ins to break this fragile application monoculture.

All results about the monoculture in Wordpress are available at http://diversify-project.eu/data.

---

components of Web applications. We identify key enablers for effective software diversification, especially at the application-code level. Our vision is that you can combine different software diversification strategies, from deploying multiple vendor solutions to fine-grained code transformations, to provide various forms of protection.

Application Monoculture

Reuse and modularity are key for liberating creativity and entrepreneurship in the Internet world. However, this bright world has a darker side. The problem is that reuse and modularity also help create a new form of massive-scale monoculture.

The concept of a software monoculture refers to a computing environment dominated largely by one software solution.1 For instance, the Windows OS has long been considered a monoculture on desktop machines. The term “monoculture” comes from agriculture, where research has shown that exploiting the same crop species over large areas is a bad practice. Similarly, software monoculture has a negative connotation.1 BOBE attacks are the main problem: with a large monoculture, attackers can exploit flaws and common failure modes on a massive scale.

The OS and database monocultures have been known for a long time,2 and both are key components of the general computing infrastructure. However, Internet computing has introduced application monoculture. The novelty is that this monoculture appears in application-level code (libraries, frameworks, and the application itself), where developers rely on leaky abstractions and are more concerned about time-to-market and useful features for their clients than securing their code. This monoculture might present even more risks than the OS monoculture, which grew over long periods of time with security concerns in mind.

Consider Wordpress. In the 500,000 most popular websites (according to www.alexa.com), we found 106,412 sites running it. Among those sites, 65,558 (62 percent) use the Akismet plug-in, and 21,849 (21 percent) use the Jetpack plug-in. (Akismet checks for potential spam in Wordpress comments. Jetpack provides users who deploy their own installation of Wordpress with features that are available at wordpress.com, such as advanced handling of social media or multimedia documents.) This demonstrates application monoculture at two levels: the application level (Wordpress) and plug-in level. A single attack on a 0-day flaw in Wordpress could compromise thousands of websites (see the sidebar, “Wordpress: A Fragile Monoculture?”).

Multitier Software Diversity

Web applications typically comprise server-side and client-side code working in concert. The client-side code is written mostly in JavaScript and runs in a browser. The server side is a software stack comprising an OS, a webserver, libraries, frameworks, and application-specific code.

Software diversity has long been promoted to enhance software dependability (see the sidebar, “Related Work in Software Diversification”). Since the seminal work of Frederick Cohen3 and Stephanie Forrest and her colleagues,4 researchers
have made significant attempts at automatic software diversification. Current techniques operate at the assembly-code level to mitigate memory safety vulnerabilities (such as stack overflow). Some of these techniques have been successfully implemented in mainstream OSs, making each OS installation different and thus mitigating the massive reuse of exploits. For example, all recent versions of Windows, Linux, and Mac OS have implemented address space layout randomization.

To deal with application monoculture, software diversification must address software layers beyond assembly code. Our approach is unique in that it diversifies the application-level code (for example, the application’s business logic), focusing on the technical layers in Web applications. Webserver deployment usually adopts a form of the Reactor architecture pattern, for scalability purposes. Multiple copies of the server software stack, called request handlers, are deployed behind a load balancer, which dispatches all incoming requests. Currently, all handlers are deployed as clones, but this kind of architecture provides a natural setting for diversification.

Multitier diversification is the simultaneous diversification of several application software components. This approach can rely on both natural software diversity and automatic diversity. Natural software diversity designates diverse software modules that provide equivalent functionalities and are developed by different communities or companies. For example, there’s a natural diversity of Java virtual machines (JVMs). You can exploit this by simultaneously deploying some request handlers that run the IBM JVM and others that run the Oracle JVM.

**RELATED WORK**

**IN SOFTWARE DIVERSIFICATION**

Frederick Cohen described 14 code diversification techniques to be combined to protect OSs. Stephanie Forrest and her colleagues emphasized the need to build diverse computing systems and suggested diversification based on code manipulation. Since these seminal works, several approaches have implemented automatic diversification transformations at the machine-code level and have combined them to increase software system dependability. Each kind of transformation targets a specific kind of vulnerability, and several research projects have started combining them to get more complete protection. These integrated software diversity techniques are particularly interesting regarding multitier diversification.

For example, Sandeep Bhatkar and his colleagues thwarted code injection attacks by integrating various forms of randomization. For instance, they randomized base addresses of memory regions to make the objects’ addresses unpredictable, permute the order of variables in the stack, and introduce random gaps in the memory layout. Matthias Jacob and his colleagues targeted tamper resistance through superoptimization to identify semantically equivalent instruction sequences and through other transformations that change the set of instructions and operands. The Genesis project mitigated return-to-libc attacks and code injection by implementing a virtual machine that integrated calling-sequence diversity and instruction set randomization through software dynamic translation. Chenxi Wang and her colleagues obfuscated critical software modules in survivability infrastructures by combining control-flow flattening and introduction of aliases.

For more references, see Per Larsen and his colleagues’ recent survey. Hamed Okhravi and his colleagues have summarized the different levels of moving-target defenses.

**References**

Application-level automatic diversity is provided by code transformations that generate diverse versions of some application components. Examples include method body intermixing, randomizing the database query language, and synthesizing sosies (program variants that exhibit the same functionality but are computationally diverse, with different control flow or dataflow).

The diversification of application code should provide diverse failures and vulnerabilities in webserver deployment. Thanks to the multiplicity of request handlers running on a webserver, we can simultaneously deploy multiple combinations of diverse software components. If one handler is hacked or crashes, the others should still be able to process client requests. Some components’ natural diversity exhibits diverse failure modes. For example, some vulnerabilities are present only in the Oracle JVM (vulnerabilities CVE-2014-4244 and CVE-2014-2490).

Meanwhile, automatic diversity can produce large numbers of local changes in the code, which affect sosies, by changing the computation flow, can modify vulnerabilities such as lack of input validation or business logic vulnerabilities.

**Proof of Concept**

We developed and diversified a prototype Web application that uses common, off-the-shelf components. The application is MDMS (MarkDown Content Management System), a multiuser blogging system. (All the code for this experiment is at http://diversify-project.github.io.) MDMS lets users view, create, edit, and delete blog posts. It’s implemented in JavaScript and runs on top of the RingoJS server-side framework. RingoJS is written in Java and complements the Rhino open source JavaScript engine. A Redis distributed database stores the application data.

Figure 1a illustrates this application stack. The architecture reflects the characteristics of many Web applications: a server-side scripting language, a server-side framework, and a NoSQL database.

We deployed MDMS on request handlers, arranged behind a load balancer, following a typical Reactor pattern to let the application elastically scale over time. Following current practices, our initial deployment was as follows. All handlers provided the same RingoJS environment, running on top of an identical JVM. This usual way of deploying the Reactor pattern resulted in non-diversified deployment of MDMS (see Figure 1b). This was an application monoculture: all servers behind the load balancer were clones, from the OS level up to the libraries and application software.

We injected diversity into MDMS at four levels: virtual machine, JVM, JavaScript library, and deployment infrastructure (see Figure 1c). Diversification at each level exploits either natural diversity or novel automatic-diversity techniques.

**OS Diversification**

MDMS has no dependencies on any particular OS, so we could use a mix of different distributions of Linux, BSD (Berkeley Software Distribution), and Windows. In our experiments, we randomly deployed MDMS on Windows and Linux virtual machines.

**JVM Diversification**

The three major suppliers of JVMs are IBM, Oracle, and OpenJDK (Open Java Development Kit), each having several versions. Although the OpenJDK and Oracle JVMs share the same code base, the latter has some built-in commercial and open source tools that the former doesn’t. These differences are significant enough to leave vulnerabilities in only the Oracle version. In our experiments, RingoJS alternately ran on Oracle-jdk1.7.0_45 (Sun/Oracle), IBM-Java-x86-71 (IBM), and Java-7-openjdk (GNU).

**JavaScript Library**

This layer is the RingoJS framework. This is an open source component with no competing functionally equivalent alternatives. To diversify this level, we used our technique for automatically synthesizing sosies. This synthesis transforms the original program through statement

---

Automatic diversity can produce large numbers of local changes in the code, which affect specific kinds of vulnerabilities.
deletion, addition, or replacement. Using this diversification technique, we synthesized 70 RingoJS sosies for MDMS.

**Deployment Diversification**

Through the cloud and its deployment interface, MDMS can be randomly deployed in two geographically distant datacenters (in Europe and the US). Our experiments used the Amazon cloud and a private cloud running an LXC container (Linux container). We used CloudML (Cloud Modeling Language)\(^7\) to model and automatically deploy the application between different cloud providers and OSs.

**Overall Deployment**

We have two versions of the OS, three diverse JVMs, 70 Rhino sosies, and two datacenters. Because the layers are mostly independent, we can combine the alternatives for each layer to create an exponential number of possible server deployments. For example, our setup could run 840 diversified webservers to deliver MDMS.

We deployed 17 instances of the server stack behind an Nginx webserver. In particular, we deployed 17 versions of the RingoJS library, a functional component that has never been diversified with other approaches. We designed eight functional test scenarios to validate MDMS’s global functionality. We checked that these scenarios

---

**FIGURE 1.** The architecture of the MDMS (MarkDown Content Management System) multiuser blogging system. 
(a) The standard MDMS software stack. (b) Homogeneous deployment of identical MDMS configurations. (c) Multitier diversified deployment of different MDMS configurations.
Automatic Diversification

Automatic diversification is the key to true diversity. For example, the automatic synthesis of 70 sosies was the main reason for the 840 different MDMS handlers. However, few solutions exist for automatic diversification of application-level source code because it requires transformations that are on the edge between functional correctness and quality of service, which might not preserve the original program’s semantics. Recent research on unsound program transformations opens the way for such novel, massive diversification techniques. This research includes Martin Rinard and his colleagues’ work on loop perforation, Westley Weimer and his colleagues’ work on code transformation for automatic patching, and our work on sosie synthesis.

To explain these transformations’ potential effects, we look at the RingoJS sosies, whose control flow or dataflow differ from the original RingoJS. This diversification has different causes. An attribute can be assigned a different value than in the original program, with no visible impact on the application. Some method calls can be removed, eliminating a complete part of the program’s computation, yet the program still provides the service. These cases occur either because the changes have no side effect (for example, the variable is never used) or, more interestingly, because they occur in plastic zones—parts of the computation that tolerate variations. These zones can appear in algorithms that compute some form of heuristic or in redundant code. Plastic zones are interesting from a security viewpoint because they indicate zones in which some potentially vulnerable code can be removed.

Code replacement has a different impact. For example, Figure 2 illustrates a statement replacement that adds an input validation. In general, such reduction of the input space, if it still provides the service, is good for security.

Eventually, we need to understand how to navigate the functional neighborhood of programs to provide different degrees of application-level diversity while maintaining functional compatibility.

Integrating Multitier Diversification into Development

The second challenge is integrating multitier diversification into Web software engineering practices, to master its impact and leverage its full potential for dependable Web applications.

First, we can deploy the diverse handlers in different ways. In our experiment, we picked 17 of the 840 handlers and deployed them to provide spatial diversity. Then, the system sent incoming requests to one of the handlers, round-robin. Several other strategies are possible; for example, we could:

- pick a single handler and deploy it 17 times;
- constantly deploy new versions of the handler, providing temporal diversity to form a moving-target defense;
- use the diverse handlers in a multiversion fashion, with a voting mechanism.

However, we need to understand what strategy best fits a given dependability goal.

Second, diversification affects distribution and maintenance. For example, when a third party must sign an application’s binary code, the production of millions of diverse variants becomes a challenge. One

---

**FOCUS: INTERNETWARE AND BEYOND**

/* The following snippet */

```java
/** The following snippet */
case JAVA_OBJECT_TYPE:
    return arg;
default:
    throw new IllegalArgumentException();
/** is replaced by */
case JAVA_OBJECT_TYPE:
    if (arg != null)
        return arg;
default:
    throw new IllegalArgumentException();
```

**FIGURE 2.** A transformation example in a RingoJS sosie. This statement replacement adds an input validation.
solution is to diversify the bytecode during installation (after the signed version has been shipped) and have the transformation recognized as legitimate (and not detected as malware). Other examples are dump trace analysis or incremental updates. This will require accurate traceability of variants and reversible code transformations, as well as new forms of code analysis for automatic patching.

Run-Time Environments
Our proof of concept taught us that the integration of multiple levels of diversification poses several technical challenges. We need a reconfigurable load balancer that can reason...
about the number of handlers it sends a request to, detect potentially vulnerable handlers, and stop using them or decide to replace them. We also need architectures that allow the composition of multiple levels of diversity. We must manage application state consistency between diverse handlers and support dynamic reconfiguration and deployment of software in a distributed infrastructure (using a virtual machine, containers, and software modules).

The MDMS architecture natively supports multitier diversification. The diverse request handlers store data in a Redis database (a distributed NoSQL solution) instead of a file system. We adapted the NGinx load balancer with specific distribution and recovery policies when a handler fails. We experimented with both Kevoree (http://kevoree.org) and CloudML (http://cloudml.org) to manage deployment of software modules on diverse and distributed virtual machines. Both frameworks provide utilities to seamlessly handle the heterogeneity of technologies for virtual machines (for example, Vmware and VirtualBox), system containers (such as docker, lxc, and jails), and app containers (such as servlet, android, and osgi). They provide flexible configuration models with built-in architecture model exploration and the ability to orchestrate coherent, transactional reconfiguration of the platform, infrastructure, and service levels. We plan to experiment also with Mesos to manage virtual machines or container deployment.

Our experiment highlights the challenges for software engineers if they want to systematically break the application monoculture of Web applications. In particular, we believe that unsound program transformations open the way for the true explosion of application code diversity. May multitier diversification be the end of multitier monoculture! ©

Acknowledgments

The research leading to these results was supported partly by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements 600654 (the DIVERSIFY project, FP7-ICT-2013-9) and 611337 (the HEADS project, FP7-ICT-2013-10).

References

Roundtable
The Future of Software Engineering for Internet Computing

Jian Lü, Nanjing University
David S. Rosenblum, National University of Singapore
Tevfik Bultan, University of California, Santa Barbara
Valerie Issarny, Inria Paris-Rocquencourt
Schahram Dustdar, Vienna University of Technology
Margaret-Anne Storey, University of Victoria
Dongmei Zhang, Microsoft Research, China

FOR THIS SPECIAL ISSUE, seven research leaders in software engineering for Internet computing discuss important issues that will shape this field’s future. The essays cover opportunities and challenges for the shifting software paradigm (Jian Lü); stepping outside the comfort zone to revisit issues such as software correctness (David S. Rosenblum); improving Internet software dependability and programmability (Tevfik Bultan); addressing software engineering issues for the Internet of Things (Valerie Issarny); exploring the relationships among the Internet of Things, people, and software services (Schahram Dustdar); supporting a participatory culture of software development (Margaret-Anne Storey); and rethinking logging in online services (Dongmei Zhang). Enjoy! —Antonia Bertolino, M. Brian Blake, Pankaj Mehta, Hong Mei, and Tao Xie, guest editors

Internetware: Shifting the Software Paradigm toward the Internet
Jian Lü

The Internet, not only of computers but also of things and human users, has been rapidly and profoundly changing how we construct, deploy, and use software applications. To achieve their application goals, software systems on this Internet platform need to coordinate autonomous third-party services and resources, adapt to constant changes in their environment and the requirements they must satisfy, and continuously maintain a quality of service (QoS) that satisfies users. So, Internet computing poses the significant challenge of how to help software engineers manage these new dimensions of complexity.

Conventional software paradigms, such as structured, object-oriented, and component-based methods, are inadequate here. We need a paradigm shift to comprehensively support Internet computing applications that are autonomous, cooperative, situational, evolvable, emergent, and trustworthy. To this end, Chinese software
FOCUS: INTERNETWARE AND BEYOND

JIAN LÜ is a professor of computer science in Nanjing University’s State Key Laboratory for Novel Software Technology. Contact him at lj@nju.edu.cn.

INTERNETWARE AND BEYOND

Coaching-centric architecture. Programmable connectors and run-time software architecture models facilitate flexible but disciplined coordination of autonomous entities.

Environment-driven adaptation. Run-time environment models, built as integral parts of Internetware systems, direct the probing and interpretation of the real environment and drive system adaptation when necessary.

Internetware also aims to provide comprehensive assurance of system quality. First, besides traditional quality factors such as correctness, performance, and reliability, it emphasizes quality factors related to the user experience, such as energy efficiency, privacy, and user-friendliness. Second, instead of setting a universal, fixed QoS target, it considers user-specific, dynamically tunable targets by taking into account user preferences and feedback. Finally, to ensure trustworthiness in the open Internet, it not only analyzes software artifacts’ quality but also manages the trust relationships among the subjects owning the artifacts.

Internetware researchers have made progress in architecture models, middleware frameworks, and development tools, along with system prototypes and case studies. However, much research remains to realize the full vision of Internetware. We need more research on systematic software engineering methodologies and enabling techniques that will eventually shift the software paradigm toward Internet computing.

References

Stepping outside Our Comfort Zone

David S. Rosenblum

The Internet poses many challenges to software engineering; these challenges are becoming particularly acute with the emergence of computing paradigms that exploit the Internet in new ways. The emergence—and convergence—of mobile and ubiquitous computing, sensor networks, cyber-physical systems, data analytics, and the Internet of Things is pulling software engineering further and further from the comfort zone of principles and techniques that have prevailed for many decades.

Looking through the prism of software engineering research, you get little sense of the enormous changes taking place and their accompanying challenges. Software engineering researchers largely still view software and its engineering precepts and solutions much as they always have. Yet even something as basic as the concept of correct behavior is being undermined by the changing nature of software.

Consider the use of machine learning in software engineering. Machine-learning research has produced a rich, diverse collection of approaches and algorithms for performing automated statistical classification of data processed by software at run time. Machine learning has matured to the point at which it’s being commoditized in powerful tools such as Weka (Waikato Environment for Knowledge Analysis) and contributing powerful functionality to a wide variety of real-world applications.

At the algorithmic level, it’s challenging enough to determine whether some implementation of a machine-learning algorithm is producing the outputs it should, given the inputs it receives. But even if we’re willing to accept at face value the correctness of the algorithm’s implementation, we must still deal with the consequences of the statistical nature of the algorithm itself, whose classification ability typically is less than 100 percent precise. A rose by any other name is still a rose, but a machine-learning classifier might occasionally say it’s a daisy. That might be the best (the algorithm of) the classifier can do. So, its output occasionally might be incorrect, but that doesn’t mean the classifier itself is incorrect in the traditional software engineering sense. If we embed this classifier within a larger application, and the application occasionally produces incorrect output, how can we tell whether this is due to the classifier’s imprecision or some fixable bug?

This problem, which Sebastian Elbaum and I are studying, is but one of the many ways the Internet threatens some of the bedrock notions of software engineering. We
as software engineering researchers have only just begun to understand and appreciate these challenges’ implications. It’s time for us to step outside our comfort zone and address them.

**Reference**

DAVID S. ROSENBLUM is a professor of computer science in the National University of Singapore’s School of Computing. Contact him at david@comp.nus.edu.sg.

---

**Internet Software’s Dependability and Programmability**

Tevfik Bultan

Internet computing has been evolving rapidly as we keep finding new ways to build and use network-connected computing devices (NCCDs). Programmable phones and tablets have overtaken PCs. In a few years, programmable glasses, watches, or cars might be the most common NCCDs.

In concert with NCCDs’ increasing diversity, we see a uniform approach to software application development based on the software-as-a-service paradigm and the multi-tiered architecture. Nowadays, most applications are software services hosted on clouds and accessed by thin clients that execute on NCCDs.

In this modern computing landscape, software engineering research for Internet computing faces two related challenges: improving dependability and programmability.

The necessity to address the first challenge was made painfully clear last year by the problems that HealthCare.gov encountered. Modern software applications are distributed systems comprising multiple components executing on multiple machines and interacting with each other by exchanging messages over the network. Given these systems’ complexity, you could argue that it isn’t surprising to have many bugs and security vulnerabilities. However, if we want to continue making software systems a critical part of every human endeavor, we need to develop software engineering techniques that are better at establishing dependability.

The second significant challenge is, how can we make software development easier, so that even end users can participate in basic application development or customization? This challenge is critical because the explosive increase in software applications will likely continue as novel NCCDs are introduced and people find new uses for them. I don’t think we’ll be able to produce enough computer scientists to meet the demand for application development. We’ll need to provide mechanisms that let end users create applications or customize existing ones.

The following research directions can help us address these two fundamental challenges.

Current software development practices based on general-purpose programming languages rely on the talents of individual developers. Success in state-of-the-art software development isn’t easily repeatable and scalable. We must develop higher-level abstractions for modern software application development. We need to think about what’s the best way to specify a software service’s behavior rather than focus on pieces of the problem such as client-side versus server-side programming. We should develop high-level modeling languages that can be automatically compiled to executable code. Let the compiler decide where to run the code (on the client or server side). We should be thinking about what’s the best abstraction for specifying the application’s behavior.

In addition, we should devise modular software development techniques that let us write an application as a combination of multiple policies, each specifying a certain aspect of the application behavior. For example, you could think of a modern Web application as a combination of several policies, such as:

- a navigation policy identifying the flow between different views of the application;
- a data update policy specifying how the data in the persistent storage is updated;
- a UI policy identifying how to present the views to the user;
- an access control policy identifying which data a user has access to.

We need to develop software engineering techniques that are better at establishing dependability.
an authentication policy specifying how users are identified, and
• an input validation and sanitization policy specifying how the user input is accepted.1

Existing Web application development frameworks somewhat support modularization along these lines. However, these policies are written using general-purpose programming languages, which make their analysis and verification difficult. We should develop high-level domain-specific languages that facilitate modular application development with formal guarantees for separation of concerns. (For example, an access control policy can’t be violated by the navigation policy.) Finally, improving dependability and programmability require increasing the level of automation in application development. We can achieve such automation by increasing the level of abstraction and modularity. So, contributions in the two research directions I just outlined would facilitate better automated techniques for code analysis, synthesis, verification, and repair.

References

TEVFİK BULTAN is a professor in the University of California, Santa Barbara’s Department of Computer Science. Contact him at bultan@cs.ucsb.edu.

Software Engineering for the Internet of Things

Valerie Issarny

The vision of pervasive computing, since its introduction by Mark Weiser in the early ’90s and throughout its redefinition along the years, hasn’t evolved much. Furthermore, this is clearly no longer a vision; nevertheless, in many of its aspects, it still challenges the computer science communities at large. The software engineering community is no exception, and the challenge is even more so with the advent of the Internet of Things (IoT). The IoT promises to blend the physical and virtual worlds, hence introducing a vast amount of knowledge into our now largely pervasive, distributed software systems.

With the IoT, sensing and actuation are called on to become a utility. To make the IoT a reality, research must solve these challenges:

• enable massive scaling, considering the foreseen trillions of things;
• devise system architectures that cope with the networking environment’s high heterogeneity and dynamics;
• extract knowledge from the sensed raw data;
• support an open environment, whereas sensor-based systems so far have been mostly closed domain-specific systems;
• guarantee robustness of the enacted systems despite the mostly unknown networking environment;
• enforce security and privacy; and
• allow the synergistic operation of humans and things.

Addressing these challenges will affect the development of the supporting software systems.

To meet these challenges, our group at Inria Paris-Rocquencourt has been studying extensively how to leverage but also revisit the traditional service-oriented-architecture paradigms.2 Indeed, service orientation combined with semantic technologies allows dealing with the IoT’s dynamics and heterogeneity. Still, the massive scale of the network of things calls for completely new protocols to discover, access, and coordinate things, including mobile things.3 In addition, development environments for applications to be deployed over the IoT remain pretty much an open issue in light of the requirements for openness and robustness. Similarly, we need software tools that enable reasoning about applications’ security and privacy. Finally, software tools to process and analyze the big data made available by the IoT have yet to be devised.

A key issue underlying these challenges is the traditional centralized architecture versus a distributed architecture. Distribution is crucial to meeting these challenges. Moreover, solutions must be probabilistic, given the uncertainty of the target networking environments.
The Internet of Things, People, and Software Services

Schafram Dustdhar

The Internet has undergone an essential transformation and has been a stunning success. It changed from being a network of networks enabling access to remote machines to a network of content, applications, people, and (software) services, thereby weaving itself into the fabric of today’s global and interconnected society. We can safely claim that today’s use of the Internet constantly transforms how people, businesses, and society as a whole operate.

The interactions we witness are such that some claim they defy the laws of behavioral physics because they’re built by autonomously interacting people who are often unpaid and intrinsically motivated. Assumptions about interaction models and patterns (between humans, systems, processes, and organizations) are seriously challenged. Novel foundational technologies and methods need proper attention from science, particularly computer science and information systems.

Future Internet (FI) research’s goal therefore must be to provide the infrastructure (networks and services) and means to deal with the changing of a software executable) but a human being. This person will (through machine-processable interfaces) provide services and (automated) interactions, called service ensembles. The building blocks of service ensembles are active, which renders

With the Internet of Things, sensing and actuation are called on to become a utility.

References

VALERIE ISSARNY is a senior research scientist at Inria Paris-Rocquencourt. Contact her at valerie.issarny@inria.fr.

SCHAHRAM DUSTDAR is a professor in the Vienna University of Technology’s Faculty of Informatics. Contact him at dustdar@dsg.tuwien.ac.at.
Supporting a Participatory Culture of Software Development

Margaret-Anne Storey

Over the past few decades, software development has transitioned from a predominantly solo activity of developing standalone programs to a highly distributed, collaborative approach that depends on or contributes to large, complex software ecosystems. The distributed and collaborative nature of software development continues to grow as new Web-based tools are proposed and adopted. These tools offer social networking features (for example, watching and following other developers or projects) and lightweight, transparent channels for knowledge sharing. Such features facilitate the emergence of a participatory development culture, in which developers are keen to learn from and cocreate what others think of their contributions.

This participatory culture emerged not just because of social tools but also owing to the advancement of remote systems and the wider availability of the Internet in the early ’80s. Development communities naturally formed around free and open source projects, with communication tools such as email, version control, and Usenet playing an important role in community formation. This participatory culture isn’t isolated to open source projects; it’s also clearly evident in industrial distributed and global software development projects. Nowadays, we see social coding, microblogging, social news sites, cloud-based development tools, and question-and-answer websites playing an essential role across many development contexts. The participatory nature of software development is embedded in and facilitated by an ecosystem of tools, developers, and content. The continued adoption of social systems and online development tools leads to three major trends.

First, we see the emergence of social developers who are passionate about contributing to community resources and who care deeply about what others think of their contributions. They nurture and use their social networks to stay up to date with technological changes, to broaden their skills and manage their own identity. Furthermore, their development tasks shift from writing new code to reusing or mashing up existing solutions. So, their success and effectiveness rely not only on technical skills but also on how they can use their social networks to find, curate, and share important information.

Second, the Internet as a hosting platform and environment for software development increases the emphasis on data over code in software engineering processes. Continuous release cycles, large-scale testing in the wild, user feedback through social media, and operational data from distributed development are just some of the data resources that can be analyzed and visualized to improve software quality, the user experience, and developer productivity. The ability to continuously analyze and visualize real-time information plays an important feedback role in the participatory-development culture.

Third, the use of the Internet to host community projects and tools across diverse domains broadens participation (for example, to scientists and other end-user programmers). Such participation will likely expand further owing to the ubiquitous nature of computation that’s visible across the Internet of Things. The Internet together with social tools supports community-authored and community-curated resources that help attract and retain those participants. However, the transparency these environments afford might also lead to participation barriers that shouldn’t be ignored.

Finally, software developers are sometimes called the “prototype knowledge workers of tomorrow.” Developers are the creators or early adopters of new technologies that knowledge workers (for example, in healthcare, the sciences, or journalism) might rely on in the future. So, the impact of understanding the challenges of and opportunities from adopting and using social tools could reach across many knowledge domains.

The use of the Internet to host community projects and tools across diverse domains broadens participation.

References
Rethinking Logging in Online Services

Dongmei Zhang

Logging is important for recording program execution process and debugging problems that occur during execution. It’s widely used in in-house software development and in telemetry that collects program run-time information in the large. In the Internet computing era, logging has become even more critical in the quality management of online services because it’s almost the only feasible diagnosis mechanism for large-scale distributed service systems. Although logging’s value is indisputable, it faces new challenges in the context of online services.

The first challenge is cost. Owing to the massive user base and enormous volume of transactions served by online services, the volume of logs will increase significantly, resulting in huge storage and processing costs. Additionally, a huge number of logs often poses serious challenges to problem diagnosis. For example, many logs are often irrelevant to the problem under investigation, thus making diagnosis like finding a needle in a haystack.

The second challenge is how to control logging quality. Logging quality deals mainly with two issues. One is to detect and avoid logging incorrect information—data bugs. Data bugs are more likely to occur in a fast-paced dynamic development environment with many engineers. Flexible logging schemas, frequent code changes, code maintenance, and so on can cause data bugs.

The other logging-quality issue relates to effectiveness and efficiency. Insufficient logging might impact the logs’ effectiveness because it might miss run-time information needed for postmortem analysis. Excessive logging might impact the logs’ efficiency because the extra logs might incur a prohibitive cost at run time and in offline storage and processing.

Finally, the huge quantity of logs demands scalable, effective analysis techniques and tools to help engineers gain insights into their service systems’ quality. On one hand, this lets engineers diagnose and resolve service problems as quickly as possible to reduce the mean time to recovery. On the other hand, engineers can use the insights to proactively detect and fix hidden problems in the systems and to enhance the monitoring mechanism accordingly.

It’s great to see the research conducted and published on logging over the past few years. I hope to see more researchers and practitioners rethink logging in the context of online services, and tackle the aforementioned challenges with breakthroughs in both research and practice.

References

FEATURE: ARCHITECTURAL LANGUAGES

The Road Ahead for Architectural Languages

Patricia Lago, VU University Amsterdam
Ivano Malavolta, Gran Sasso Science Institute
Henry Muccini, Università dell’Aquila
Patrizio Pelliccione, Chalmers University of Technology and University of Gothenburg
Antony Tang, Swinburne University of Technology

An exploration into the usability requirements of architectural languages in terms of language definition, language features, and tool support can lead to the next generation of architectural languages.

"IF YOU THINK GOOD architecture is expensive, try bad architecture."1 With this reflection, Brian Foote and Joseph Yode convey the message that developing an architecture is complex and expensive. One of the most complex and expensive tasks in software architecture design is the precise specification and communication of that architecture. A badly specified architecture design causes design and implementation flaws in a system and can create misunderstanding. In this article, we build on empirical studies to examine architectural languages (ALs)2 and model-driven engineering (MDE)3,4 as a means to improve architecture design.

ALs provide a way to describe a software system's architecture. According to the ISO/IEC/IEEE 42010-2011 Systems and Software Engineering—Architecture Description standard,5 an AL is “any form of expression for use in architecture descriptions.” So, an AL can be a formal language such as Acme, Darwin, or Architecture Analysis and Design Language; a UML-based notation; or any other way to describe a software architecture. A plethora of ALs has been proposed since the late 1980s, starting with box-and-line notations to describe systems as sets of components and connectors. In the late '90s, researchers remarked on their limited usefulness to provide automated analysis and implementation,6 which led to a thread of research on formal ALs (for example, Wright, Cham, and Darwin) and resulted in tens of different languages. Some ALs provide features for specific application domains (such as automotive or avionics), whereas some UML-based languages and UML profiles are general purpose.7

However, our previous work involving 48 practitioners from 40 IT companies revealed a number of AL needs, many of which have a practical orientation (see the sidebar “What Industry Needs from Architectural Languages”).2 Building on those results and an in-depth analysis, this article defines, classifies, and clusters the requirements into a well-organized framework for designing and developing new ALs. In addition, to better clarify AL requirements, we explore MDE as a technology to help realize next-generation ALs.8 MDE refers to the systematic use of models as first-class entities for describing specific aspects of a software system (such as data persistence, security policies, or software architecture) and the use of suitable engines for defining, analyzing, and manipulating those models.
WHAT INDUSTRY NEEDS FROM ARCHITECTURAL LANGUAGES

To understand what organizations using architecture descriptions really need, we conducted an empirical study with 48 practitioners from 40 different IT companies in 15 countries. The main purposes of the study were to understand which and how architectural languages (ALs) are used in the software industry, why some ALs aren’t used in practice, and what AL features are lacking according to practitioners’ needs.

We interviewed industrial experts who have used different types of ALs in production (including formal, semiformal, and informal ones). The study participants’ software development experience ranged from two to 40 years, and averaged 19 years. Organizations participating in the study included both small to medium-size companies (52 percent) and large companies (48 percent). These organizations develop systems pertaining to both critical domains (for example, automotive, avionics, industrial automation, business information, and finance) and noncritical ones (such as media and entertainment, education, and project management).

We found that 86 percent of the respondents’ organizations use UML or a UML profile, whereas approximately 9 percent use ad hoc or in-house languages, the remaining 5 percent of respondents declared to not use any modeling language for representing the software architecture of the system. Apart from ad hoc languages, the most-used ALs are Architecture Analysis and Design Language (around 16 percent), ArchiMate (around 11 percent), Rapide (around 7 percent), and EAST-ADL (around 4 percent). Moreover, only around 12 percent of respondents use architecture description languages (ADLs) exclusively, around 35 percent mix an ADL and UML, and around 41 percent use UML exclusively.

At the core of the study is a reflection about the needs and perceived limitations about ALs in industry. On one side, the most important identified needs are (in order) design (around 66 percent of respondents), communication support (around 36 percent), and analysis support (around 30 percent). Surprisingly, the least important needs are for code generation and deployment support (around 12 percent) and development process and methods support (6 respondents, 18 percent). On the other side, the most recurrent identified limitations are related to the insufficient expressiveness for non-functional properties (12 respondents, around 37 percent), insufficient communication support for nonarchitects (8 respondents, around 25 percent), and the lack of formality resulting in languages with no precise semantics, usually with no clear workflow on how to use them (around 18 percent).

Furthermore, some participants also declared that they have not adopted any AL for the following main reasons:

- formal ALs’ need for specialized competencies with insufficient perceived return on investment,
- overspecification as well as the inability to model design decisions explicitly in the AL, and
- lack of integration in the software life cycle, lack of mature tools, and usability issues.

Interestingly, the study showed that software architects have two dual and complementary roles that must be appropriately reflected in ALs:

- Analyst and quality auditor. The architect’s skills are mostly oriented to the disciplined development of the architectural design model. Tasks involve the design, analysis, and capturing of design decisions that have an impact on quality attributes such as cost, safety, evolution, cost, and security.
- Negotiator and communicator. The architect’s skills are mostly oriented to the communication of architectural decisions and knowledge to other stakeholders. The architect communicates and collaborates with project teams, customers, and developers by sharing and coediting system designs and the underlying knowledge using various media ranging from verbal discussions, email, and wikis to the documenting of architecture specification documents.

Readers can refer to our study1 for more details about our results and a thorough discussion about the current use of ALs in industry.

References

throughout the system development life cycle. Such modeling activities include code generation, performance analysis, and so on. MDE has proven to be very effective, and many well-established MDE techniques can satisfy AL requirements.

### A Requirements Framework for Next-Generation ALs

To better understand how next-generation ALs will support various architecting activities, let’s review a framework of AL requirements. A large number of requirements focusing on specific aspects of ALs, architecting activities, and methodologies emerged from our analysis. To organize those requirements, we classified them into three clusters on the basis of the three standard elements a modeling language in software engineering has to consider:

- language definition elements that make up the notation for modeling relevant concepts;
- language mechanisms that, built upon such concepts, offer mechanisms to change, refine, and organize the concepts in a certain context or perspective; and
- tool support that offers tools and applications for carrying out modeling activities for individual and collaborative modeling.

Figure 1 shows the requirements for next-generation ALs grouped into these three clusters.

#### Cluster A: Architectural Language Definition

The language definition cluster contains requirements about defining a language composed of abstract syntax, concrete syntaxes, and semantics. In this context, practitioners suggested many needs and concerns that we elaborate into three main requirements:

1. Support to specify nonfunctional properties. There’s a growing need for non-functional analysis such as dataflow analysis, runtime dependency analysis, and analysis on performance, scalability, security, requirements, and change impact.
2. Formal semantics. Formal languages (such as process algebras, state charts, CSP (Communicating Sequential Processes), and π-calculus) give a precise and unambiguous semantics to the language. While practitioners consider formal semantics an obstacle to usability and dissemination of ALs, they also recognize it as an important enabler for analysis and other automatic tasks.
3. Support for graphical and textual specification. Practitioners report the need to use a combination of textual and graphical representations in the same project. For instance, graphical representations can be useful for knowledge sharing and discussion, whereas expert users might use textual representations for rapidly building a model.

---

**Figure 1.** A framework of architectural-language requirements organized into three clusters: language definition, language features, and tool support.
These requirements underpin the needs for an AL language to express an architecture model.

Cluster B: Architectural Language Feature

The language mechanisms cluster contains three requirements that offer features to organize, change, and refine architecture descriptions:

1. Multiview management. There’s an emerging need for multiview modeling, where each view delivers a different perspective on the architecture or addresses a different concern or stakeholder. In this context, ALs should be able to manage multiple views of the same architecture as well as maintain consistency across the various views.

2. Extensibility and customization. Practitioners need improved support for extending ALs to better express domain- and project-specific concepts, for specifying constraints, and for enabling additional analysis capabilities.

3. Programming framework. Practitioners need instruments—that is, suitably defined APIs—to programmatically access and operate on architecture descriptions. More specifically, a programming framework must expose facilities to manage, create, and modify different views of architecture descriptions in a coordinated way. These facilities will play a key role in the seamless integration of the software architecture description with all the other artifacts used and produced across the whole development process.

These requirements are the language features that software architects can use in different situations.

Cluster C: Architectural Tool Support

Building on language elements and mechanisms, an AL should provide tools to carry out individual and collaborative modeling activities. Our discussions with practitioners resulted in the definition of six main requirements:

1. Automated analysis. Practitioners need automated support for analyzing their systems, especially against non-functional properties. Automation must be able to mask the complexity of the analysis engine, thus reducing the demand for specific skills and competencies for performing these tasks.

2. Support for software architecture-centric (SA-centric) design. SAs should be used as high-level design blueprints during system development and later on for maintenance and reuse. Therefore, ALs should be integrated into development processes, specifically with system requirements, implementation, maintenance, and so on.

3. Large-view management. Architecture descriptions of complex systems can encompass several large, different, and interrelated views. In light of this, architectural information relevant for a specific stakeholder might be scattered across different views. Practitioners need new tools that create accessible architecture descriptions easily and pragmatically, despite this complexity and fragmentation.

4. Support for collaboration. Globalization of software development requires collaborative services across geographic areas. Collaborative services ought to support both synchronous and asynchronous collaboration.

Synchronous collaboration tools enable real-time communication and collaboration in a “same time, different place” mode, but they require that all participants must be available at the same time. Asynchronous collaboration tools enable participants to connect at their own convenience and schedule, at the cost of possible delays of interaction.

5. Support for versioning. Versioning tools allow system stakeholders to maintain a repository of artifacts with special emphasis on keeping track of their various versions throughout the project duration. Because software architects produce and consume artifacts of a heterogeneous nature, scope, and interlocutor, AL tools should provide facilities to seamlessly version architectural artifacts in an easy, transparent, and homogeneous way.

6. Support for knowledge management. Typically, organizations rely on their employees to be proactive in capturing and promoting knowledge sharing across development sites. As knowledge
is typically dispersed and ill-organized, there are many challenges in sharing and retrieving. To overcome this issue, next-generation ALs ought to leverage knowledge-sharing tools such as wikis and semantic wikis to record and discuss architectural design decisions and their rationale. This requirement is strictly connected to the well-known problem of architectural knowledge vaporization, which leads to high maintenance costs. 10

In order for AL to be workable, tools must be available to support these requirements. In order to build tools that are flexible enough to work in varying situations, we investigate the application of MDE.

Architects’ Dual and Complementary Roles

Both the architecting activities presented in the “What Industry Needs” sidebar and the requirements for ALs we just described emphasize the need to further improve ALs in supporting the dual and complementary roles of software architects: analyst and quality auditor as well as negotiator and communicator. Whatever role architects play, their ALs should satisfy a combination of the requirements from the three clusters, regardless of the kind of system being developed, the type of involved organizations and people, and the various constraints and risks of the project being carried out.

A Technological Solution for Building Next-Generation ALs

MDE is a possible technological solution for successfully supporting the requirements of next-generation ALs. In MDE, architects use domain-specific modeling languages (DSMLs) to describe the system of interest. The concepts of a DSML—its first-class entities, relationships, and constraints—are defined by its metamodel. According to this, every model must conform to a specific metamodel, similar to how a program conforms to the grammar of its programming language. In MDE, it’s common to have a set of transformation engines and generators that produce various types of artifacts. Practitioners can take advantage of transformation engines to obtain source code, alternative model descriptions, deployment configurations, inputs for analysis tools, and so on.

MDE Techniques in the Software Architecture Domain

An AL can be considered a DSML tailored to the software architecture domain. From this perspective, architecture models describe the software architecture of a system according to the structure and constraints dictated by the AL metamodel, and model transformation engines and generators (as well as other MDE techniques) can be used to accommodate the AL requirements discussed earlier. More specifically, in the following we present how MDE techniques might be successfully used for defining and managing domain-specific languages in the software architecture domain:

1. MDE can be used to define precise and unambiguous ALs that contain only the model elements that the domain requires, as well as UML profiles that extend the UML infrastructure. Empirical studies show that UML isn’t universally accepted, 3 while DSLs are far more prevalent than expected. Metamodeling and profiling provide the techniques required to support the definition or extension of modeling languages with a focus on non-functional properties such as performance, scalability, and security.

2. MDE tools give behavioral semantics to an AL by means of constraint languages—for example, OCL (www.omg.org/spec/OCL)—by mapping the language’s structure onto a semantic domain (for example, via model transformations).

3. Finally, MDE provides a set of engines for graphical, tree-based, and textual editors with various levels of automation, such as GMP (www.eclipse.org/modeling/gmp) and GME (www.isis.vanderbilt.edu/projects/gme).

In the following, we highlight the main MDE mechanisms and techniques for managing architecture descriptions with a focus on multiview modeling, language extensibility, and programmatic access to architecture model elements:

1. MDE promotes the use of multiple views linked by means of suitable relationships, which are fundamental to understanding the impact of design decisions. Each viewpoint and view can be described by a set of domain-specific languages (DSLs), and MDE can be used for developing DSLs and for tailoring them to the various needs of the architect. MDE provides model weaving, which establishes typed relationships between models. This technique stores the relationships in dedicated models called weaving models. MDE also provides model transforma-
Model-driven engineering provides different techniques to manage language and tool extensibility.

1. Practitioners can use model transformations such as QVT (www.omg.org/spec/QVT) and ATL (www.eclipse.org/atl) to automatically obtain analysis models from architectural models and to propagate analysis results back to architectural models. MDE researchers have proposed many model transformation languages, each with specific features such as directionality, incrementality, tracing support, and so on.\(^1\)

2. Practitioners can also use model transformations to automatically obtain various types of artifacts spanning the development life cycle. They can also be able to use model weaving for similar purposes. Practitioners can use them to carry out traceability analysis (between SA elements and requirements, design decisions, generated skeleton code, financial prospects, and so on) and change impact analysis while maintaining the system.

3. When dealing with a large ecosystem of models (that is, a large set of models representing the same system), MDE provides a technique called megamodeling (http://wiki.eclipse.org/AM3), which lets practitioners keep an organized register of all the involved models and their relations. MDE also provides a complementary approach to megamodeling called virtual modeling (https://code.google.com/a/eclipselabs.org/p/virtual-emf). It promotes the management of a large amount of information with a single metamodel representing all the domain concepts and, when needed, projects specific information into smaller and more manageable models on demand. When dealing with a large number of models, MDE techniques and tools allow practitioners to orchestrate a set of transformations among them and to consider those transformations as a unique chain (http://atenea.lcc.uma.es/index.php/Main Page/Resources/Wires\(^*\)).

4. MDE techniques, such as those provided by EMF Compare (www.eclipse.org/emf/compare) and CoDesign (http://softarch.usc.edu/?ronia/codesign), can automatically calculate model differences. This is an essential process to control model changes and evolutions made by geographically distributed users. MDE also provides techniques with various levels of automation to propagate model changes to geographically distributed users and to notify users about conflicts upon concurrent modifications.

5. MDE techniques provide ways to support the management of different versions of software architecture models, to effectively match and merge different versions of one model, and to identify and solve possible conflicts.

6. Knowledge-sharing tools such as wikis and semantic wikis can support architects in their decision-making process and rationale. MDE can enable
Limitations of MDE in Industrial Adoption

MDE techniques and tools address many concerns an AL designer might have, but empirical studies show that some barriers exist for its adoption. Using a combination of online questionnaires (449 responses) and interviews (22 MDE practitioners), one study showed that the barriers hampering the industrial adoption of MDE are not only technical and tool related but also social and organizational. As examples of organizational change management, the study pointed out that the successful adoption of MDE techniques needs a progressive and iterative approach, integration with existing organizational commitments, and a clear business focus.

Another study reported a taxonomy of factors that play a role in MDE adoption. This empirical study was based on 19 interviews.
with MDE practitioners working in 18 different companies. The researchers used their analysis of the data to define the taxonomy that they then validated through another 20 interviews carried out in two companies. The study shows that “MDE can be very effective but it takes effort to make it work.” From the technological point of view, the main limitations are the immaturity of tool support as well as its complexity and lack of usability. Practitioners highlighted that MDE often lacks consideration for how people think and work. Moreover, MDE requires investment in training, process change, and cultural shift.

Moreover, the success of MDE technologies depends on the domain they are applied to. Lessons from the first of the two studies just mentioned show that MDE techniques are useful in creating well-defined software architectures. In fact, the interviewees unanimously argued that MDE makes it easier to define explicit architectures, especially when MDE is a ground-up effort.

Our research provides practitioners, researchers, and tool vendors a practitioner-proven guide to focus on the requirements for designing and developing new ALs. In this context, this article offers a starting set of sources of MDE technologies, together with a thorough mapping of MDE techniques with respect to next-generation AL requirements.

The mapping between MDE techniques and AL requirements helps in understanding how an existing technological solution (MDE, in this case) can be leveraged to successfully support the requirements of next-generation ALs. More importantly, the mapping helps in elaborating and reusing the knowledge base about ALs accumulated over the years. Indeed, MDE offers more than a way to realize ALs; it further suggests how ALs can be integrated in the broader development process where architecture is the main driver of the development of a software system. Overall, this article is suggesting ways for practitioners to finally bring ALs to industry.

References

How Do I Know Whether to Trust a Research Result?

Martin Shepperd

MAGAZINES SUCH AS THIS, together with many journals, are filled with articles presenting findings, new methods, tools, and all kinds of software engineering advice. But how can we know whether to trust a research result? Or, to put it another way, should the research influence decision making? For practitioners these are important questions. And it will come as no surprise in a column such as this that the role of evidence is emphasized. It helps us decide whether we can indeed trust the research. So, it’s gratifying to see that empirical evaluation is increasingly becoming a tool for software engineering researchers.

That should be the end of the matter. Look at the evidence. Unfortunately, it doesn’t seem quite that simple. In many situations, the evidence might be contradictory or at least equivocal. This can happen even when you examine the body of evidence using a systematic literature review to identify all relevant studies and then apply meta-analysis (statistical techniques to combine multiple results into a single answer).

This lack of consistency suggests that it’s not just a matter of seeking replication of results—although this is obviously important—because the results might not accord. (See the sidebar “Inconsistent Results in Software Engineering.”) However, the problem doesn’t stop there. In other fields such as medicine, there’s “strong evidence of an association between significant results and publication; studies that report positive or significant results are more likely to be published and outcomes that are statistically significant have higher odds of being fully reported.”¹ Is this also the case in computer science?

Why False Findings Occur

In the classic but controversial paper “Why Most Published Research Findings Are False,” John Ioannidis stated there’s “increasing concern that in modern research, false findings may be the majority or even the vast majority of published research claims.”² Rejecting the explanation that most scientists are charlatans, why might this be so?

One issue is the expedient but open-to-debate tendency of claiming conclusive findings solely on the basis of a single study assessed by the formal statistical significance, typically for \( p < 0.05 \), that the null hypothesis of no effect is true. This is fine but neglects consideration of the prior probabilities.

For example, suppose I conduct an experiment and announce to the world I’ve developed a working antigravity machine \( (p = 0.049, \alpha = 0.05) \). I shouldn’t be surprised if the wider community doesn’t accept this as compelling evi-
dence. This is because my $p = 0.049$ is dominated by the a priori probability that such a result is extremely unlikely.

Of course this example is foolishness, but some fields are vulnerable. Ioannidis suggests that this will most likely occur when

- there’s little theory, so the primary research methods are experimental;
- such methods, protocols, and analysis techniques are still evolving;
- effect sizes aren’t expected to be large; and
- the prior probability of the research finding being false is high.

He highlights machine learning as being particularly vulnerable.

Another difficulty derives from the selective reporting of results. Researchers might prefer some results over others—for example, they might perceive positive results as more useful or acceptable. Other results might simply be more in accord with the researchers’ prior beliefs. The likely bias arising from selective outcome reporting is to “overestimate the effect of the experimental treatment.”3 (For a look at researcher bias and other types of bias, see the related sidebar.)

**Our Meta-analysis**

To determine whether these problems occur in computer science, Tracy Hall, David Bowes, and I conducted a meta-analysis. We wanted to understand why different studies that looked at the same or overlapping questions might come up with different answers.4 We focused on software defect prediction because this area has seen considerable research activity and we could capitalize on a systematic review by Hall and her colleagues.5

Our meta-analysis examined all the published studies we could find that provided sufficient information for our purposes. This came to 42 primary studies containing 600 experimental results in which each experiment tried to compare the predictive performance of a particular classifier (for example, logistic regression or support vector machines) for a given dataset. Typically, an experiment compares multiple classifiers across multiple datasets using what’s formally called a repeated-measures design.

Through some reverse engineering, we extracted a common response variable of prediction performance for all the studies, which was the Matthews correlation coefficient (MCC). This ranges from +1 for a perfect classifier through 0 for a random classifier to –1 for a perfectly perverse classifier.

Figure 1 shows the distribution of

---

**INCONSISTENT RESULTS IN SOFTWARE ENGINEERING**

The following are three examples of systematic literature reviews that failed to find “conclusion stability,”1 followed by a large experiment comprising many individual tests.

Magne Jørgensen reviewed 15 studies comparing model-based to expert-based estimation.2 Five of the studies supported expert-based methods, five found no difference, and five supported model-based estimation.

Carolyn Mair and I compared regression to analogy methods for effort estimation and similarly found conflicting evidence.3 Of 20 primary studies, seven supported regression, four found no difference, and nine favored analogy.

Barbara Kitchenham and her colleagues identified seven primary studies.4 Three found that cross-company models weren’t significantly worse than within-company models; four found that cross-company models were significantly worse.

Thomas Zimmermann and his colleagues’ research learned defect predictors from 622 pairs of projects <P1, P2>.5 In only 4 percent of pairs did the predictors learned in P1 predict effectively for P2.

**References**

predictive accuracy grouped by the classifier type. Considerable overlap existed between the methods, with only the naive benchmarks clearly performing worse than most of the other classifier types. Effectively, the variation within a type of classifier was greater than the variation between the classifiers. This is awkward because it suggests that the thing researchers want to know—the best way to predict software defects—is being swamped by variation from other sources.

To better understand what was happening, we modeled the results using the classifier type and three other moderators. The moderators are variables that could affect the relationship between the variable of interest (choice of classifier) and response variable of predictive performance. We used these moderators:

- **Dataset.** We introduced this because we might reasonably expect some dataset characteristics (for example, size, presence of categorical features, and noise) to favor some classifiers over others.
- **Input metrics.** The classifiers can use quite different sets of inputs—some based on process measures, others based on the changes between releases, and others using static code analysis. Perhaps this was affecting the results.
- **Research group.** We added this because we wondered whether different groups might have access to different types of expertise. We determined groups using coauthorship links and agglomerative clustering, leading to 23 clusters containing 1 to 10 authors.

In addition, our model allowed for higher-order interactions between any of the factors.

The results were startling (see Table 1). By itself, the choice of classifier was scarcely significant and contributed a little over 1 percent. In contrast, the research group dominated and contributed more than 30 percent. Also, the interaction of first-order factors created two significant second-order factors. The table shows that the main factor influencing the research results wasn’t what the researchers were investigating (how to predict defects) but who did the research.
So, it would seem that at least some areas of computer science aren’t immune to researcher bias. A separate meta-analysis of experimental results by Magne Jørgensen and his colleagues also uncovered evidence of researcher bias. Statistically significant results occurred approximately twice as frequently as you might expect from modeling the base rates.

Although an element of speculation exists, contributory reasons include varying levels of expertise, comparing highly optimized versions of some classifiers with “vanilla” versions of others, and selective reporting. This might seem highly pejorative about us researchers. However, this isn’t an attack on anyone’s integrity; we merely wish to spur progress in an important area of scientific research. After all, the ultimate aim of scientific methods is to reduce bias. Consequently, we recommend that researchers conduct blind analyses, improve reporting protocols, and conduct more intergroup studies to mitigate expertise problems.

Rather than mistrust all scientific research, researchers should seriously consider the following questions. First, how likely is a finding true, on the basis of a priori scientific knowledge? Second, to what extent did the research use blind testing? (For example, blind analysis might protect against selection bias.) Third, did the research use appropriate benchmarks—in particular, naive and random methods? Finally, are there independent replications?

References


MARTIN SHEPPERD is the head of the Department of Computer Science at Brunel University London. Contact him at martin.shepperd@brunel.ac.uk.

TABLE 1

The proportion of total variance in predictive accuracy (indicated by the Matthews correlation coefficient).*

<table>
<thead>
<tr>
<th>Factor</th>
<th>Influence (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research group</td>
<td>31.0</td>
</tr>
<tr>
<td>Dataset</td>
<td>11.2</td>
</tr>
<tr>
<td>Research group: classifier</td>
<td>6.6</td>
</tr>
<tr>
<td>Input metrics</td>
<td>5.2</td>
</tr>
<tr>
<td>Classifier family</td>
<td>1.3</td>
</tr>
<tr>
<td>Research group: dataset</td>
<td>1.0</td>
</tr>
<tr>
<td>Error</td>
<td>43.7</td>
</tr>
</tbody>
</table>

* A colon indicates a higher-order interaction between two factors.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.
The Five Properties of Successful Architectural Oversight

Nick Rozanski

**ARCHITECTURAL LEADERSHIP** is probably the most important part of a software architect’s role. A key aspect of architectural leadership is architectural oversight, which helps ensure that development teams design and build their systems in the right way for their stakeholders—the people who have an interest in the system’s success. Unfortunately, “the right way” means different things to different people.

Obviously, a system’s architectural components must meet its functional requirements. For example, a sales-ordering system will probably need order entry screens, an order manager, and an order database. These components will need to be wired correctly so that the system can perform correctly. New orders will need to flow from the order entry screens to the order database via the order manager.

However, the system will also need to exhibit the right quality properties, such as scalability, resilience, and security (often called nonfunctional requirements). Orders should be accepted only from authenticated customers, and the system should be able to scale to adapt to the volume at busy times. It’s necessary—and often more difficult—to put as much thought into these architecture aspects as you would into functional and information architecture.

Architectural oversight is about keeping a watchful eye on projects as they move from concept and architecture to building, testing, and deployment, ensuring adherence to the original vision and architecture. (And if not, ensuring there are good reasons for this and that architectural changes are properly evaluated, communicated, and agreed upon.) Overseeing in a way that ensures that the system has the right architecture can easily become overly bureaucratic or poorly leveled (for example, focusing on some characteristics in detail while ignoring others). Worst of all, this process can fail to ensure that the delivered systems are sound and fit for the purpose.

Effective architectural oversight has five important properties: it must be timely, objective, systematic, constructive, and, most important, pragmatic (see Figure 1). Here, I examine each property to understand what it means, why it’s important, how you can achieve it, and what might happen if you don’t.

For this article’s purposes, I’ll assume you’re an architect with several systems in your portfolio, including new builds or systems undergoing substantial changes. You might be the architect of a collection of systems that perform a particular business function, or of the systems in one location. However, if your role is different—for example, you’re a...
solution architect looking after a single system or an enterprise architect responsible for the whole systems landscape—much of what I describe here will still apply to you.

Timely
Architectural oversight must occur at the right times. Get involved in development projects early, when the architecture is still being formulated and important decisions are being made. Such decisions include

- what the system’s fundamental structure will be,
- how the system will manage data and use other systems’ data,
- how it will scale and be resilient and secure, and
- how it will use advanced or unfamiliar technologies or approaches.

These decisions are difficult to over-turn or change later during the project. For example, if your team starts out building a monolithic system, turning it into a collection of loosely coupled, semi-independent components will be a lot of work.

If you’re using waterfall or iterative development, expect to scrutinize the architecture at the start of higher-level design. There are strong arguments for you to be involved even earlier, while the architecture is still being formulated. This lets you give your team the benefit of your experience and knowledge of what works (and what doesn’t).

If you’re using agile methods, your interaction model will be a little more fluid. Most agile methods include some architecture envisioning early on, although this might be less precise than it would be for waterfall or iterative development. The architecture will likely evolve and crystallize during subsequent agile sprints or iterations. You’ll need ongoing oversight to ensure that the architecture is moving in a sensible direction and that important aspects aren’t being ignored.

Architectural oversight performed at the wrong time (for instance, after the big architectural decisions have already been made and acted on) has little chance of improving the system’s architecture.

Objective
Architectural oversight must be based on clearly stated principles, guidelines, and patterns, rather than on a subjective opinion. We all have our own ideas about how best to build software systems. An old joke says that if you put two architects in a room, they’ll come up with at least three ways to design a system. Although there are common architectural principles and patterns that we all work toward, there’s more than one “good” way to build a system. This lets you get involved in the process.

The goals of objectivity are to come to the same overall conclusions about the architecture regardless of who does the actual oversight and, specifically, how that person would have designed the architecture. This helps ensure a fair, objective process. A good way to achieve objectivity is to have a set of written guidelines, standards, and patterns that project teams can follow (or are required to follow). These guidelines should quickly lead team members to the right decisions, while leaving room for innovation and creativity.

Systematic
Architectural oversight must follow well-defined, repeatable processes whose objectives and outcomes are clearly understood. Systematic oversight, whether lightweight or thorough, will follow a defined sequence of steps with known inputs and outputs to achieve a well-defined goal in a finite (and hopefully relatively brief) time period. The participants are known, and everyone understands their role and responsibilities. The tasks are clearly defined, and all participants clearly understand the overall objective.

Ensure your outputs are recorded in enough detail so that they are understood, can be acted on, and are clearly communicated to your stakeholders. Itemize actions and close them when complete, and record decisions along with their rationale (why the decision was made) and implications (what must happen next). Clearly document exceptions involving noncompliance with policy, standards, or general good practice, along with the reasons for the exceptions.

If your oversight isn’t systematic,
your stakeholders won’t clearly understand who’s involved, why the oversight is being done, when it was started or finished, and its outcomes.

**Constructive**

Architectural oversight must lead to better architecture. Constructive oversight results in real, significant, and valuable change. In practice, this often means that the resulting architecture is more scalable, resilient, highly available, and secure. These qualities often receive less attention than they deserve when developers are ensuring that a system meets its functional requirements. However, a system that is unreliable, is too slow, or has security holes won’t be viewed as a success and might even be abandoned.

The most constructive oversight leads to specific architectural changes or improvements. For instance,

- functional components might be added,
- the system might be better designed for resilience or high availability,
- the system might be given scale-out or scale-up features so that it can continue to perform under high loads, or
- security features might be improved to better control access to sensitive functions or data.

If architectural oversight doesn’t lead to specific improvements, your stakeholders will view it as a waste of time or a bureaucratic form-filling exercise they must endure, rather than an important part of software development.

**Pragmatic**

Architectural oversight must take into account real-world constraints such as time, cost, and the availability of skills, without diluting the oversight’s purpose or effectiveness. You’ll most often have to deal with time and cost constraints. There’s no point in insisting on an architectural capability or feature if implementing it would substantially overrun or blow the project’s budget. You must be ready to compromise while ensuring the architecture’s overall integrity.

Start by ensuring that you and your stakeholders have a common understanding of the business importance of the system you’re looking at. This will help you pitch your oversight at the right level. You can then assess the benefits and risks to help you come to a decision.

For example, a proposed business-critical system with no disaster recovery capabilities would be a high risk because a significant incident could shut down the business. This aspect should be remediated before the system goes into production. On the other hand, it’s probably reasonable to not implement such a feature for a proof-of-concept system that will be rewritten anyway if it’s successful.

In any case, don’t expect everyone to understand and agree on all the details of a system’s architecture, especially at the early stages of a project. If some architecture aspect is still being developed, focus on tracking this to a decision rather than labeling the system as “non-compliant” in some way.

**Software Engineering Radio**

The podcast for professional developers is looking for hosts to interview some of the top minds in software engineering.

Contact bbbrannon@computer.org for more information.

**Sponsored by**

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

NICK ROZANSKI is a lead architect in a major UK bank’s Chief Technology Office. Contact him at nick@rozanski.org.uk.
We’ve got this big application. It’s been growing for two-and-a-half years or five years or 10 years, but we can’t maintain it anymore. It’s just too difficult to actually make any functional changes to it. We need to deploy this application into the cloud. We need software as a service, but at the moment that is impossible.

As a result of that, the ideas evolved of starting to splitting applications into smaller cooperating components that are running out of process and talking to one another, which can be maintained separately, scaled separately, or thrown away if needed.

A number of different communities have grown over time that have demonstrated that this approach to building software is viable for production. When you look at companies on the scale of Netflix, then it’s almost a necessity as they grow income. Adrian Cockcroft has said that they work this way because they want to build systems and make changes as fast as possible.

To answer your question, ‘Why is it so popular now?’ a lot of organizations have built up technical debt over the last number of years. They have realized that to scale more, to be more effective at delivering software into production, and to take advantage of things like continuous delivery, they need an approach that allows them to do scale along different axes independently of things like continuous delivery.

I think it’s about the right time for an idea like microservices to take off because a lot of companies are facing the same problems.

You said something interesting about how people with a large monolithic application are splitting it into microservices. Is there a typical form of introducing microservices?

That’s a great question and one I’ve actually been struggling with. It goes right to the heart of the question: do you start with microservices, or do you refactor to them later?

Empirically, most of the organizations have actually started with something big and have split that big thing up. That’s the case for most organizations that are building a microservices-style implementation. For example, Netflix. The canonical example is Amazon. Amazon started with a big database and then moved to a service-oriented architecture.

Let’s talk a little bit more about how you technically build a microservice. When I build a microservice for user authentication, what languages would I use? What standards do I build on, and what do I need to do to make it happen?

One of the guiding principles behind this is that you get the freedom to choose a lot of your tooling on a case-by-case basis. Rather than it being a particular language or particular back-end data store for your entire product stack, you get the flexibility to make informed decisions based on the right tooling for the situation at hand.

There are no right or wrong choices. If you’re talking about a user service, it is easily implemented in C#, Java, or any other modern programing language. Pretty much any programming language is going to be suitable.

The key thing is to make the stack lightweight. Rather than using the traditional heavy stacks and deploying them into big application containers (like JBoss and Tomcat), you can use lightweight alternatives, such as embedded Jetty, embedded Tomcat, SimpleWeb, or Weblt.

.NET-land is an interesting place at the moment because traditionally it has deployed into IIS. We’ve deployed all of our applications into this managed environment. But even in the .NET world, there’s been a movement to bring in some of their learnings from the Unix and Java communities around using embedded services. For example, we’re seeing more projects using a non-CFX alternative to some of their web APIs or MVC frameworks, and then using things like Owen. It’s about recognizing the centralization of the model that requires you to put all of your logic in one place. That place is the ESB, which provides all of the routing and data transformation required to get your applications talking to each other.

Is the “smart endpoint and dumb network” a reference to the Unix model?

It could be read like that. The reason we chose that name was more around the enterprise service bus (ESB) model. Inside Thoughtworks, for as long as I can remember, there’s been a tendency to distrust heavy iron when it comes to integration.

Big ESB products make a lot of promises about solving all your problems. I have seen a lot of implementations of “service-oriented architecture” with everything hanging off a big central ESB. I have never seen one of those succeed. It’s about recognizing the centralization of the model that requires you to put all of your logic in one place. That place is the ESB, which provides all of the routing and data transformation required to get your applications talking to each other.
Relying on one of these things to solve all your problems is, in my mind, not the right approach. There’s a great talk by Jim Weber and Martin Fowler called “Does My Bus Look Big in This?,” which they did as a keynote at QCon some years ago. Jim talked about the idea of the spaghetti box: the ESB as the panacea for all your ills. His line on that is it makes your diagrams look nice. You look at your enterprise architecture [diagrams], and they’ve got all these crossing ugly lines. It’s really tempting to put the ESB box in the middle because suddenly all your lines are straight. That’s a great thing if you’re an architect.

But of course all the lines are still there. They’re just in the middle of a spaghetti box. It still looks like a spaghetti box.

But when all the routing isn’t done by the ESB, who does the routing? Do I need to do the routing?

You certainly need to understand more about how your applications communicate with one another. If you’re building more services you end up with more integration problems. In the past, you might have been unlucky to talk to three external systems. Now you have to be cognizant of integration problems when you talk to your own systems. And there are ways to do that. Event-driven applications (with either publish-and-subscribe messaging, or HTTP and resource representation) allow you to decouple compared to using point-to-point RPC the whole time.

Isn’t that a bit like moving the complexity from the monolith into the networking layer?

The short answer to that is yes. Actually, when I originally talked to people about this, one of the great comments I got back, from Martin Fowler, was that we’re shifting the accidental complexity (in the sense that Fred Brooks used the term) from inside our application in glue code in our components and modules within our application out into infrastructure.

This is one of the reasons that now is a good time for this because we have many more ways to manage that complexity: programmable infrastructure, infrastructure automation, the movement to the cloud, the cloud being ubiquitous. Those sorts of problems, the problems of understanding how many applications we have, how they’re talking to one another—we have better tools to address those things now.

You mentioned domain-driven design in the beginning. Is microservices domain-driven design with a “service” label?

Microservices is the coming together of a bunch of better practices from a number of different communities. It is a combination of great stuff from the domain-driven-design community around strategic design, bounded context, subdomains, how to separate out your domains, and how to partition a very big problem domain into smaller domains so that you can manage them. It’s also taking a bunch of the better practices from operational automation and programmable infrastructure, de-
development operations communities, cloud communities, and the integration communities.

You’ve been working hard to make people aware they can solve integration problems using just the tooling available for free that drives the Web, without having to invest in big iron.

From the domain-driven-design community, the way you do “architecture” has to be driven from the business in the business context. You have to understand what the business problems are, what the business landscape looks like, and what the business processes are, and then drive a software product underneath that. For me, that’s the heart of domain-driven design.

One of my colleagues uses the great phrase “business and architecture isomorphism.” This is the idea that your business and the design of your systems should be very similar. When you look at your business, you should see your IT systems and look at your architecture and see your business. If you’re a technologist or business person, there should be recognition both ways that this is going on.

How big are these services?
That’s something we’ve been talking about internally for quite a while. I’ve seen them ranging from a couple of hundred lines of code up to a couple of thousand lines of code. The guidance I’ve been giving people is it does one thing and one thing only. It’s difficult to imagine a million lines of code doing one thing and one thing only. The guidance is you should be able to understand them. They should have a single reason to change, and they probably shouldn’t be more than a couple thousand lines of code.

When you get to that point, the number becomes important. It’s probably more important to think about how many of them you’re capable of supporting operationally than it is to think about how small they actually are because it’s better to have slightly bigger ones and fewer of them if you don’t have fully automated deployment into production.

JOHANNES THÖNES is a developer and consultant for ThoughtWorks. Contact him at johannes.thoennes@gmail.com.


Postmaster: Send undelivered copies and address changes to IEEE Software, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.

Reuse Rights and Reprint Permissions: Educational or personal use of this material is permitted without fee, provided such use: 1) is not made for profit; 2) includes this notice and a full citation to the original work on the first page of the copy; and 3) does not imply IEEE endorsement of any third-party products or services. Authors and their companies are permitted to post the accepted version of IEEE-copyrighted material on their own webservers without permission, provided that the IEEE copyright notice and a full citation to the original work appear on the first screen of the posted copy. An accepted manuscript is a version which has been revised by the author to incorporate review suggestions, but not the published version with copyediting, proofreading, and formatting added by IEEE. For more information, please go to: http://www.ieee.org/publications_standards/publications/rights/paperversionspolicy.html. Permission to reprint/republish this material for commercial, advertising, or promotional purposes or for creating new collective works for resale or redistribution must be obtained from IEEE by writing to the IEEE Intellectual Property Rights Office, 445 Hoes Lane, Piscataway, NJ 08854-4141 or pubs-permissions@ieee.org. Copyright © 2015 IEEE. All rights reserved.

Abstracting and Library Use: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy for private use of patrons, provided the per-copy fee indicated in the code at the bottom of the first page is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.
Microservices

Johannes Thönes

ONE GOAL of the Software Engineering Radio podcast is to be a source of information about the latest architectural trends. Trends emerge from practice and take a while to show up in written form. The first book on microservices isn’t due until spring 2015. For the professional software engineer, conferences, online talks, and podcasts are often the best sources for the newest information.

In this month’s podcast (episode 213), Johannes Thönes talks with James Lewis about microservices. This podcast is the third one in the fall schedule to address this topic. In episode 210, Stefan Tilkov discusses architecture and microservices; in episode 216, Netflix architect Adrian Cockcroft discusses the cloud-based platform. The upcoming episode 217 on the Docker container covers a popular piece in the deployment of these systems (see the sidebar).

The following excerpt contains only a fraction of the show. Space didn’t permit us to include discussions covering the relationship between microservices and Conway’s law, CQRS (Command Query Responsibility Segregation), REST (representational state transfer), operational complexity and the impact on development operations, “isn’t this just SOA?,” agile development, testing, and monitoring.

I hope you’ll download the entire show and listen. —Robert Blumen

What’s a microservice?

A microservice, in my mind, is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility. It is a single responsibility in the original sense that it’s got a single reason to change and/or a single reason to be replaced. But the other axis is a single responsibility in the sense that it does only one thing and one thing alone and can be easily understood.

What would such a single thing be?

An example of a single thing might be a single responsibility in terms of a functional requirement, or it might be in terms of a nonfunctional requirement or, as we’ve started talking about them, cross-functional requirements.

An example might be a queue processor—something that’s reading a message from a queue, performing a small piece of business logic, and then passing it on. Or it might be something that’s cross-functional, or nonfunctional, or it might be something that has the responsibility for serving a particular resource or resource representation.

Like a user.

Like a user or, say, an article, or it might be a risk in insurance or something like this, but something that’s very focused and very small and that performs a single task on its own.

I have the impression that microservices have become quite popular. Why do you think that is?

Continued on p. 113
Focus on Your Job Search

IEEE Computer Society Jobs helps you easily find a new job in IT, software development, computer engineering, research, programming, architecture, cloud computing, consulting, databases, and many other computer-related areas.

New feature: Find jobs recommending or requiring the IEEE CS CSDA or CSDP certifications!

Visit www.computer.org/jobs to search technical job openings, plus internships, from employers worldwide.

http://www.computer.org/jobs

The IEEE Computer Society is a partner in the AIP Career Network, a collection of online job sites for scientists, engineers, and computing professionals. Other partners include Physics Today, the American Association of Physicists in Medicine (AAPM), American Association of Physics Teachers (AAPT), American Physical Society (APS), AVS Science and Technology, and the Society of Physics Students (SPS) and Sigma Pi Sigma.
3D Printing Will Actually Change the World! Are You Ready?

Every company needs to prepare and implement 3D printing in order to remain relevant in their industry! No one can sit this phenomenon out!

Get ready at Rock Stars of 3D Printing, a one-day event featuring the experts, early adopters, and visionaries that are driving this revolution.


Here’s a list of other Rock Star speakers for Rock Stars of 3D Printing:
- Paul Brody, Vice President and Global Industry Leader of Electronics, IBM
- Brian David Johnson, Futurist and Director, Future Casting and Experience Research, Intel
- Cliff Waldman, Council Director and Senior Economist, Manufacturers Alliance for Productivity and Innovation

17 March 2015
The Fourth Street Summit Center
San Jose, CA

REGISTER NOW
Early Discount Pricing Now Available!

computer.org/3dprinting