MAY/JUNE 2013

WWW.COMPUTER.ORG/SOFTWARE

# IEEE Software

## SAFETY-CRITICAL SOFTWARE



IEEE

IEEE computer society


# Two Conferences in One Location

*Register and attend sessions from both events!*

## Super Early Bird Savings!

REGISTER BY APRIL 5 AND
**SAVE UP TO $400**
GROUPS OF 3+ SAVE EVEN MORE!

EXPLORE THE FULL PROGRAM AT
**adc-bsc-west.techwell.com**

*PMI® members can earn PDUs at both events*

PMI R.E.P. Project Management Institute

IEEE

IEEE computer society
The Community for Technology Leaders

# Focused on Your Future

Now when you join or renew your IEEE Computer Society membership, you can choose the membership package focused specifically on advancing your career:

- Software and Systems—includes *IEEE Software* Digital Edition
- Information and Communication Technologies (ICT)—includes *IT Professional* Digital Edition
- Security and Privacy—includes *IEEE Security & Privacy* Digital Edition
- Computer Engineering—includes *IEEE Micro* Digital Edition

In addition to receiving your monthly issues of *Computer* magazine, hundreds of online courses and books, and savings on publications and conferences, each package includes never-before-offered benefits:

- A digital edition of the most requested leading publication specific to your interest
- A monthly digital newsletter developed exclusively for your focus area
- Your choice of three FREE webinars from the extensive IEEE Computer Society selection
- Downloads of 12 free articles of your choice from IEEE Computer Society Digital Library (CSDL)
- Discounts on training courses specific to your focus area

**Join or renew today at www.computer.org/membership**

IEEE Software
TABLE OF CONTENTS
May/June 2013



## FOCUS

### SAFETY-CRITICAL SOFTWARE

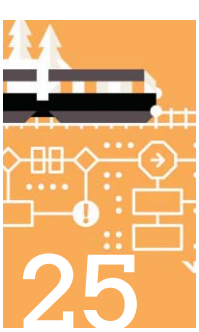


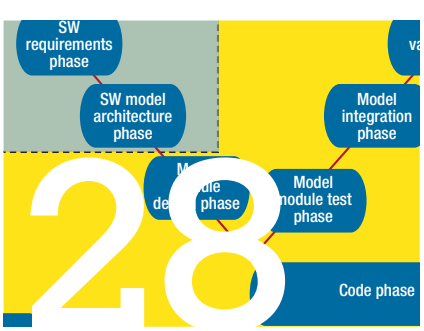
## FEATURE

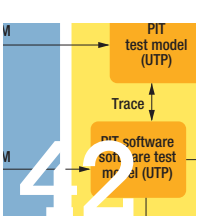


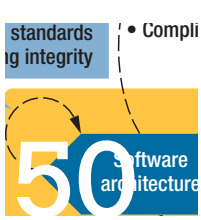
### INSIGHTS

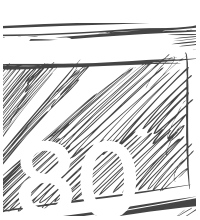





Building the Community of Leading Software Practitioners
www.computer.org/software

# DEPARTMENTS



## MISCELLANEOUS



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



For more information on computing topics, visit the Computer Society Digital Library at www.computer.org/csdl.

FROM THE EDITOR

Editor in Chief: **Forrest Shull**
Fraunhofer Center for Experimental Software Engineering, fshull@computer.org

VIDEO

# Sharing Your Story

Forrest Shull

***IEEE SOFTWARE* ACCEPTS** less than 25 percent of the articles submitted for consideration, and I'm keenly aware that all of those submissions—whether eventually accepted or rejected—entail many hours of effort on the part of authors, reviewers, and magazine staff.

In addition to being selective, *IEEE Software* is also a somewhat unique venue. Our mandate is to be the authority on translating software theory into practice—meaning that while we're interested in rigorous and well-tested research results, those results also need to be explained in a way that can reach our intended reader (the reflective software practitioner) and help her understand something important about the software profession. For this reason, we prioritize writing with an accessible style and relatively tight word limits. Good advice for juggling these constraints can be found in articles written by my predecessors as editor in chief, Steve McConnell and Hakan Erdogmus.[1,2] To add to what they've written, in this article, I'd like to focus on a special type of submission: the experience report.

> Our mandate is to be the authority on translating software theory into practice.

I'm an advocate of experience reports because I'm a firm believer in just about any approach that stands a chance of improving the communication between software research and practice. Here at *IEEE Software*, we receive fewer experience reports than other types of submissions, and this is understandable. I realize how tough it can be, as a professional developer, to get the time to reflect on an experience and write about it, and I appreciate those who do so. Although it can be difficult, the effort to produce an experience report is almost always rewarding and helps the author reflect on the true causes for success and failure amid all the noise and pressure of day-to-day deadlines. Well-written experience reports can be among the most compelling pieces that we publish in *IEEE Software*.

In this article, I'd like to take the time to reflect, myself, on what we are looking for in experience reports and provide some guidance that can help authors.

## What Is It, and What Does It Do?

An increasing number of conferences and periodicals in software engineering are featuring experience reports. From a quick and admittedly subjective perusal of the author guidelines, however, the calls for experience reports often seem to suffer from the lack of a clear definition of exactly what is being sought. This is a danger because—especially in research-focused venues—without a clear definition, experience reports are often perceived as the place to send work that won't be accepted in normal technical tracks.

But experience reports are an important type of article in their own right—not just technical pieces that didn't quite make the bar. Experience reports should provide a benefit that more "traditional" research studies cannot: this is a bit of an oversimplification, but let's call this benefit "depth"—that is, a more detailed and nuanced understanding of what happened in a single environment (or single project). Experience reports, in describing a single environment, can only describe what happened

*IEEE Software* Mission Statement: To be the best source of reliable, useful, peer-reviewed information for leading software practitioners—the developers and managers who want to keep up with rapid technology change.


0740-7459/13/$31.00 © 2013 IEEE

## SOFTWARE EXPERTS SUMMIT 2013

One of my goals as editor in chief has been to find ways to get the excellent experience reports and the latest research found in every issue of *IEEE Software* to software practitioners in new and more convenient ways. For example, we've been reaching out through our digital edition, through new media such as audio and video files, and through discussion forums. One of the most exciting of these new initiatives is the Software Experts Summit, a public event that showcases many of the thought leaders associated with the magazine during a day of presentations, panel discussions, and networking.

Timed to coincide with the publication of our upcoming July/August 2013 issue looking at the impact of software analytics on decision making ("Software Analytics—So What?"), our theme this year is "Smart Data." We'll be tackling the question of how organizations can best make reliable, secure, and quick decisions on datasets of many types, despite the challenges we all face with

- using even small sets of data to guide decision making due to inconsistent data structures,
- making sense of the incredible diversity of data and media in which they are embedded, and
- effectively using the technologies that create and manage data.

The event will be Wednesday, 17 July 2013, at the Microsoft Conference Center in Redmond, Washington. For more information, see our website at www.computer.org/ses13 and the ad on the back cover of this issue.

to the authors; they don't provide sufficient data to argue that if other teams follow the same approach, they can confidently expect the same outcome. To make up for this lack, a good experience report provides enough of a narrative to discuss with confidence *why* a certain result was seen.

*IEEE Software* is interested in publishing experience reports for a number of reasons. In my mind, the most important is that they help keep research grounded. Our field has self-organized in such a way that many software researchers aren't familiar with the contemporary experience of working in a software development environment, and sharing that vision can help keep research focused on compelling problems and help produce results capable of operating under reasonable constraints. Software professionals can also benefit from hearing about what development is like in other contexts. None of us have the time or opportunity to experience all types of environments, and many of us can find some benefit in looking at practices in other types of organizations. A developer in Silicon Valley, for instance, might find some value in looking at practices on systems at NASA, and a NASA developer might find value in understanding more about the development of mobile apps.

Other reasons for valuing experience reports are that they can often provide the most practical advice to practitioners. I often respond with more interest and curiosity to someone telling me a story ("Oh, look, someone who is doing similar work to mine swears by this tool—I think I'd better give it a closer



IEEE Software

STAFF

Lead Editor
**Brian Brannon**
bbrannon@computer.org

Manager, Editorial Services
**Jenny Stout**

Editors
**Camber Agrelius and Linda World**

Publications Coordinator
**software@computer.org**

Production and Design Editor, Webmaster
**Jennie Zhu-Mai**

Contributor
**Alex Torres**

Cover Artist
**Eero Johannes**

Director, Products & Services
**Evan Butterfield**

Senior Manager, Editorial Services
**Robin Baldwin**

Senior Business Development Manager
**Sandra Brown**

Membership Development Manager
**Cecelia Huffman**

Senior Advertising Coordinator
**Marian Anderson**
manderson@computer.org



**Editorial:** All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by IEEE or the IEEE Computer Society.

**To Submit:** Access the IEEE Computer Society's Web-based system, ScholarOne, at http://mc.manuscriptcentral.com/sw-cs. Be sure to select the right manuscript type when submitting. Articles must be original and not exceed 4,700 words including figures and tables, which count for 200 words each.

**IEEE prohibits discrimination, harassment and bullying:** For more information, visit www.ieee.org/web/aboutus/whatis/policies/p9-26.html.

## MULTIMEDIA EDITOR SOUGHT

Given our effort in moving beyond print, the position of multimedia editor is a central and important one for the magazine. We're currently looking for candidates who would be interested in taking on this role.

The role entails:

- overseeing the multimedia production schedule, and making sure that we have sufficient multimedia pieces allocated for upcoming issues;
- coordinating with our department editors, special issue guest editors, Software Engineering Radio (www.se-radio.net) podcasters, the EIC, and IEEE Computer Society staff to track progress and suggest opportunities; and
- suggesting hot topics and important thought leaders that could be the focus of work by our multimedia teams.

This is a high-visibility position and one that provides the opportunity to interact with software engineering thought leaders. Moreover, the multimedia editor will be working with a great, productive, and fun team.

Interested? Please contact lead editor Brian Brannon at bbrannon@computer.org for more information or to send an application, which should be comprised of a cover letter and resume or CV. Applicants should have a proven ability to manage projects and deliver reliably.

We express our sincere thanks to Bob Rosenstein of the Software Engineering Institute. Bob served as multimedia editor for the first year of our digital edition and helped define the role. We're grateful for the guidance and help he gave us in getting this important new project off the ground, and wish him all the best with his new responsibilities at the SEI!

look") than to reams of data ("This tool vendor claims to have reduced the amount of effort needed for the job and claims that 9 out of 10 customers are highly satisfied"). I assume that other humans are motivated in similar ways.

Finally, experience reports can provide fast feedback to the community on new technologies or approaches being advocated. Long before anyone can have enough data to start to consider statistically significant effects, we may be able to share success (or failure) stories from individual projects. These should be taken with the appropriate caveats, of course, just like any study. But if the results of the experience report are compelling, they can help readers understand whether this is an area worth expending time and effort on.

Why do I mention all of this? Mainly so that prospective authors can use this as fodder for their own article reviews prior to submission. The single most important thing that any author can do as part of a self-critique is to think of the reader. Will an experience report help a reader keep up with what she needs to know to be effective in the software profession?

## Tell Me a Story

Potential authors who ask for feedback from me on abstracts of planned papers will almost always get a response structured around the following set of questions.

### Environment

Is it clear what type of environment your story takes place in? Other readers would like to benefit from your insights, but they need to have a good sense of how likely your findings are to translate to their projects. If you're building a website app and I'm building embedded satellite software, I might find your story thought-provoking, but I might approach the idea of applying the same techniques in my work more carefully.

### Focus

Is it clear what you did? What method, tool, or practice did you apply? In short, what is your story about? If a reader finds your experience compelling and is willing to try it out in his own work, would he know what to do or where to get more info? Above all, keep focus. Don't describe everything you did on the project. Be ruthless in down-selecting to just those facts that support the coherent story you're trying to tell. When choosing the focus of that story, keep in mind that *IEEE Software* has a broad coverage area, and we're interested in methods and tools related to the nuts and bolts of software development as well as management and human factors issues.

### Results

What were the results of what you did—and how do you know that those results were caused by the method, tool, or practice you're advocating? Our reviewers are looking for reports that describe a concrete result. If you're telling me a story that revolves around applying a new approach (let's say an automated tool that attempts to detect hidden technical debt items), you have to tell me the end of the story. How well did the tool work? Was the project a success—and was that success traceable back to the tool in any meaningful way?

When it comes to describing results, there are other issues to con-

sider. How do you know that your results really mean? And how would a reader have confidence that your story can be trusted? We don't expect experience reports to have reams of hard, quantitative data, but there are other ways of addressing this issue. When appropriate, these might include subjective forms of evidence such as feedback from key stakeholders or management—in this case, the more specific the author can be, the more convincing the story tends to be. Direct quotes can be helpful in this regard. Comparison to prior projects is always useful, as a way to show what has changed as a result of the new approach. Often, what the author's organization is willing to do on the basis of the results speaks volumes. If the results are convincing enough to impact day-to-day practices across other projects, then they're probably compelling enough for readers to pay attention to.

Also, when describing results, authors shouldn't claim to have found a silver bullet. Readers appreciate a careful weighing of pros and cons and it's very rare indeed to be able to see progress on one dimension without tradeoffs on others. Truly great experience reports are those that look at multiple types of impact—say, a tool's impact on the eternal triangle of project cost, quality, and schedule. If a tool really helps improve the delivered quality of a product, what does a project have to give up for that result—a substantial amount of extra effort? An impact on the schedule? And how about one-time costs like investments in training?

## Where to Go from Here

I welcome experience reports submitted through the usual channels. But if all of the above constraints seem daunting, don't despair. Our Insights department, helmed by Linda Rising, was established especially to help—in fact, the one on page 9 of this issue focuses on stories. Proposals to Insights are reviewed by Linda and her distinguished advisory board and, if accepted, shepherding is provided. Please see Linda's inaugural column for much more helpful information and guidance.[3]

If I could boil all of this guidance down to a simple test, it would be this: Is there more to an article than just a description of, "we did this" or "we built this"? Is there a meaningful principle exemplified through an experience report that readers will care about, be intrigued by, and possibly think of applying themselves? Linda has compared a good experience report to a project retrospective: "We not only want teams to look back and say what happened, but we also want analysis."

I couldn't put it better than that. And, like Linda, I remain excited by the idea of hearing more reflection and analysis from the ambitious software development projects going on throughout the industry today—with results we can all learn from together.

## References


1. S. McConnell, "How to Write a Good Technical Article," *IEEE Software*, vol. 19, no. 5, 2002, pp. 5–7.
2. H. Erdogmus, "Tips for *Software* Authors," *IEEE Software*, vol. 24, no. 5, 2007, pp. 5–7.
3. L. Rising, "Telling Our Stories," *IEEE Software*, vol. 27, no. 3, 2010, pp. 6–7.


**FORREST SHULL** is a division director at the Fraunhofer Center for Experimental Software Engineering in Maryland, a nonprofit research and tech transfer organization, where he leads the Measurement and Knowledge Management Division. He's also an adjunct professor at the University of Maryland College Park and editor in chief of *IEEE Software*. Contact him at fshull@computer.org.



IEEE Software

# HOW TO REACH US

**WRITERS**

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access www.computer.org/software/author.htm.

**LETTERS TO THE EDITOR**

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

**ON THE WEB**

www.computer.org/software

**SUBSCRIBE**

www.computer.org/software/subscribe

**SUBSCRIPTION CHANGE OF ADDRESS**

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

**MEMBERSHIP CHANGE OF ADDRESS**

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

**MISSING OR DAMAGED COPIES**

If you are missing an issue or you received a damaged copy, contact help@computer.org.

**REPRINTS OF ARTICLES**

For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

**REPRINT PERMISSION**

To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.


IEEE Software

**Now Available**
in Enhanced Digital Format

## More value, more content, more resources

The new multi-faceted *IEEE Software* offers exclusive video and web extras that you can access only through this enhanced digital version. Dive deeper into the latest technical developments with a magazine that is:

**Searchable**—Quickly find the latest information in your fields of interest. Access the digital archives, and save what's most relevant to you.

**Linked**—Click on table of contents links and instantly go to the articles you want to read first. Article links go to additional references to deepen your new discoveries.

**Engaging**—Experience concepts as they come to life through related audio, video, animation, and more. Email authors directly. Even apply for jobs through convenient ad links.

**Mobile**—Read the issue anytime, anywhere, at your convenience—on your laptop, iPad, or other mobile device.

**computer.org/software**

IEEE | IEEE computer society

INSIGHTS: **STORYTELLING**

# Storytelling for Software Professionals

**Arjen Uittenbogaard**, inspearit

The first book I read about storytelling was by David Armstrong, then vice president of Armstrong International. I was skeptical at first, but got caught up in the stories. As I got more into patterns, I found that in trying to tell readers about my context, problems, forces, solutions, and resulting context, I was simply trying to tell a good story. Over time, I learned that the better the story, the more useful the pattern. I still believe this 20 years later. The following article is not only about storytelling, but it tells its own good tale. So, sit back and enjoy! —*Linda Rising, Associate Editor*

**AS LONG AS** humans have existed, stories have been told. It's my claim that we as software designers and IT architects can benefit much more from this ancient craft than we currently do. A good story is much more powerful than any UML diagram can ever be. Telling stories can effectively communicate your message; listening to them helps you better understand your users' needs. Crafting stories together with your stakeholders builds a common vision of the system.

## Getting Attention

When I first started training people about agile development, my presentations contained many slides on Scrum. They had descriptions of the different roles and responsibilities, the types of work products, and the distinct types of meetings. You've probably had similar experiences in projects with presentations for management teams, user groups, or analysts. Your slides showed the details, covered all the arguments, and came to a well-founded conclusion and advice. Nevertheless, I'm sure that those presentations didn't resonate with your audience. People tried to comprehend the information and occasionally even asked a sharp question about something on a slide, but most of the time, they were looking at their watches as their thoughts wandered out of the room.

One day, I decided to do things differently. I crafted a story about the situation in our project and used it at the beginning of my presentation. As soon as I started with "Once upon a time," I noticed everyone perk up. During the tale, people stayed with me and I even saw a smile or two. The story was a welcome break between the other talks. Sure, it didn't cover all technicalities, but at least everyone paid attention and could retell it. Isn't that a giant step forward, to have a shared basis, a common ground for all stakeholders?

What would you rather listen to, a story about people you can relate to—people of flesh and blood, with real feelings, who are puzzled by problems that seem real and relevant for the situation at hand—or a PowerPoint presentation that covers the basic facts in excruciating detail? Which of the two would you remember the next day or maybe even the following year?

### Example 1: Stories about Your Message

Our home-grown object request broker had to be replaced. For years, we kept building on it, but now it was very brittle. We had already raised the issue a couple of times, but convincing management that we needed serious investment in it wasn't easy: "We've managed fine so far—why buy expensive new software if the old stuff still works?"

0740-7459/13/$31.00 © 2013 IEEE

## CRAFTING A STORY: GUIDELINES FOR DOING IT YOURSELF

I like metaphoric stories very much, but maybe you prefer more realistic ones. Stories can be anecdotes or they can be entirely made up. In my experience, either of these classes can work or fail. Here are some basic guidelines for crafting stories that work:

- Whether the story is realistic or metaphoric, anecdotal or made up, it has to be true, meaning the message it conveys must be authentic. Otherwise, the story reads like a lie, and your audience will know; they will sense that something is wrong, and you won't be believed.
- Every story has a protagonist. This is the one character that your audience will empathize with, the hero. When looking for a metaphoric story, freely associate to find your protagonist. Ask yourself, if this organization/problem/system were a vehicle, what vehicle would it be? What if it were an animal, fairy tale character, or other profession?
- Stories with too many themes or messages will be complex at best but more often tend to be boring. Kill your darlings. Less is more.
- Your story will be about how the hero reaches his or her goal by overcoming one or more obstacles. Of course, tragic stories exist in which the hero doesn't reach this goal, but for the purposes of motivation and inspiration, these endings aren't appropriate.
- Try it out. Test your story by telling it to different audiences and finding out if its message comes across. If not, keep adapting the story until it does.

In this context, I wrote a fairy tale about a castle (see the sidebar) and started telling it. It worked. The story had an open end for starting the conversation: "Dear CEO, if you were the king, what would you do?" As if by magic, this led us into the discussion with our management about business-IT alignment that we had been looking for. Before we had this story, we were talking in technical terms to an audience that didn't understand or care. Talking in the domain of a fairy tale, however, paved the way to a real discussion of the problems at hand.

### Example 2: Feeling the Pain

I tell the story about the Soviet economist in Rotterdam a lot (see the sidebar), especially in coaching and trainings. I can explain to software designers or IT architects why big design up front doesn't work, and I can explain to managers that in a complex world, it's no longer possible to control everything. But if I use complexity theory and systems thinking to explain it, everyone nods, thinking they understand. However, they won't actually feel the angst that comes with letting go. They won't feel the fear that comes with having to trust others. Until you have felt that fear yourself, you don't truly understand the message. The story about the Soviet economist brings you closer to this feeling. You can empathize with the Soviet economist and with his despair: "How can this possibly work?!"

### Listening and Understanding

The power of stories can also be applied when gathering information. Call it information analysis or requirements engineering—your task is to find and listen to the stories being told.

Up until about 15 years ago, our requirements specifications and information analyses consisted of long lists of statements that started with "The system shall…" or "The system should…." Although this wording and format was sometimes required for legal reasons, seeing the forest for the trees was difficult. Then Ivar Jacobsson introduced use cases, which provided a logical structure for users: one use case per user goal. Of all techniques in the UML, use cases come closest to stories. After all, they have a protagonist (the user) and a series of events leading that person toward a goal. But they're intentionally analytical—if they're stories, they're stories with all the life sucked out of them. Before we can specify use cases, we have to go out and talk to users. They have to tell us about their work, about what they like and where improvements are needed, about where they're satisfied and where they see room for improvement. Users tell us stories, which we turn into use cases.

### Learning and Sense Making

I firmly believe that having templates for work products will never be sufficient. Only by using and sharing your experiences with them and swapping anecdotes will you get better at your job. Therefore, I urged the members of one team for quite some time to stop discussing the templates and start using them. At every retrospective, I asked for volunteers to tell about their experiences. For several months no one did, until one day Enno told his story.

He had been working on the vision document for the project and was trying to fill the template: "It was of no

## EXAMPLE 1: THE CASTLE FAIRY TALE

Once upon a time, there was a king and a queen. They were happy together and ruled the country to the satisfaction of all. They lived in a beautiful castle with a large hall, a guest tower, a huge inner court, and a moat surrounding it. One beautiful day, a baby prince was born. As the boy grew up, he would need to have his own rooms to sleep, play, exercise, and study in, so while the queen was still pregnant, carpenters and servants emptied, cleaned, and redecorated one of the towers for him. When the prince was two years old, a second prince was born. And shortly after that, two princesses and another prince entered the family. Of course, all were entitled to their own rooms, so every time a baby was born, the castle was restructured slightly. Fortunately, not everything needed to be built anew each time—some rooms could be shared. Nevertheless, some puzzling was required with the arrival of every little prince or princess.

The children grew up, fell in love with other nearby royals, and married. The king and queen invited the couples to come and live in the castle, which they all did, but as they required more privacy than they did as children, the castle again had to be renovated. Towers were made higher. Some of the rooms at the exterior of the castle were extended over the moat. Parts of the inner court were claimed for new rooms. Luckily, the king employed a great master builder and skillful masons and plumbers. No challenge was too big. Soon, the large hall wasn't large anymore, and the inner court had to renamed "courtlet."

After some more years, the first grandchildren were born. Like their parents before, they too were entitled to their own rooms. Annexes were enlarged and supported with ingenious scaffolding. Arches were built between towers to support them, and once an arch was in place, it in turn provided more space to build upon. But when the fifth baby of the fifth prince was born, all options were exhausted. Until then, the master builder had been able to find solutions time and again. But now, the plumbing had become an intricate knot. The interior walls blended into exterior walls in such ways that no further rebuilding was possible. If the towers went any higher, they would collapse.

The master builder was desperate and asked for an audience with the king. For this youngest grandchild, there was no room in the castle…

use. It was as if I had to invent everything all over. None of the sections made sense." He talked about the difficulties he had encountered, including stakeholders that he couldn't reach and the sponsor who didn't want to talk about his problems and needs. As his story went on, we learned how he had dealt with these situations: "The template was of no use, so I created my own spreadsheet. It helped me collect the minimal information I needed." The interesting thing was his conclusion: "I have learned a lot about making a vision. Looking back, I think I could have used the template after all, if only I had understood it better." Not only had Enno described some lessons he had learned, but by telling his story, he also taught us a valuable lesson: there will be obstacles, but they can be conquered. (Also, be pragmatic about those templates.)

## EXAMPLE 2: A SOVIET ECONOMIST IN ROTTERDAM

Many years ago in the Soviet Union, the Kremlin made five-year plans that, I'm told, determined who was allowed to buy what in which quantities on what date and with which supplier. Back in those days, a Soviet economist was visiting a Dutch colleague for a conference at Rotterdam University.

They were driving along the highway when the Dutch professor took an exit to a gas station to fuel his car. When they got back on the road, the Soviet looked puzzled: "What a remarkable coincidence…." The Dutchman asked what he was wondering about. "Well, exactly on the day you're allotted this amount of gasoline at this gas station, you're actually passing it on your way to the conference!" Now the Dutchman was puzzled, which provoked the Soviet economist to say, "Ah, of course! Because you're a professor, you have some privileges." The Dutch professor shook his head. "Or it is because of the conference: you're allowed to fuel up during these days?" When the response remained negative, a worrying thought dawned on the Soviet: "Are you telling me that each and every car on this highway is allowed to buy any amount of gasoline at any gas station at any time he desires?"

The Dutch professor acknowledged that things went like that over here. "But that can't possibly work!" the Soviet exclaimed.

## ABOUT THE AUTHOR

**ARJEN UITTENBOGAARD** is a trainer, coach, and storyteller at inspearit, working in requirements engineering, agile teamwork, and scenario planning. At the Saturn 2012 conference, his presentation "Mythology for IT Architects" won the New Directions Award. Contact him at arjen.uittenbogaard@inspearit.com or arjen@verhalenmaker.nl.

### Finding a Common Vision

Kent Beck introduced the system metaphor to guide all development: a simple shared story about how the system works. For example, comparing the system to a castle gives business and IT stakeholders a similar vision. The entire team chooses the system metaphor, typically at project kickoff. Using association and discussing which metaphor resonates best helps the team jell and build a common vision.

If a metaphor is a picture worth a thousand words, I would suggest that a story is even more valuable. In the context of a component-based development project, the architects wanted a good metaphor for their generic components. They wanted this metaphor to help them convince projects to reuse components instead of building everything themselves. In the metaphor workshop, two architects brought stories in which they compared themselves to traveling salesmen trying to sell their goods. The third architect told the story of the stone soup (http://en.wikipedia.org/wiki/Stone_soup). The difference was striking. Up to that point, the architects had been trying to push their components to reluctant projects. The stone soup story suggested a radically different approach: let's make the component library so attractive that the projects will gladly contribute to it! This insight had a great impact, and from that point forward, instead of drafting blueprints and making projects comply, the architects started cooperating and harvesting whatever useful components the projects brought to the table.

Of course, stories aren't the panacea for every problem at hand, and of course, you might still fail when using a story. If you didn't craft a good one, the wrong message might come across. Sometimes your audience is only in the mood for hard facts. Certainly, telling too long a story can do harm. But I've learned that most often you yourself are the biggest obstacle: if you aren't convinced that telling the story will work, your hesitation will show. If you find yourself in this spot, try out your story in a safe-to-fail environment.

Most of the time, stories do work, so tell them, listen to them, and, working with others, craft them.



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

REQUIREMENTS

Editor: **Jane Cleland-Huang**
DePaul University,
jhuang@cs.depaul.edu

# Are Requirements Alive and Kicking?

Jane Cleland-Huang

**ABOUT 10 YEARS** ago, I attended a large meeting for requirements engineers and business analysts in Europe. At that time, one of the organizers asked how many of us thought agile methods would significantly impact the way mainstream companies developed software and managed requirements. None of us raised our hands.

In hindsight, of course this seems particularly shortsighted. In our defense, most of the people in that room had invested a significant amount of time developing techniques for improving the requirements process. We knew how important it was to identify relevant stakeholders, proactively elicit requirements, analyze them carefully, identify trade-offs, emerge and negotiate conflicts, and specify clearly and unambiguously what the system needed to do and how it needed to be done. At that time, the agile movement seemed to be telling us to throw these practices away, threatening the whole concept of requirements as we knew it.

On the other hand, around the same time, I coauthored a book with Mark Denne, then from Sun Microsystems, called *Software by Numbers: Low Risk, High Return Development* (Prentice Hall, 2003), in which we laid out a highly incremental, ROI-driven approach to prioritizing and sequencing features. What I didn't then realize is that we had serendipitously discovered an idea that fit naturally into the agile development process. Our approach still assumed that we would identify stakeholders and work with them to discover their needs, but instead of laying out a long-term development plan, it used a simple financial analysis to figure out which features (architectural or functional) to build first. Furthermore, the approach embraced change by allowing the upcoming set of features to be determined as the project progressed. Consequently, I came to realize how requirements and agility could live side by side.

> I came to realize how requirements and agility could live side by side.

So, why am I focusing on agility in a column on requirements? For my debut installment of this column, I want to discuss the challenges that impact requirements processes today and the context in which most of us deal with requirements. Agility is certainly one of the major influences.

## Changing Times

As an academic, I've had the opportunity to engage in industrial projects of many sizes. Some of these projects are very linear in nature and basically follow the typical waterfall model, whereas others are agile and fully embrace concepts such as informal user stories, short delivery cycles, and changing requirements. From my perspective, both kinds of projects have had their own challenges, but in the end, both kinds of projects have been successful.

The bottom line is that the IT environment we work in today is simply different from the environment 10 years ago, and for this column to continue to address pertinent topics, it's essential that we understand the critical forces that drive requirements today.

For example, we can no longer assume that projects will be preceded by stringent, up-front processes in which we carefully elicit all the requirements and then design a complete solution before jumping into the actual development process. There are many assumptions that we might have made a decade ago that simply aren't true today.

We can no longer assume that it's possible to bring all stakeholders

0740-7459/13/$31.00 © 2013 IEEE

together in the same room at the same time for face-to-face meetings. In many current projects, stakeholders are distributed around the globe, and sometimes the only way to reach them all is by leveraging social networking and collaboration tools. This introduces numerous challenges. How can we make our tools more effective so that meetings made up of people from different continents produce meaningful discussions that bring forth real issues and result in a deep understanding of stakeholder needs?

We're gaining a far deeper understanding of the synergies between requirements and architectural design. In fact, in many cases, new requirements are introduced to an existing system, and previous architectural, platform, and hardware decisions create constraints on the viability of new feature requests. Although it's true that almost anything is possible, it's also true that some requirements are far easier to deliver than others and that existing solutions can facilitate or hinder future change. So, how does this more incremental life cycle affect the way we elicit and manage requirements?

Software development environments (not just the stakeholders) are becoming increasingly global in nature. When the outsourcing trend began a decade or so ago, we had fairly clear hand-off points in the life cycle. That line is far fuzzier now. Outsourcing companies are now often responsible for not just coding and testing but also the requirements elicitation process. We must consider how this affects project success and how we can ensure that project goals and requirements are fully explored and understood when the people eliciting the requirements aren't even on the same continent as the primary stakeholders.

## New Demands

There's also a move to push up the abstraction level of the software development effort by specifying software requirements as models and then automatically generating code from those models. In the world of model-driven development (MDD), how do we ensure that requirements are specified correctly in models and verify that the generated code actually satisfies stakeholders concerns?

Yet another challenge comes from the increasing desire for systems, especially Web services, to adapt at runtime to changes in their environments. One of my graduate students recently told me that his company's attempt to achieve adaptation failed primarily because it didn't know how to clearly specify requirements for adaptive systems, and therefore never fully understood the system's goals. As industry increasingly moves toward building adaptive systems, we must find better ways to specify adaptation goals.

And what about safety-critical software development? In critical systems, it's particularly important to perform a rigorous risk analysis, identify hazards and faults, and then specify mitigating requirements, which are carefully tracked throughout the system's development. How should the requirements process be conducted so that the end result is a carefully constructed assurance or safety case that demonstrates a system is safe for use? What requirements techniques are effective in these kinds of environments?

Although the IT environment has changed over the years, the importance of eliciting and understanding requirements is timeless. The end result may be represented in different ways—as traditional "shall" statements, use cases, sketches, user stories, acceptance tests, formal logic, goal models, or state charts. However, as long as we continue to build software systems and care whether those systems meet the needs of our customers and other stakeholders, then we must continue to emphasize the importance of understanding, analyzing, and specifying requirements.

In taking over this column from Neil, I realize that I have some metaphorically large shoes to fill. Neil brought us eight years of wisdom, debate, and sometimes hilarity in the form of Colin Codephirst. I'm not even going to try to fill those shoes. Instead, I hope we can go on a new journey that winds its way through new challenges that emerge along the way. I hope to discuss current issues related to requirements and to take on some of the most controversial issues head on.

We must continue our quest to learn better ways to work with stakeholders to discover requirements, embrace change, deliver safe systems, engage distributed stakeholders, and support innovation and change. Although the challenges are endless, the benefits are immense. Requirements are very much alive and kicking!

If you have ideas for requirements-related topics that you would like to see discussed, please email me at jhuang@cs.depaul.edu.

**JANE CLELAND-HUANG** is an associate professor at DePaul University. Contact her at jhuang@cs.depaul.edu.

ON COMPUTING

Editor: **Grady Booch**
IBM, grady@computingthehumanexperience.com

AUDIO

# In Defense of Boring

Grady Booch

**THE PURPOSE** of good software is to make the complex appear simple.

Complexity is the key factor in the cost of software and the time it takes to develop and evolve it. If you reduce Barry Boehm's software economics model down to its essence, you'll see that this cost/time is a function of the complexity of the system, raised to the power of process times the team, times the tools (and weighted in that order, from the most to the least significant). Having the right tools helps—having the right team helps far more—but whatever you can do to control complexity has the most significant impact on a system's development life cycle. Furthermore, a good process will dampen complexity, while a bad process will amplify it.

There's also a subtle yet important interaction between an organization's process and its team. The very best teams will embody an emergent process that's perfectly tuned to its culture, its domain, and its history. This is the nature of all highly effective teams, wherein the process becomes a part of the atmosphere. However, to paraphrase Garrison Keillor's description of Lake Woebegone, every organization likes to believe that theirs is a place where all their developers are strong, all their code is good looking, and all their system metrics are above average. Nonetheless, on average, the average developer is average. This means that this process/team relationship is far more complex: the stronger the team (and the greater the risk to or value of the system), the less that a high ceremony process is needed. The better the team (and the less risk or value present), there can and there must be a reduction in ceremony.

As it turns out, all of this is very hard to do.

## The Dynamics of Complexity

First, as Fred Brooks has told us time and again (and as we need to be reminded time and again), there's an essential complexity to software, a complexity that's inescapable and irreducible. Build yourself a natural language question/answer system, manage the textual and visual brain droppings of about a billion users, craft a vehicle that can semiautonomously explore an alien planet: these are all things that multiple people spend multiple careers trying to get right.

> On average, the average developer is average.

Second, however, there is self-imposed, accidental complexity. Stick your head inside the workings of any 1 to 10 million SLOC system, and most of the time you will see a muddle. Yes, there will be obvious lines of demarcation, faults where you can observe the impact of some major technical or business tectonic shift, and reoccurring fossils in the software's geological levels, laid down by different individuals with different styles over different times. The most significant design decisions are probably visible, evident in the major edifices and reflected in the dark corners of the system. Nonetheless, I've yet to see any ultra-large software-intensive system without some vestigial organs and strange irregularities. This is the very nature of how large systems evolve, be they natural, organic, or human-made.

To that end, the best we can do is simply strive to manage complexity. We can neither reduce nor eliminate a system's intrinsic complexity. From a system's engineering perspective, this is where we apply all the tricks of our trade to devise crisp abstractions, a good separation of concerns, and a balanced distribution of responsibilities. A discipline of steady incremental and iterative executable releases helps to steer a project in the right direction, which is not necessarily the direction first envisioned. A discipline of patterns serves to establish the system's texture and attends to crosscutting concerns. A discipline of refactoring is hence the result of combining the best practices of a rhythm of releases with the motifs of textures. Refactoring helps to take off the sharp, unnecessary edges of a brittle system. When done right, the result is positively, beautifully, breathtakingly boring. As it should be.



0740-7459/13/$31.00 © 2013 IEEE

## Smooth Edges

These concepts apply not only to the inside of a software-intensive system but also to its outside. When used as a part of a system of systems, the edges of any subsystem must play well with others, especially with others that didn't even exist at the time you built your system. If a subsystem offers up APIs or services that are awkward to use, too fine-grained, too big, or just plain irregular, then you have a problem. That's not boring, because you'll find you have to force a fit by writing some one-off code that hides the evils of the existing interface, bridges the gaps, and sometimes routes around it, either by jumping across levels of abstraction or replacing some functionality entirely. When the edges of a subsystem are well designed, they are approachable and understandable, they snap together easily with other edges, and their behavior is predictable. Hence, they are boring.

On one hand, we seek to build software-intensive systems that are innovative, elegant, and supremely useful. On the other, computing technology as a thing unto itself is not the place of enduring value, and therefore as computing fills the spaces of our world, it becomes boring. And, that's a very good and desirable thing.

This is the perspective of boredom as seen from inside a software-intensive system looking out. Looking at such a system from the outside in is an entirely different matter. Let us then look at software through the lens of the human experience.

## Technological Babysitting

Recently, I was in Silicon Valley, where I did a little shopping. I'm a people-watcher, and a charming young boy, perhaps three or four years old, caught my eye. He was with his father, and the two were apparently waiting for the boy's mother, who was trying on clothes. Time and again, the young boy tried to engage his father's attention, to no avail. Completely frustrated with the interruption of wherever the father's thoughts were taking him, the dad whipped out a smartphone, put on a movie, and shoved it under his son's face. The father continued looking out into space, while the child, slack-jawed, focused on the movie, his face bathed in the usual smartphone glow (a phenomenon I call receiving an iTan).

In the father's defense, he might have been having a Really Bad Day, but I don't think so. Rather, the father was medicating his son with an iPhone. In so doing, using Sherry Turkle's terminology, the father and son could now be alone together. This is a scene I see play out all the time. I'm no longer surprised when, walking along the beach, I see a whale breaching, only to look back at the shore and see a family, heads down in their smart devices, oblivious to the world beyond their screens. I suppose, using a title from the Grant Naylor book, they found their computing experience to be *Better Than Life*.

I am an expert in computing, not in children (although my wife is, as a child and family therapist who was in private practice), and I have no children of my own (although we have been godparents to about a dozen kids and have also brought a single mother and her child into our household for a few years). That said, I recognize when technology is being used as a substitute for reality, and what I was witnessing was one such case. From my perspective, a child needs time to dream, and while tablets and such are useful in moderation, they are never a substitute for human interaction, especially when one is learning how to grow up.

Turkle's *Alone Together* and Carr's *The Shallows* offer some evidence of the effect that technology has upon us when we immerse ourselves inside it, at the expense of being fully present in the world. There is work to be done to deeply, scientifically understand the implications of computing, but nonetheless…look! Squirrel!!!

Sorry, I was distracted there for a moment.

But that's the point. We don't yet know fully the implications of intimate computing on the individual, nor likely will we for a generation or so. While I'm confident that the human spirit will adapt, I'm also certain that all of us—especially children—need some boredom in our life. The intentional use of computing is a good thing, even if that means intentionally not using that technology from time to time, as a sort of digital sabbatical.

As such, we need more boring software, software that's so fundamentally boring that it disappears. If you must have a tablet in a child's face, then devise a killer app that would engage the child and the people in the immediate vicinity in such a way that they're required to interact with one another. Perhaps this might be an augmented reality app for a child's game of I Spy, or counting or spelling games that are contextual to the world around the child. You know, stuff that is part of the boring real world.

Now that's the kind of boring software we need much more of.

**GRADY BOOCH** is an IBM Fellow and one of the 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.

See www.computer.org/software-multimedia for multimedia content related to this article.

Editor: **Diomidis Spinellis**
Athens University of Economics and Business, dds@aueb.gr



# Systems Software

Diomidis Spinellis

**SYSTEMS SOFTWARE IS** the low-level infrastructure that applications run on: the operating systems, language runtimes, libraries, databases, application servers, and many other components that churn our bits 24/7. It's the mother of all code.

In contrast to application software, which is constructed to meet specific use cases and business objectives, systems software should be able to serve correctly any reasonable workload. Consequently, it must be extremely reliable and efficient. When it works like that, it's a mighty tool that lets applications concentrate on meeting their users' needs. When it doesn't, the failures are often spectacular. Let's see how we go about creating such software.

## Writing

As an applications programmer, the first rule to consider when writing a vitally required piece of systems software is "don't." To paraphrase the unfortunate 1843 remark of the US Patent Office Commissioner Henry Ellsworth, most of the systems software that's required has already been written. So, discuss your needs with colleagues and mentors, aiming to pin down the existing component that will fit your needs. The component could be a message queue manager, a data store, an embedded real-time operating system, an application server, a service bus, a distributed cache—the list is endless. The challenge is often simply to pin down the term for the widget you're looking for.

Once you start writing, focus on the data structures and algorithms you'll adopt. You're building infrastructure and therefore you can make few, if any, assumptions about your workload. Use reasonably efficient algorithms to avoid surprising your clients with resource hoarding and unwelcomed bottlenecks. If a design can let you serve requests in nearly constant time, your clients will expect you to implement such a behavior. In such a case, it's unreasonable for the time you take to service a request to increase with the number of elements you've served.

The data structures you choose should also gracefully accommodate the workload without placing any artificial limits on it. That's not as easy as it sounds: you're most likely to program in C and lack access to the sophisticated container libraries available in higher-level application frameworks. Use dynamically expanding buffers, memory pools, or linked lists to handle arbitrary amounts of data.

Error-checking is a related problem. The C language doesn't offer exceptions, which you're obliged to catch, so functions return error codes, which you should check scrupulously. If you fail to do that, your code might lose data or crash and burn. As an example, at the time of writing, the GNU *time* and Windows *route* commands will silently lose their output if redirected to a full disk. Recovery from most errors is difficult, but your code should handle those well-documented cases in which the proper response to an error or short result is to retry the operation.

Then come the nitty-gritty details that affect efficiency. Be a good citizen by having your code block when it has nothing to do. Looking around for work in a polling loop wastes precious resources. Instead, determine who might have something for your process, and use the POSIX **select** and **poll** calls to wait until such work becomes available. Design your system's communication patterns using this pattern, so that a lack of work will idle all its processes.

Modern memory is at least an order of magnitude slower than the CPU, so stay away from it. Avoid repeatedly processing data in memory. Cache intermediate results, and try to obtain all the data you need from a memory location with a single access. Where possible, sidestep memory copying. For instance, the POSIX **mmap** system call

Post your comments online by visiting the column's blog:
www.spinellis.gr/tools



0740-7459/13/$31.00 © 2013 IEEE

allows you to transfer data between files and your application without having the operating system copy it to its buffers, while the **readv** and **writev** calls allow you to combine data from multiple buffers into a single I/O request. These two things save you the cost of copying data into a single buffer or that of multiple system calls (another fine way to waste CPU time). Thus, you exploit the goodies that modern hardware and operating systems offer you to make your code more efficient.

Although intricate dependencies on lower layers are fair game for systems software, horizontal ones aren't. Systems software should be free-standing as much as possible; your client software is likely struggling to balance multiple conflicting requirements. Arriving at the party with your own long list of uninvited guests isn't polite. Therefore, eschew dependencies on obscure libraries, tricky-to-install components, and large frameworks that might not be available by default. Make your software play well with package management systems, allowing its painless installation and updating.

In contrast to application software, where the lack of a thick manual can be a virtue, systems software should be accurately and comprehensively documented. The documentation is the contract you draw with clients; strive to write precisely how your tool will behave, how it can be configured, and how it can fail.

## Testing

Testing systems software can be tricky because it often contains complex algorithms that are subjected to grueling stress levels. Instead of the leisurely input that many application programs receive from the keyboard and mouse over a working day, systems software typically has to deal with machine-generated input arriving through a fire hose over a period of months. Worse, input coming from the outside world can even be maliciously crafted for diverse nefarious purposes.

You can accelerate stress testing your software by configuring your testing environment to exercise its edge cases. For instance, if your software's dynamically grown buffers are 64-Kbytes, test its behavior when they're just 16 bytes. If you expect to service 10 clients, check what happens when you service 500. On top of that, write a test harness to feed your software with a huge number of test requests of all shapes and sizes.

> Accelerate stress testing your software, by configuring the testing environment to exercise its edge cases.

You can go a step further by actively downgrading the environment in which your software runs. We saw the importance of error checking; you can verify how you handle errors by introducing faults behind your software's back using tools like the *libfiou* library (http://blitiri.com.ar/p/libfiu/) or Chaos Monkey.

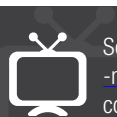
## Debugging

Debugging systems software when rare, nondeterministic errors crop up is just as difficult as testing it. These aptly-named *heisenbugs* will appear only when input, timing conditions, and the software's internal state line up. Reproducing such errors can take days of stress testing. Good luck tracing them by single-stepping through a debugger. Worse, a decent debugger might not even be available, either because your code runs on a resource-constrained system or because your code is part of the infrastructure in which the debugger would normally run,

The solution to this problem involves instrumenting your software with copious amounts of configurable logging. This will present the software's internal state, data structures, and how one step leads to another. Hopefully, you can reproduce the error with logging turned on and then locate its cause by trawling through the detailed log records. I recently had a case where just 3 out of 7 million requests were mishandled. I was fortunate, for I could find a rare misalignment issue in the logs. Some colleagues were less lucky and had to hook a logical analyzer in the computer's guts to locate an operating system error.

So, with mean and lean code, paranoid testing, and comprehensive logging, you'll write the systems software that your applications deserve.

**DIOMIDIS SPINELLIS** is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of the books *Code Reading* and *Code Quality: The Open Source Perspective* (Addison-Wesley, 2003, 2006). Contact him at dds@aueb.gr.



See www.computer.org/software-multimedia for multimedia content related to this article.

SOFTWARE TECHNOLOGY

Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

# Software Agents in Industrial Automation Systems

Stephan Pech

In recent years, agent-oriented software engineering has evolved into a powerful software engineering paradigm. Agents enable abstractions not only from the problem domain but also toward dynamic solutions that evolve in real time, depending on environmental stimuli to the software system. Agents complement the structured development of reliable industrial automation software systems by providing the necessary flexibility and adaptability. Author Stephan Pech looks at the engineering of agent-oriented systems and provides practical guidance to get started. 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*

**HIGHLY DEVELOPED INDUSTRIAL** automation (IA) systems are the result of long-term engineering experience and the core elements of several different fields of industry. By integrating technologies and application domains, IA software systems tend to become collections of distributed service-provider and service-consumer elements that are interlinked during runtime by dynamically defined workflows.[1] This leads to increasing complexity in the overall software system and its associated applications, requiring some sort of dynamic adaption to changing requirements and interfaces. Because agent-oriented software engineering is a mature software engineering methodology, it can address this need, complementing the structured development of reliable IA software systems by providing the necessary flexibility and adaptability.[2]

Compared to ordinary enterprise environments, IA environments have a different use context. Their inclusion of various roles, such as engineers, process personnel, and managers, along with their different views on systems, is challenging. Furthermore, the diverse IA system landscape breeds multiple individual solutions that are difficult to scale and maintain. To tackle these challenges, researchers have developed miscellaneous middleware and knowledge-intensive expert systems to automate human processes and provide technology-independent access to information. However, such solutions still lack adequate user-oriented assistance.



0740-7459/13/$31.00 © 2013 IEEE

TABLE 1

Examples of agent-based concepts in industry applications.

| Use context | Current challenges | Supporting tools | Emerging benefits using software agents |
|---|---|---|---|
| Agent-based engineering in plant automation[4] | Manual human processes to handle the engineering of an industrial automation (IA) plant for technical components, functionalities, and information | Computer-aided design and engineering tools | Active technical support for IA plant engineering processes; autonomous investigation of evolving technical correlations within plants |
| Agent-based dynamic scheduling for flexible manufacturing systems[5] | Dedicated resources and statically defined jobs on shop floor resulting in fixed workflows; inability to react to manufacturing disturbances | Computerized, numerically controlled software systems; automatic guided transport vehicles | Flexible and dynamic scheduling of available shop floor resources to requested jobs; production flow scheduled to maximize the utilization of these resources |
| Agent-based monitoring systems in process automation[6] | Requirements for increasing production effectiveness resulting in integrated production systems; only a few process operators monitoring a lot of changing values in process operations | Device maintenance, optimization, diagnostic, reporting, and monitoring tools; integrated process automation and control systems | Process personnel supported to cope with an increasing amount of responsibilities; software agents autonomously fulfilling knowledge handling and data processing tasks |
| Agent-based information retrieval for IA systems[7] | Heterogeneity of the data infrastructure itself and the diverse system landscape in IA systems; individual solutions that are challenging to use and maintain | Business intelligence systems, specific industrial software systems, knowledge-based systems | The information user's workflow gets structured, leading to better search results and higher efficiency in the information retrieval workflow |

Software agents can help close this gap: their autonomy, proactivity, goal orientation, interaction, and mobility provide much needed flexibility at runtime.[3] Moreover, depending on the system's design goals, these characteristics don't have to appear simultaneously. The basic concepts of agent orientation make up the next steps—specifically, toward the development of flexible and adaptable agent-based software systems. Conceptual considerations of how to design a specific software system are shifting from the solution space (specific technologies) to the problem space (specific working routines), and the implemented software modules are becoming more and more "intelligent" because they decide their own control flow.[3] These characteristics make software agents uniquely qualified for use in dynamic environments such as IA and are the major benefits compared to conventional system design methodologies. Having control flows and decision processes distributed to autonomous software agents leads to a decoupling of system elements and to reduced centralized complexity.

## Exemplary Agent-Based Application Scenarios

The software agent community has developed several agent-based applications in the IA domain. Table 1 gives a brief overview of some specific examples.[4–7]

## Application Scenario in Plant Automation

Typical industrial plants in the process industry consist of many process and information systems. To holistically handle the tasks related to process optimization, the information aggregated in the enterprise resource planning system isn't sufficient alone. Additional information from other systems must be manually collated, which can be an elaborate task for process personnel. Research in this area shows that we can separate the entire query process into partial query steps that software agents can handle independently and thus provide partial results.[7] These query steps constitute the atomic building blocks of more sophisticated queries. We can also reuse them to query a variety of other data sources. But due to the tremendous amount of possible queries and the resulting combinational alternatives, it isn't feasible to foresee all possible options in an information system's development phase. We therefore must combine relevant partial query steps for runtime, which indicates the use of software agents.

At the architectural level, design decisions for a multiagent system's structure focus on the decomposition of system functionalities and the connection to heterogeneous data sources. This is based on the comprehensive representation of system elements through software agents and ontologies. The outcome is a layered architecture—as in Figure 1—that contains three functional and cooperative layers as well as two different types of ontologies. In general, ontologies provide the preconditions

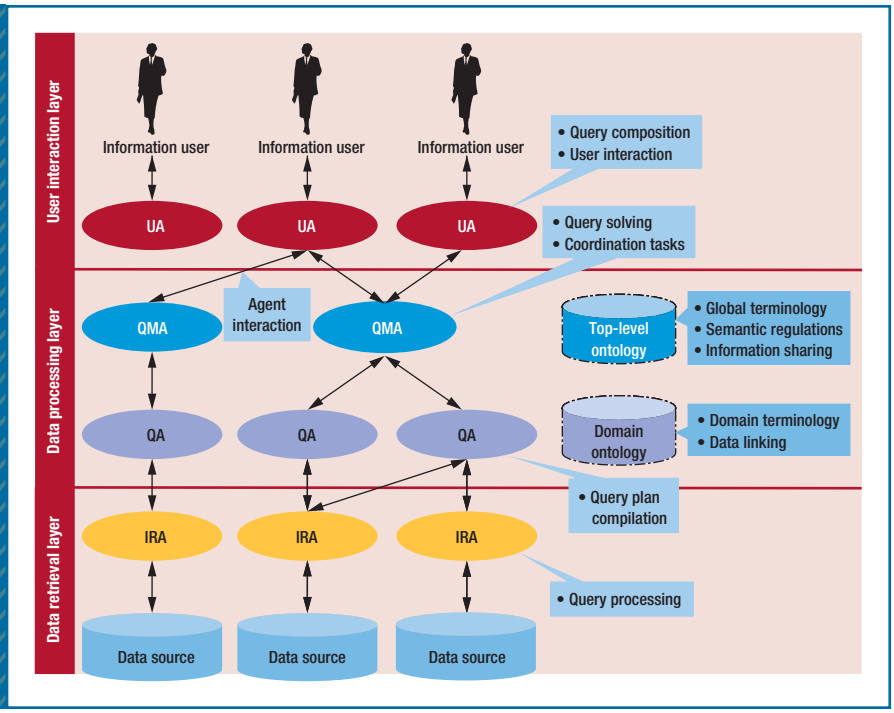



**FIGURE 1.** Information retrieval concept using software agents. This figure depicts four agent types: user agents (UAs) assist information users in composing queries and provide user interaction, and query management agents (QMAs) work on solving queries by cooperating with UAs and query agents (QAs) within the network. QAs help compile query plans, which information retrieval agents (IRAs) use to query data sources.

for establishing cross-domain semantic queries. Domain ontologies represent domain-specific knowledge, whereas global domains provide a global view of information.

Within a research project, we defined and evaluated four different agent roles, each of which is assigned to one of the following agent types: user agents (UAs) assist users in composing queries and provide user interaction, and query management agents (QMAs) work on solving queries by cooperating with UAs and query agents (QAs) within the network. QAs help compile query plans, which information retrieval agents (IRAs) use to query data sources. QAs then cooperate with IRAs to abstract data storage out of data management and can therefore process queries on any type of data source.

The advantage of the conceptual separation of IRAs and the data sources' information and communication models is the flexible extensibility concerning new data sources at runtime. Semantic Web technologies frequently apply ontology-based information integration concepts to explicitly describe data source semantics. In our concept of agent-based information retrieval, ontologies are regarded as classes, entities, properties, and relationships that describe a standardized terminology exchanged between agents. The cross-domain connection of terms within the different ontologies fosters a consistent view of the different data sources and guarantees consistent data management. With the automated compilation of queries, workflows conducted by software agents can reduce the number of errors otherwise introduced by human information users.

## Applicability of the Agent-Based Concept

A major criterion for assessing the applicability of this agent-based concept is its seamless integration in an information-driven industrial environment. Compared to development from scratch, there's less effort to adapt to heterogeneous interfaces, communication protocols, and proprietary software systems. With regard to raising efficiency, flexibility, and adaptability, the agent-based concept contributes positively in different ways.

Table 2 shows a selection of several standards, methodologies, and agent platforms that support the implementation of agent-based systems.[8,9] From a practitioner's view, the following issues are of particular importance:

- The quality of search results is important in determining how the user's information needs are covered. For manually executed queries, the range and the accuracy of search results depends primarily on user-specific knowledge about the application domain. With support from an agent-based information system, the user's workflow is both structured and repeatable.
- Flexibility during an IA's operating phase—especially for dynamic information retrieval instead of statically planned queries—is most apparent when a modification of the information environment structurally changes the data models. By using ontologies, the efforts for necessary changes can be reduced to a minimum—the adaptation of the ontologies itself. Otherwise—following conventional approaches—the whole system architecture has to be adapted to the modified data models.

TABLE 2

**Standards, methodologies, and platforms that support the implementation of agent-based concepts.**

| Name | Type | Description | Reference |
|---|---|---|---|
| Foundation for Intelligent Physical Agents (FIPA) | Standard | FIPA is the IEEE Computer Society standards organization for agents and multiagent systems; it promotes agent-based technology and the interoperability of its standards with other technologies. | www.fipa.org |
| Agent Platform Special Interest Group (Agent PSIG) | Standard | Agent PSIG works with the Object Management Group platform to promote standard agent modeling languages, specifications, and techniques in the area of agent technology. | http://agent.omg.org |
| Multiagent Systems Engineering (MaSE) | Methodology | MaSE is a general methodology for developing heterogeneous multiagent systems using graphically based models in UML to describe the system elements and the internal agent design. It's strongly oriented on the Unified Process and supports capturing agent goals, defining roles and tasks, and describing interaction possibilities. | "An Overview of the Multiagent Systems Engineering Methodology"[8] |
| Process for Agent Societies Specification and Implementation (PASSI) | Methodology | PASSI is a UML-based methodology for the specification and implementation of multiagent systems. It integrates design models and concepts from both object- and agent-oriented software engineering. | "A CASE Tool Supported Methodology for the Design of Multi-agent Systems"[9] |
| Whitestein Living Systems Technology Suite (LS/TS) | Platform | LS/TS is an industry-grade, Java-based development and runtime platform for the development and execution of agent-oriented software systems. It supports the main concepts of autonomic computing and comprises a set of development tools. The LS/TS API supports OWL and therefore allows semantic communication. | www.whitestein.com |
| Java Agent Development Framework (JADE) | Platform | JADE is an open source software framework fully implemented in Java. It simplifies the implementation and operation of multiagent systems. The agent platform can be distributed across machines (including mobile devices), which might or might not run the same OS. | http://jade.tilab.com |
| Cognitive Agent Architecture (Cougaar) | Platform | Cougaar is an open source, Java-based architecture for the construction of large-scale distributed agent-based applications with minimal consideration for the underlying architecture and infrastructure. Cougaar can be easily integrated with libraries and other technology platforms and supports the development of distributed real-time, peer-to-peer applications. | www.cougaar.org |

- System independency of the agent-based information retrieval concept is one of the features that were followed with the realization. This deals with several different facets: independence from the control of the technical process, the technically independent access to data sources, and the flexible building of the query structure and semantics.

In spite of the agent-based concept's advantages, it has some limitations as well. For instance, as multiagent systems are a community of autonomous software entities, there's no guarantee that all of the assigned queries are answered in time. One reason is the nondeterministic behavior of the agent collaboration itself. But because the information retrieval process is independent from the control functionality of the IA, this doesn't matter in most cases.

The main benefit of using software agents in IA is the combined application of agent-oriented software engineering with other fields, such as semantic technologies. Software agents provide flexibility, which is often the key requirement for creating software system architectures that can evolve during runtime.

## References

1. A.P. Kalogeras et al., "Vertical Integration of Enterprise Industrial Systems Utilizing Web Services," *IEEE Trans. Industrial Informatics*, vol. 2, no. 2, 2006, pp. 120–128.

# SOFTWARE TECHNOLOGY


2. H. Mubarak, "Developing Flexible Software Using Agent-Oriented Software Engineering," *IEEE Software*, vol. 25, no. 5, 2008, pp. 12–15.
3. M. Wooldridge, *An Introduction to Multi-Agent Systems*, 2nd ed., Wiley, 2009.
4. T. Wagner, "An Agent-Oriented Approach to Industrial Automation Systems," *Proc. 3rd Int'l Symp. Multi-agent Systems, Large Complex Systems, and E-Businesses*, Springer, 2003, pp. 314–328.
5. I. Badr, "An Agent-Based Scheduling Framework for Flexible Manufacturing Systems," *Proc. Int'l Conf. Industrial Eng.*, IEEE, 2008, pp. 465–470.
6. T. Pirttioja, "Applying Agent Technology to Constructing Flexible Monitoring Systems in Process Automation," PhD dissertation, Faculty of Electronics, Communications and Automation, Helsinki University of Technology, 2008.
7. S. Pech and P. Göhner, "Multi-agent Information Retrieval in Heterogeneous Industrial Automation Environments," *Proc. 6th Int'l Workshop Agents and Data Mining Interaction* (ADMI 10), Springer, 2010, pp. 27–39.
8. M.F. Wood and S.A. DeLoach, "An Overview of the Multiagent Systems Engineering Methodology," *Proc. 1st Int'l Workshop Agent Oriented Software Eng.*, LNCS 1957, Springer, 2001, pp. 207–221.
9. M. Cossentino and C. Potts, "A CASE Tool Supported Methodology for the Design of Multi-agent Systems," *Proc. 2002 Int'l Conf. Software Eng. Research and Practice* (SERP 02), CSREA, 2002, pp. 295–306.



**STEPHAN PECH** is a former scientific staff member at the Institute of Industrial Automation and Software Engineering at the University of Stuttgart. He currently works as an automation engineer at BASF SE. His main research area is in manufacturing execution systems and applications. Contact him at stephan.pech@ias.uni-stuttgart.de.

FOCUS: GUEST EDITORS' INTRODUCTION

# Safety-Critical Software

**Xabier Larrucea**, Tecnalia

**Annie Combelles**, inspearit Group

**John Favaro**, Intecs SpA

**TODAY, WE LIVE IN A WORLD IN WHICH OUR SAFETY IS MORE AND MORE DEPENDENT ON SOFTWARE-INTENSIVE SYSTEMS.** This is the case for the aeronautic, automotive, medical, nuclear, and railway sectors, as well as many more. Organizations everywhere are struggling to find cost-effective methods to deal with the enormous increase in size and complexity of these systems, while simultaneously respecting the need to ensure their safety. Consequently, we're witnessing the ad hoc emergence of a renewed

0740-7459/13/$31.00 © 2013 IEEE

FOCUS: GUEST EDITORS' INTRODUCTION

## OPEN PLATFORM FOR EVOLUTIONARY CERTIFICATION OF SAFETY-CRITICAL SYSTEMS

Safety-critical software faces a costly aspect: the certification process. OPENCOSS, a large-scale collaborative project of the EU's Seventh Framework Program, focuses on the harmonization of safety assurance and certification management activities for the development of embedded systems in automotive, railway, and aerospace industries. The main goal is to reduce both the time and cost overheads inherent to the safety (re) certification of safety-critical systems, via facilitating the reuse of certification assets. The strategy is to focus on a compositional and evolutionary certification approach with the capability to reuse safety arguments, safety evidence, and contextual information about system components in a way that makes approvals for operation and certification more cost-effective, precise, and scalable.

OPENCOSS is defining a common certification language (CCL) by unifying the requirements and concepts of different industries and building a common approach to certification activities. Much of what is being done will have a transformative effect on the safety-critical software community if the take up really occurs. An industrial adoption program is being overseen by an advisory board with members from key organizations such as the European Railway Agency, Airbus, Eurocopter, NASA, and Renault. For more information, see the website: www.opencoss-project.eu.

discipline of safety-critical software systems development as a broad range of software engineering methods, tools, and frameworks are revisited from a safety-related perspective. A major goal of this special issue of *IEEE Software* is to take stock of these individual initiatives and try to see the bigger picture.

### Complexity Scales

As an example of important paradigms currently being revisited in a safety-related context, Thales recently announced the use of object-oriented technologies and agile software development methodologies to optimize its safety-critical systems development (www.erts2012.org/Site/0P2RUC89/7A-4.pdf). Likewise, NASA is exploring the study and application of agile development in its safety-critical systems (http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120013429_2012013093.pdf).

But it isn't just the popular, headline-grabbing software engineering techniques such as agile development that are being revisited in the safety-critical systems community. Understanding the effects of fundamental software engineering activities, including verification, validation, and certification, and choosing the right combination of them to yield systems that meet today's ambitious requirements in a cost-effective manner has become even more important. Consider the requirements engineering activity: How is it possible that, given the crucial importance of clear, concise, unambiguous requirements in critical software systems engineering, most tools in common use today still represent a requirement as a simple, unadorned string? The European Space Agency's recent study on next-generation requirements engineering, in which it used semantic wiki technology to nudge forward the state of the art, is just one example of the critical software community's growing impatience with traditional methods.

Several new and unprecedented factors are converging to change the nature of the challenges facing safety-critical systems development. One such factor is the unrelenting trend toward open, interconnected, networked systems (such as "the connected car" and the cloud), which has brought a security dimension with it, exacerbating the problem of ensuring safety in the presence of security requirements. Similarly, the model-driven architectures (such as AUTOSAR in the automotive industry) needed to handle these new large, networked systems are only now being equipped with mechanisms to handle safety-related aspects. The rise of these complex, critical systems has spawned several recent initiatives to promote reuse, both of the technical artifacts and the artifacts and procedures that certify their suitability for use in safety-related contexts. An example of such an initiative is OPENCOSS, an all-out, full-frontal assault on managing the problem of certification of software-intensive critical systems in multiple domains using model-based approaches and incremental techniques (see the sidebar).

### In This Issue

This special issue collects three papers from academia, two from industries, and two from academia with an industrial perspective. This balance provides a rather complete view of the current challenges faced in safety-critical industries despite the specific transportation industries represented. Model-based development and engineering is discussed in "Model-Based Development and Format Methods in the Railway

Industry," "Validating Software Reliability through Statistical Model Checking," and "Engineering Air Traffic Control Systems with MDE."

These articles address the challenges and failed expectations in applying these techniques, and highlight the missing link between academia and industry regarding this topic and the importance of tools to support implementation. We thank the authors of these three articles for providing real examples on how to deploy these techniques and believe that their expertise can be reused. "Testing of Formal Verification," based on DO-178C, is another easy-to-read article that digs into the attractiveness of formal methods technology for high-integrity systems. It's important to look at the trends in that domain, especially when two major aircraft manufacturers—Airbus and Dassault-Aviation—report the benefits realized.

This issue includes two other articles describing real cases as well. The article from Moog India Technology Center—another aircraft player—provides a collection of mistakes made and their root causes; the focus is on the numerous interactions the aircraft or flight system has with embedded systems that make certification of these systems so complex. "Strategic Traceability for Safety-Critical Projects" likewise targets the traceability issue, which is one of the key facets of certification; the authors provide a fairly detailed analysis of a few traceability issues and the way they were corrected.

Although embedded systems generally come to mind first when thinking of safety-critical software, another class of applications is equally important: the protection of the infrastructures that are critical to our everyday lives, such as transport systems. Although threats usually come from nature, such as hurricanes, earthquakes, and rainstorms, some threats are man-made, such as terrorism and sabotage. The software systems that protect these infrastructures must span international borders and bring a host of technical, legal, and cultural compatibility challenges with them that in many respects equal or surpass those faced in critical embedded systems. The last article of this issue, "SCEPYLT: An Information System on Explosive Control" provides insight into the issues faced by this type of critical system.

One unmistakable trend that emerges out of the articles in this special issue is a strong interest in applying model-driven engineering techniques to safety-critical systems development over the entire life cycle. The implementation community has been interested in model-based techniques for years, but the validation and certification community is slowly coming around to a perception that such approaches could provide the key to more efficient and effective management of their own tasks. We believe that this observable transition of a research technique into an industrial environment in which certification bodies are neither system nor software technology specialists is a significant step forward in safety-critical systems engineering and an interesting achievement to be reported in this magazine.

## ABOUT THE AUTHORS

**XABIER LARRUCEA** is a senior project leader at Tecnalia, Zamudio, Spain. He's also a part-time lecturer at the University of the Basque Country. His research interests are focused on safety-critical software systems, software quality assurance in multimodel environments, empirical software engineering, and technology road mapping. Larrucea has a PhD in software engineering from the University of the Basque Country. Contact him at xabier.larrucea@tecnalia.com.

**ANNIE COMBELLES** is the founder and CEO of inspearit, an advisory company in software and systems operating in France, Holland, Italy, and Asia. She's an associate editor of this magazine, a member of the Scientific Committee for Quality Engineering Laura Bassi Lab (QE LaB) in Austria, and a member of the executive committee of Les Journées de l'Entrepreneur. Contact her at annie.combelles@inspearit.com.

**JOHN FAVARO** is a senior consultant at Intecs SpA in Pisa, Italy, where he's also deputy director of research. His technical interests include efficient safety analysis of critical systems, safe and secure software reuse, and requirements engineering. Favaro has an MS in electrical engineering and computer science from the University of California, Berkeley. Contact him at john@favaro.net.

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

FOCUS: SAFETY-CRITICAL SOFTWARE

# Model-Based Development and Formal Methods in the Railway Industry

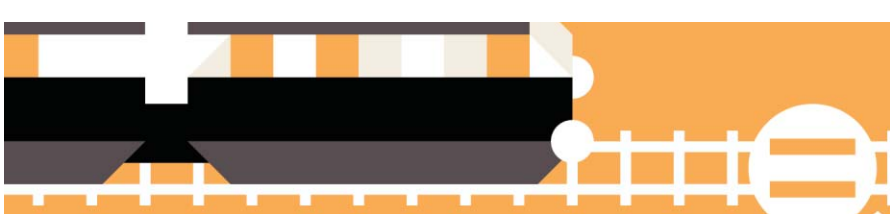
**Alessio Ferrari**, CNR-ISTI

**Alessandro Fantechi**, Università di Firenze

**Stefania Gnesi**, CNR-ISTI

**Gianluca Magnani**, General Electric Transportation Systems, Florence

*// The transition from a code-based process to a model-based one isn't easy, particularly for companies that operate in the safety-critical sector. A railway signaling manufacturer adopted general-purpose, model-based tools aided by formal methods to develop its products, facing challenges and learning lessons along the way. //*



**GENERAL ELECTRIC TRANSPORTATION SYSTEMS** (GETS), Florence, Italy, is a medium-sized branch of a global railway signaling manufacturer. Approximately 10 years ago, to address safety-critical industries' rising interest in formal methods,[1] the company started experimenting with formal modeling and verification. To this end, it contacted experts from the local university to support some initial experiments.

The team evaluated several formal tools, but the developers preferred a semiformal tool suite—namely, Simulink/Stateflow.[2] The Simulink language uses a block notation to define continuous-time dynamic systems; the Stateflow notation is based on David Harel's statecharts[3] and supports the modeling and animation of event-based, discrete-time applications. The main drivers for this choice included the large amount of packages available in the tool suite—packages that developers could use throughout the development process—and the widespread knowledge about the tools found within the company.

Initially, the developers used the models designed through Simulink/Stateflow solely for requirements elicitation, but in 2007, the company wanted to explore using such models for code generation as well. One year later, this technology became part of the development process, but changing the development paradigm from code-based to model-based required additional changes in the verification process. The company adopted model-based testing and abstract interpretation, as well as language restrictions to reduce the tool suite's semiformal semantics to a formal semantics.[4]

The new model-based approach sped up development and allowed the company to handle more complex systems. As projects grew in size, they required new technologies that could rigorously handle system requirements. The company selected SysML, a unified modeling language for system development, to address this issue. After three years with SysML, the company established a formal development approach that integrates SysML and Simulink/Stateflow.

In this article, we describe the



0740-7459/13/$31.00 © 2013 IEEE

challenges and lessons learned by the company throughout this 10-year experience.

## Challenges

Over the course of this experience, GETS faced several challenges that deserve some extra attention.

### Modeling Language Restriction

The code used in safety-critical systems must conform to specific safety standards, so companies typically use coding guidelines to avoid using improper constructs that might be harmful from a safety viewpoint. When a safety-critical company adopts modeling and autocoding, the generated code must conform to the same standards as handcrafted code. The adopted code generator—Simulink Coder—induces a tight relation between the generated code and any modeling language constructs. Hence, the identification of a safe subset of the modeling language enables the production of code that complies with the guidelines and that can be successfully integrated with the existing code.

GETS did this by first defining an internal set of modeling guidelines for Simulink/Stateflow—specifically, these guidelines were practical recommendations on language construct usage. The idea was that any model-generated C code following the guidelines would comply with the company's coding standard.

The company based the initial guidelines on a code analysis generated from a model previously designed for requirement elicitation. Because this preliminary set of guidelines had the limit of being derived from a specific model, it could lack generality, so in the projects that followed, GETS extended it with other recommendations borrowed from the automotive domain.[5]

To ease formal analysis, the company decided to complete the modeling style guidelines by restricting the Stateflow language to a semantically unambiguous set. To this end, it used studies that focused on translating a subset of Stateflow into the Lustre formal language.[6] The company's current models are therefore independent from the simulation engine, a choice that opened the door to formal verification.[4]

### Generated Code Correctness

Safety-critical norms, such as CENELEC EN 50128, the European standard for railway software, ask for a certified or proven-in-use translator. In the absence of such a tool, like in the case of available code generators for Simulink/Stateflow, a strategy must be defined to ensure that code behavior fully complies to model behavior, and that no additional improper functions are added during the code synthesis phase. The objective is to perform verification activities at the abstract model's level, minimizing or automating any operations on the code.

GETS adopted a model-based testing approach called translation validation,[7] completed by static analysis via abstract interpretation.[8] In translation validation, you execute test scenarios based on functional objectives at the model level. Then, you repeat the same tests on the generated code, checking that the model's outputs and the corresponding code are consistent. To ensure runtime error freedom, the company uses the Polyspace tool to perform abstract interpretation.[9] This final step verifies a program's correctness on an overapproximation of the range of program variables.

Certification authorities considered this overall approach suitable for bypassing the tool qualification required in current safety regulations. (Railway norms aren't as specific about tool qualification as, say, avionic norms are,[10] so companies in the railway sector must agree on possible strategies with certification authorities.)

### Multiple Formalisms

Safety-critical systems are large, complex platforms with several interacting units and architectural layers. To manage such complexity, their development is based on multiple levels of abstraction, a setup that requires different models with different granularities. Indeed, a model used for code generation is hardly usable for reasoning at the system design level. Simulink/Stateflow don't support a flexible hierarchical development approach, so system designers must adopt other modeling languages that can express the higher abstractions that the process inherently requires.

GETS addressed this issue by adopting SysML (www.sysml.org). After an initial experience with the TOPCASED tool for SysML (www.topcased.org), which at that time wasn't considered mature enough for industrial usage, the company adopted the Magic Draw platform (www.nomagic.com/products/magicdraw.html). Early GETS projects that used code generation didn't use SysML support—the need came when the systems the company produced started to radically increase in terms of complexity, for example, when the LOC exceeded 100,000.

> When a company adopts autocoding, the generated code must conform to the same standards of handcrafted code.

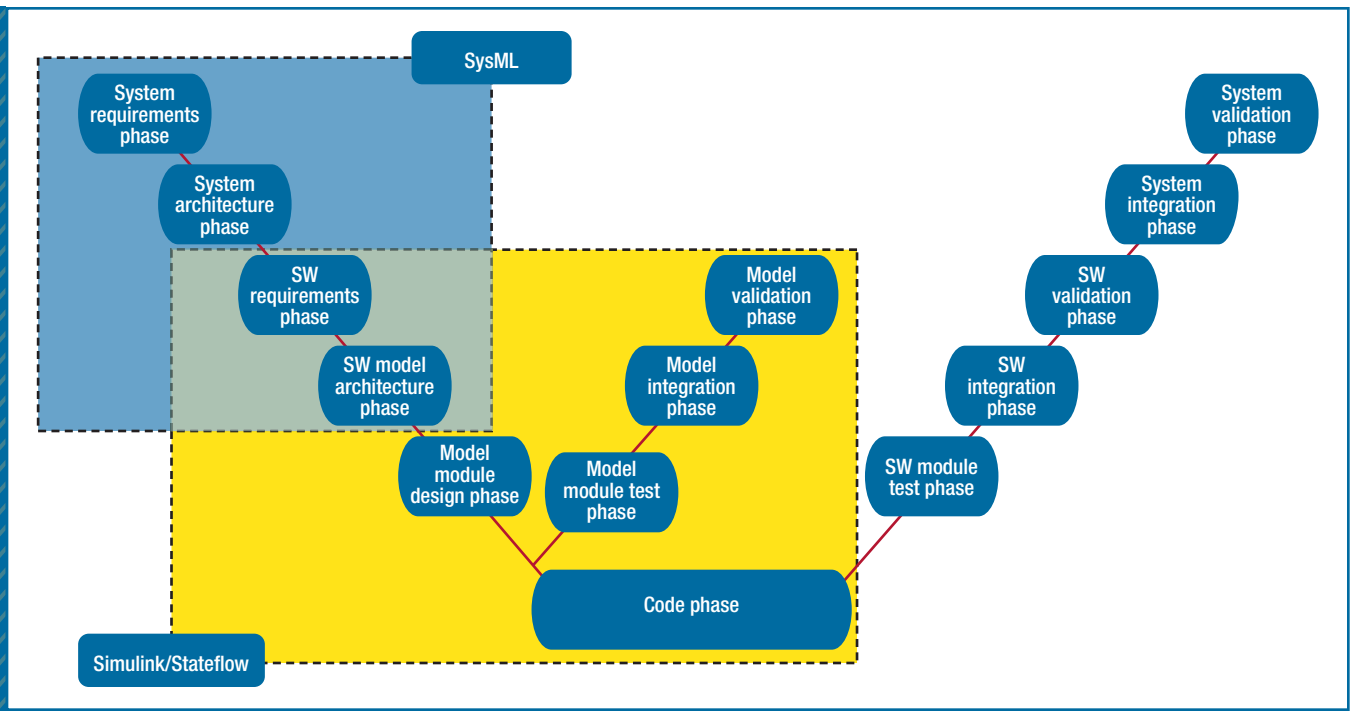



**FIGURE 1.** Overview of the model-based development process that GETS adopted. The left side of the V shape incudes the design activities, while the right side includes the verification activities. The small right branch on the left side of the V includes the verification activities performed at the model level.

SysML's current role is as follows. Immediately after requirement elicitation, requirements appear as unstructured Post-It notes on the requirements manager's whiteboard. The manager identifies high-level system requirements among this initial set and expresses them in the form of SysML requirement diagrams. These diagrams allow the requirements manager to specify hierarchical relationships and dependencies among single requirements, replacing the chaotic Post-It view with a structured graph-like model.

Next, the design team uses block diagrams to specify the interfaces of the system modules that are supposed to implement the requirements. The team adopts an approach based on decomposition, which allows specializations of each module into submodules to realize the actual implementation. Each module has some high-level requirements apportioned to it, and if needed, the design team can refine requirements into lower-level requirements when modules are specialized.

SysML models are structured as packages in a single root model, where each package corresponds to a phase of the V-based development process prescribed by the CENELEC norm for railway safety-critical systems. Therefore, a well-defined mapping between process phases and the diagrams are used in each phase.

SysML could also be used to define the actual implementation's behavior and to generate code. However, modeling and simulation at the behavioral level is much faster and more flexible with Simulink/Stateflow, and the generated code has higher quality. Therefore, SysML's role ends at the software architecture level.

## Process Integration

Companies develop products via processes, which define a framework made of tasks, artifacts, and people. The introduction of new technologies in an established process requires adjustments to the process structure, which has to maintain its coherence through these changes. This is particularly true in the case of safety-critical companies, whose products must be validated according to normative prescriptions. Hence, a sound process must be defined to integrate modeling and code generation within the existing framework.

As Figure 1 shows, GETS defined an enhanced V-based process that embeds two verification branches: one for the activities performed on the models, and the other for the tasks concerning source code and the system. In the figure, we highlight the parts that strictly concern software development (based

on Simulink/Stateflow modeling) and the parts related to system development (based on SysML modeling). The two process fragments overlap in the SW Requirements phase and in the SW Model Architecture phase. Indeed, software requirements are expressed in SysML, as well as in the software architecture. An equivalent architecture is expressed through Simulink in the form of interacting blocks, which are the model's functional modules—the components. We can manually trace SysML requirements to the Simulink model. The Model Module Design phase refines the Simulink blocks into Stateflow statecharts.

Note that this process is somehow adaptable to both manual coding and to autocoding. After completing the SysML modeling activities, you can decide to adopt either handcrafted code or Simulink/Stateflow modeling to develop the application. Indeed, in some applications—for example, firmware, systems with limited software, or platforms with strong dependencies from legacy code—the code generation technology is considered inconvenient, so developers use handcrafted code.

## Lessons Learned

Facing all the challenges in model-based development allowed GETS to learn some important lessons.

### Abstraction

Models let you work at a higher level of abstraction, and they can be manipulated more easily than code. The company experienced the actual relevance of this statement in the transition from code-based to model-based testing. The model-based testing approach it adopted allowed developers to define behavioral test scenarios at the component level without disrupting the model structure. This approach would have been impracticable on hand-crafted code. Indeed, with handcrafted code, it's common to perform tests on single functions, but it's much more difficult to identify the functions that contribute to a software component's behavior. With models, you build a system in terms of its components. Therefore, component identification and testing happens in a natural way.

GETS also learned that abstraction is a delicate concept that must be carefully handled, with the proper degree of abstraction clearly identified for an artifact to be useful. For example, in their initial experience with SysML, designers adopted natural language requirements throughout the process until they reached the lowest level of model detail. At that point, their content was basically equivalent to the Simulink/Stateflow models. Such requirements' level of abstraction had to be raised because they appeared to be redundant: any slight modification to the models would have implied a modification to the requirements.

### Expressiveness

Graphical models are closer to natural language requirements, yet they're also an unambiguous way of exchanging or passing artifacts among developers. The GETS team experienced this observation first hand when the project passed from its initial developer to another developer within a month and without much support.[4]

Up to that point, if someone in the company was a piece of software's father, he would have remained the one and only repository of knowledge about that software. This is a common problem in many small- and average-sized companies that negatively affects both the company itself, which has to rely on a single person to modify and reuse its core software, and the developer, who normally wants to extend his competencies beyond his initial fragments of code.

### Cohesion and Decoupling

Automatically generated software is composed of modules with higher internal cohesion and better decoupling with respect to manual coding. Interfaces among functionalities are based solely on data, and the control flow is simplified because there's no cross-calling among different modules. Decoupling and well-defined interfaces help ease the outsourcing of the modeling activity, which is a relevant aspect in the development of products that have to tackle time-to-market issues. The company acknowledged these advantages in the course of its development process.

> With strictly defined modeling guidelines, you can look at the generated code as if it were written by the same programmer.

Structured development gives developers greater control over components and ultimately leads to software with fewer bugs. At GETS, developers experienced this when the number of bugs found through verification dropped from 10 bugs to three per module after the company introduced a rigorous model-based process.[4]

### Uniformity

Generated code has a repetitive structure that facilitates automated verification activities. When you have strictly defined

modeling guidelines, you can look at the generated code as if it were consistently written by the same programmer. Thus, any code analysis task can be tailored on an artificial programmer's design habits. The abstract interpretation procedure adopted to reveal runtime errors worked only on the generated code because systematic analysis on handcrafted code was made harder by its variable structure and programming style.

> When passing from traditional code unit testing to model-based testing, verification costs fell approximately 70 percent.

SysML also guarantees uniformity at the process level. The use of a unified modeling language—and a single tool—in most of the development phases eases the development in all of the activities that involve interfaces among phases. Indeed, in a V-based process, a phase's output artifact is the input artifact for the following phase. The use of SysML makes this handover more rigorous.

### Traceability

Software modules are directly traceable to their corresponding blocks in the specification modeled with Simulink/Stateflow. Traceability is a relevant issue in the development of safety-critical systems because any error must be traced back to the process task or artifact defect that produced it. With the support of Simulink/Stateflow, GETS introduced a structured development approach that lets developers define navigable links between single code statements and requirements.

At the SysML level, traceability involves the links between requirements diagrams and related SysML diagrams. Simple drag-and-drop operations manually define the traceability links, automatically generating the traceability matrixes. In a traditional process, the developer manually edits traceability matrixes without any tool support, leading to maintainability issues.

At GETS, customer-issued change requests normally involve system-level requirements. The tool support available in the company's model-based approach allows changes to be traced from such requirements to the module-level requirements and the corresponding models. Therefore, both developers and the requirements managers have a complete view of a change request's impact. Contrast this with the traditional process, in which someone would have to inspect the traceability matrix and check for artifacts affected by the change request, an activity that can be rather time-consuming and error-prone (unless supported by automated tools).

Automatic traceability support among SysML and Simulink/Stateflow models is still an open issue because no tool currently implements such a feature.

### Documentation

For safety-critical systems, the official documentation associated with each process phase and artifact is as important as the actual system. Process certification is essentially based on an external authority's inspection of such documentation. It's therefore important to have documentation that's formal, expressive, and up to date on product status.

In the process that GETS currently uses, both SysML and Simulink/Stateflow models provide documentation for process artifacts. Simulink/Stateflow models with proper comments automatically generate software documentation, thus keeping documentation and software totally aligned. In addition, SysML diagrams are integrated into the manually edited documentation. Documents can be automatically generated from SysML as HTML pages, but certification authorities typically require paper-like documents focused on text, rather than navigable HTML documents. The main reason is that the certification authority normally enters at the end of the development process to validate compliance with standards and wants to analyze the process as a sequential history—with a paper trail—not as an interwoven graph of HTML pages.

The integration of SysML models into the documents does pose maintainability issues. Indeed, if the model changes, that change isn't automatically reflected in the documentation. However, the one-to-one correspondence between SysML packages and process phases—and the associated documents—eases the effort of manually updating the documentation. Furthermore, the traceability links between models in different packages help maintain the cross dependencies among documents. When a model changes, its package clearly identifies the document that must be modified as well. Anyone with access rights can follow the traceability links and retrieve the other models affected by the change. Such models belong to packages with associated documents, so the link among models indirectly creates a relationship among those documents, and the overall SysML model becomes a sort of navigable index for the process documentation.

### Verification Cost

The introduction of a new development process at GETS reduced some of the costs associated with verification activities, while ensuring greater confidence in product safety. When passing from traditional code unit testing based on structural coverage objectives to testing based on functional objectives aided by abstract interpretation, verification costs fell approximately 70 percent. The new approach was comparable to the previous one in terms of compliance with CENELEC EN 50128 requirements on verification, but the results were much more cost-effective.[9]

Although GETS has achieved consistent cost improvements, manual test definition still bottlenecks the process, requiring approximately 60 to 70 percent of the whole unit-level verification cost. Preliminary experiments with formal verification applied at the unit level demonstrate that this technology might considerably reduce verification costs for most requirements. Indeed, recent experiments with formal verification via Simulink Design Verifier have shown that verification cost can further drop by 50 to 66 percent.[4]

### Complexity

The main drawback of introducing automated code generation is the resulting software's size and overall complexity. Although these aspects don't complicate verification activities, they pose challenges from the performance viewpoint.

Real-time constraints for railway signaling systems aren't as demanding as they are for other kinds of embedded systems, since the typical required response times are in the range of hundreds of milliseconds. But they're still reactive systems that need to activate failure recovery procedures in a brief amount of time to reach a safe state, should a failure occur. Execution time influences reaction time. In the first experiments with automated code generation at GETS, this execution time took four times longer compared to the time required to execute the corresponding handcrafted code. So as not to abandon the advantages of autocoding, a hardware upgrade solved this timing discrepancy.

But to design new, more complex systems, this issue must be taken into account when defining the hardware architecture. The hardware designer has to consider that the code is both larger in size and less flexible in terms of source-level optimization (recall that compiler-level optimizations aren't recommended for safety-critical systems): when designing the platform, you must plan for a larger amount of memory if you want to use automatic code generation.

### Knowledge Transfer

From the GETS effort's outset, one research assistant from the university who operated within the company was fully focused on the technology to be introduced and an internal development team put that research into practice on real projects after the exploratory studies indicated success was possible.

The results obtained during this experience wouldn't have been possible via intermittent collaborations alone. On the other hand, to separate the research effort from the time-to-market issues, the research assistant's independence from the development team had to be preserved. Large companies can profit from dedicated internal research teams or even entire research divisions, but medium-sized companies often have to use the same personnel both to keep the organization on track for market needs and to take care of daily software development. We argue that the research management model adopted for GETS can be adapted to other medium-sized companies with comparable results.

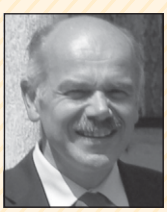
The people behind GETS were able to understand the benefits of a model-based process aided by formal methods thanks to the initial enthusiasm associated with automated code generation. Such technology showed its potential in a few months, and its adoption was relatively straightforward. After that, a butterfly effect occurred that brought forward the easy adoption of other techniques, such as model-based testing, abstract interpretation, and system modeling with SysML.

Formal verification isn't part of the GETS development process yet. But we've observed that formal verification with model checking is often the focus of a company's first experiments with something more formal, especially in the safety-critical systems domain. In most cases, those companies don't go much further than these initial experiments, notwithstanding the achieved evidence of lower verification costs. Indeed, the adoption of formal verification without intermediate steps isn't common: the difficulties associated with the steep learning curve required

When designing the platform, you must plan for a larger amount of memory if you want to use automatic code generation.

## ABOUT THE AUTHORS

**ALESSIO FERRARI** is a researcher at CNR-ISTI (Consiglio Nazionale delle Ricerche–Istituto di Scienza e Tecnologia dell'Informazione, Pisa). His research interests are primarily requirements engineering and formal modeling. Ferrari received a PhD in computer engineering from the University of Florence. Contact him at alessio.ferrari@isti.cnr.it.



**ALESSANDRO FANTECHI** is full professor at the University of Florence, where he's an active researcher in the areas of formal development and verification of safety-critical systems, with a particular emphasis on railway signaling systems. Contact him at fantechi@dsi.unifi.it.

**STEFANIA GNESI** is director of research at CNR-ISTI (Consiglio Nazionale delle Ricerche–Istituto di Scienza e Tecnologia dell'Informazione, Pisa) and head of its Formal Methods and Tool group. Her research interests include development of new logics for the formal specification and verification of concurrent systems and the application of model-checking techniques. Contact her at stefania.gnesi@isti.cnr.it.

**GIANLUCA MAGNANI** is a software engineer at General Electric Transportation Systems, Florence, signaling division. His technical interests concern the application of SysML modeling to the development of railway systems. Contact him at gianluca.magnani@ge.com.

by formal methods often tend to discourage industrial practitioners and managers, who need to see the evidence of productivity gains within a short time frame.

The experience we report here shows that it might be more effective to start with less formal tasks—automated code generation—and then later adopt more formal tasks, such as verification, when the company has matured into full awareness of the actual benefits of "being formal."

### References

1. J. Woodcock et al., "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, no. 4, pp. 19:1–19:36.
2. S. Bacherini et al., "A Story about Formal Methods Adoption by a Railway Signaling Manufacturer," *Proc. 14th Int'l Symp. Formal Methods*, LNCS 4085, Springer, 2006, pp. 179–189.
3. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science Computer Programming*, vol. 8, no. 3, 1987, pp. 231–274.
4. A. Ferrari et al., "The Metrô Rio Case Study," to be published in *Science Computer Programming*, 2013; doi: 10.1016/j.scico.2012.04.003.
5. A. Ferrari et al., "Modeling Guidelines for Code Generation in the Railway Signaling Context," *Proc. 1st NASA Formal Methods Symp.*, NASA, 2009, pp. 166–170.
6. N. Scaife et al., "Defining and Translating a 'Safe' Subset of Simulink/Stateflow into Lustre," *Proc. 4th ACM Int'l Conf. Embedded Software*, ACM, 2004, pp. 259–268.
7. M. Conrad, "Testing-Based Translation Validation of Generated Code in the Context of IEC 61508," *Formal Methods in System Design*, vol. 35, no. 3, 2009, pp. 340–389.
8. P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proc. 4th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, ACM, 1977, pp. 238–252.
9. A. Ferrari et al., "Adoption of Model-Based Testing and Abstract Interpretation by a Railway Signaling Manufacturer," *Int'l J. Embedded and Real-Time Communication Systems*, vol. 2, no. 2, 2011, pp. 42–61.
10. A.J. Kornecki and J. Zalewski, "Certification of Software for Real-Time Safety-Critical Systems: State of the Art," *Innovations in Systems and Software Eng.*, vol. 5, no. 2, 2009, pp. 149–161.

FOCUS: SAFETY-CRITICAL SOFTWARE

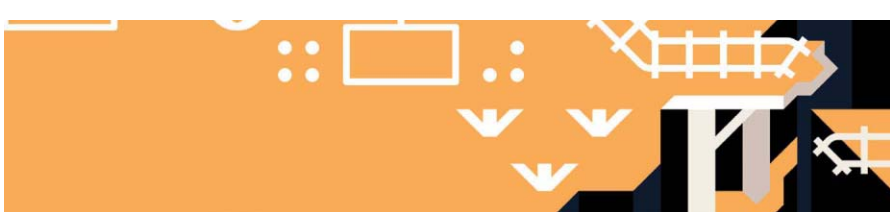
# Validating Software Reliability Early through Statistical Model Checking

**Youngjoo Kim**, S-Core

**Okjoo Choi**, **Moonzoo Kim**, **and Jongmoon Baik**, Korea Advanced Institute of Science and Technology

**Tai-Hyo Kim**, FormalWorks

*// A proposed framework employs statistical model checking to validate software reliability at an early stage. This can prevent the propagation of reliability allocation errors and design errors at later stages, thereby achieving safer, cheaper, and faster development of safety-critical systems. //*



**DURING THE PAST** few decades, the proportion of software in safety-critical systems has significantly increased. So, to ensure high-level safety, it's essential to improve software reliability. Consequently, it has become important to implement and acquire highly reliable software and to satisfy the safety requirements imposed by functional-safety standards, such as IEC 61508 and ISO 26262.[1–3] These standards define *safety integrity level* (SIL) and automobile SIL (ASIL) as measures of a system's quality or dependability.

To develop a highly reliable software-intensive system, developers allocate a reliability goal for a target system according to a target SIL or ASIL after hazard analysis and risk assessment.[4] Then, they allocate reliability goals to each software component early in the life cycle. Each component's reliability goal is usually validated through failure detection during software testing, which can result in high costs to correct defects.

We propose a framework to validate the reliability goals of safety-critical systems at an early stage by using *statistical model checking* (SMC) to obtain safety certification. SMC validates a target system's reliability by computing the probabilities that an executable model of a target system satisfies given functional-safety requirements. (For more information, see the "Statistical Model Checking" sidebar.)

## The Framework

Our framework (see Figure 1) extends IEEE Standard 1633, which covers software reliability practices. (For more information, see the "Software Reliability Engineering" sidebar.) It employs the following process.

### Specify the Functional-Safety Requirement

This step uses hazard analysis methods such as FTA (fault tree analysis), FMEA (failure mode and effects analysis), and Fracas (failure reporting, analysis, and corrective action system) to identify safety-related functions for each component $C_i$.[4] It then converts functional-safety requirements for those functions into bounded linear temporal logic (BLTL) requirements $req_{ij}$ of $C_i$.

### Allocate the Reliability Requirement

On the basis of the results of the "Specify the reliability requirement" step of IEEE Standard 1633, this step allocates a reliability goal $R_i$ to $C_i$.

### Validate the Reliability Requirement

This is the step we added that extends IEEE Standard 1633. Here, SMC

0740-7459/13/$31.00 © 2013 IEEE

## STATISTICAL MODEL CHECKING

Statistical model checking (SMC) uses randomly sampled simulation traces to compute the probabilities that a target model will satisfy given requirement properties.[1] Figure A gives an overview of SMC, which consists of a simulator, a bounded linear temporal logic (BLTL) model checker, and a statistical analyzer. It receives

- a stochastic target model $M$, which is an executable simulation model;
- a BLTL formula $\phi$, which formally represents a functional-safety requirement of the target system; and
- precision parameters with which to determine a calculated probability's accuracy.

The simulator executes $M$ and generates a sample execution trace $\sigma_i$. The model checker determines whether $\sigma_i$ satisfies $\phi$ and sends the result (success or failure) to the statistical analyzer. The statistical analyzer calculates the probability $p$ that $M$ satisfies $\phi$ by checking whether $\sigma_i$ satisfies $\phi$. The statistical analyzer then asks the simulator to generate $\sigma_i$ repeatedly until the number of successful results of $\sigma_i$ over the total number of $\sigma_i$ is distributed within a given precision boundary.

Unlike conventional formal verification techniques such as model checking, SMC doesn't analyze a target system's internal logic. So, it can validate complex safety-critical systems without the state explosion problems caused by those systems' complex hybrid (continuous dynamics plus discrete computation) characteristics.

### Reference

1. P. Zuliani, A. Platzer, and E.M. Clarke, "Bayesian Statistical Model Checking with Application to Stateflow/Simulink Verification," *Proc. 13th ACM Int'l Conf. Hybrid Systems: Computation and Control* (HSCC 10), ACM, 2010, pp. 243–252.

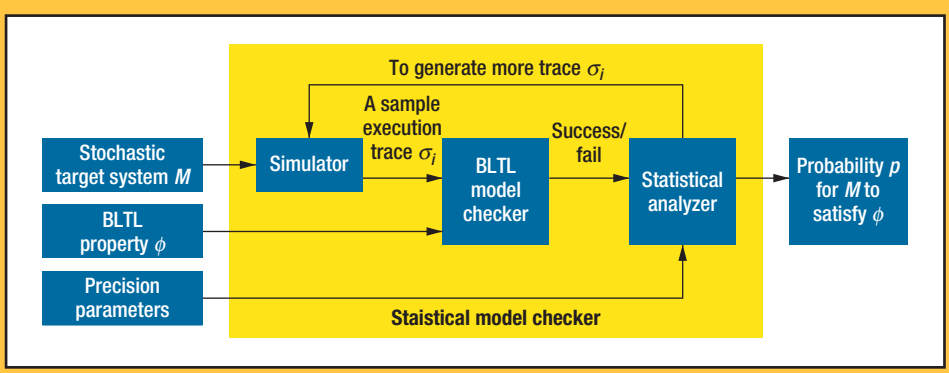



**FIGURE A.** Statistical model checking uses randomly sampled simulation traces to compute the probabilities that a target model satisfies given requirement properties.

generates random sample execution traces $\sigma_i$ repeatedly until the number of the traces is enough to calculate the probability that $C_i$ satisfies $req_{ij}$ (that is, $P(req_{ij})$). If not, SMC simulates $C_i$ again to generate more sample traces.

#### Validate the Reliability Goal

This step validates $R_i$ by comparing it with the calculated reliability $R_i'$, obtained on the basis of $P(req_{ij})$ and the corresponding weight values for $req_{ij}$.

#### Continue Validation or Reallocate

If $R_i'$ satisfies $R_i$ (that is, $R_i' \geq R_i$), validation continues for the next component $C_{i+1}$ regarding $R_{i+1}$. If the calculated reliabilities of all the components satisfy the allocated reliability goals, software reliability assessment continues.

If $R_i'$ doesn't satisfy $R_i$, this step reallocates all the components' reliability goals. If the reallocation continues to fail, this could indicate that the target component was designed incorrectly. If this is the case, after several trials of the reliability reallocation, the component should be redesigned to improve its reliability.

## Employing the Framework: A Case Study

The top part of Figure 2 diagrams a fault-tolerant fuel control system (FFCS),[5] a safety-critical component of an automobile's engine controller. The FFCS receives input from sensors for throttle angle, speed, exhaust gas oxygen (EGO), and manifold absolute pressure (MAP). It then generates a proper fuel injection rate and air-to-fuel ratio. It also detects sensor faults and shuts down the engine for safety if necessary. It has three components: a sensor failure detector and estimator (SFDE), an airflow calculator, and a fuel calculator.

The SFDE consists of a sensor failure detector and a sensor data estimator. The detector receives all the sensor

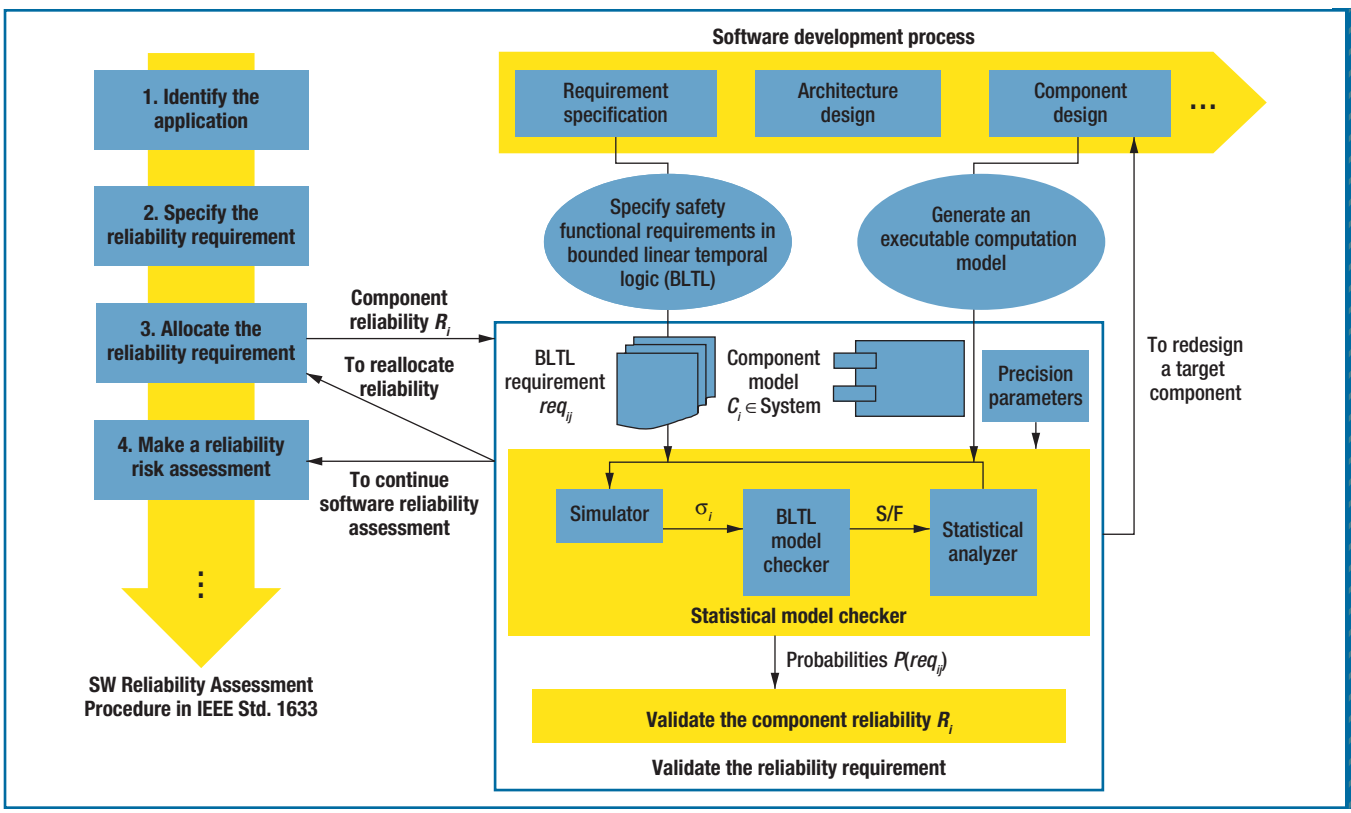



**FIGURE 1.** Our software reliability validation framework extends IEEE Standard 1633 by adding the step "Validate the reliability requirement" after the "Allocate the reliability requirement" step during software reliability assessment.

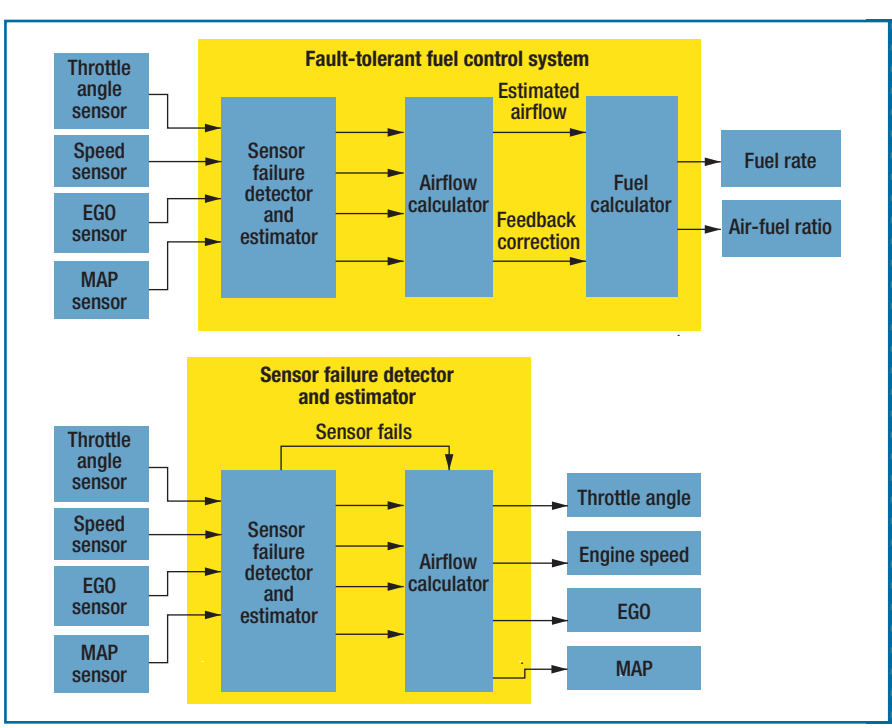



**FIGURE 2.** A fault-tolerant fuel control system (FFCS). Using input from sensors for throttle angle, speed, exhaust gas oxygen (EGO), and manifold absolute pressure (MAP), the FFCS generates a proper fuel injection rate and air-fuel ratio. It also detects sensor faults and shuts down an engine for safety if multiple sensor failures occur.

data and decides whether a sensor has failed. It delivers all the data to the estimator; if a sensor fails, it notifies the estimator of the failure. If multiple sensors fail, the detector shuts down the engine because the air-fuel ratio is uncontrollable.

The Simulink/Stateflow FFCS model's size and complexity in terms of the Halstead metrics[6] are as follows. The model has 65 operator blocks, 111 operands, 35 distinct operators, and 95

## SOFTWARE RELIABILITY ENGINEERING

Software reliability engineering (SRE) deals with predicting, estimating, and evaluating a target software system's reliability.[1] To apply statistical SRE techniques, developers collect reliability-related metrics throughout the development life cycle by testing the system on the basis of its operational profile.[2] So, SRE is essentially a quantitative study of software development regarding the given reliability goal. This activity repeats until it achieves the reliability goal. IEEE Standard 1633 provides guidelines with which to evaluate reliability by applying software reliability models.[3]

Recently, researchers have developed several software reliability prediction models to quantitatively manage software reliability at early development phases (the architecture and design phases), on the basis of system structure and the system usage profile.[4] However, these models are unrealistic owing to a lack of empirical data, especially for the early development phases. Also, they assume that each target component's reliability is known, which isn't true for real-world software components. On the other hand, our proposed software reliability validation framework—based on statistical model checking (see the main article and the other sidebar)—validates reliability at an early stage without such limitations.

### References


1. M.R. Lyu, "Software Reliability Engineering: A Roadmap," *Proc. Future of Software Eng. Conf.* (FOSE 07), IEEE CS, 2007, pp. 153–170.
2. J.D. Musa, "Operational Profiles in Software-Reliability Engineering," *IEEE Software*, vol. 10, no. 2, 1993, pp. 14–32.
3. *IEEE Std. 1633, Recommended Practice on Software Reliability*, IEEE CS, 2008.
4. L. Cheung et al., "Early Prediction of Software Component Reliability," *Proc. ACM/IEEE 30th Int'l Conf. Software Eng.* (ICSE 08), IEEE CS, 2008, pp. 111–120.


distinct operands. So, the calculated program volume $V$, representing the model's size, is 1,234, and the program difficulty $D$, representing the model's complexity, is 20.7. The automatically generated C code from the model has 222 functions in 8,266 SLOC. More information on the FFCS model is at www.mathworks.co.kr/products/simulink/examples.html?file=/products/demos/shipping/simulink/sldemo_fuelsys.html.

### FFCS Software Reliability Validation

An FFCS requires the ASIL D safety goal, and ASIL D in ISO 26262 requires a $1 - 10^{-3}$ to $1 - 10^{-9}$ reliability goal. So, we specify an FFCS's reliability goal as 0.9999. To determine the reliability goals for each component (the SFDE, airflow calculator, and fuel calculator) and the weight values for the functional-safety requirements, we consulted field experts from FormalWorks. This company produces software tools to test automobile software and conducts consulting for ISO 26262 certification. To obtain the reliability goals and the weight values more accurately, we can use Wideband Delphi estimation[7] with several iterations of experts' evaluations. We can also use Probe (proxy-based estimation),[8] another effective technique.

**Specifying the functional-safety requirement.** Through discussion with the FormalWorks experts who performed hazard analysis, we decided to specify functional-safety requirements for each of the component's output values. (For example, we specify four requirements for the SFDE, each corresponding to the output values for throttle angle, speed, EGO, and MAP.) So, we specified four safety-critical requirements for the SFDE, two requirements for the airflow calculator, and two requirements for the fuel calculator. During the entire execution period, the SFDE has these requirements:

- $req_{throttle}$. The throttle output shouldn't be out of the throttle opening range (from 3 to 90 percent) for 1 second.
- $req_{speed}$. The engine speed output shouldn't exceed 628 radians per second (6,000 rpm) for 1 second.
- $req_{EGO}$. During the initial warm-up period (25 seconds), the EGO output should not be out of the range [0, 1] for 1 seccond. After the warm-up, the EGO output should be between 0.03 and 0.97.
- $req_{MAP}$. The MAP output shouldn't exceed one atmosphere.

Assuming that the execution period is 60 seconds, the requirements become these BLTL formulas:

$$req_{\text{throttle}} : \neg\left(F^{60}G^{1}\left(throttle_{\text{out}} < 3 \,||\, throttle_{\text{out}} > 90\right)\right),$$

$$req_{\text{speed}} : \neg\left(F^{60}G^{1}\left(enginespeed_{\text{out}} > 628\right)\right),$$

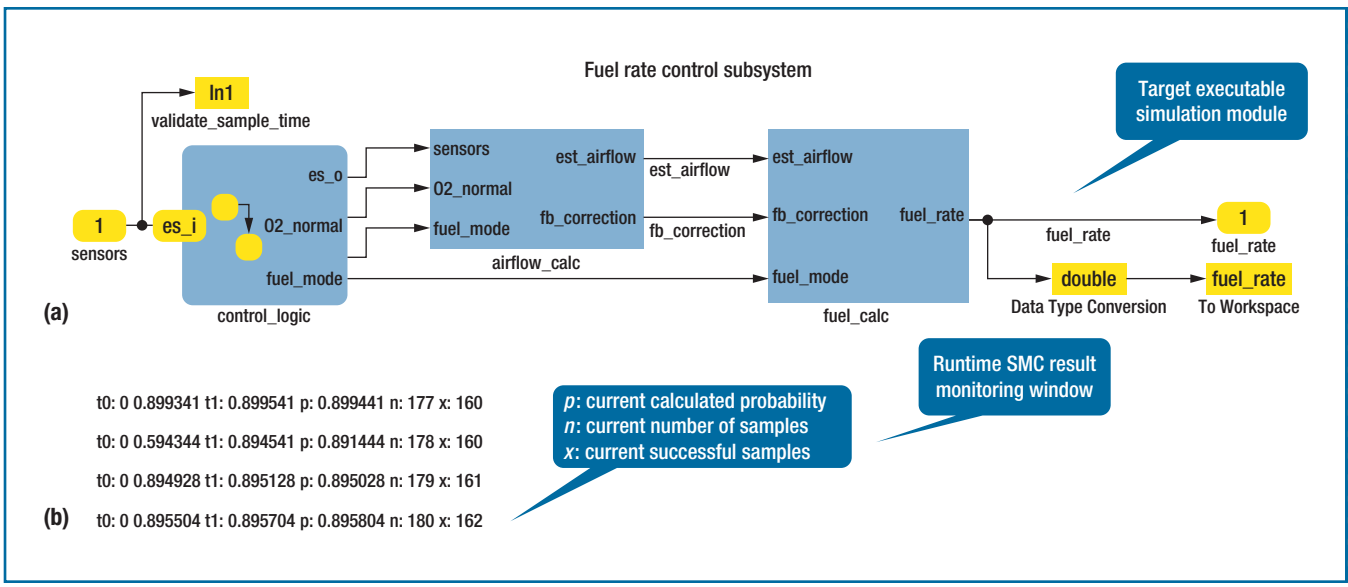



**FIGURE 3.** Screenshots of an SMC experiment on an FFCS. (a) A diagram of the fuel rate control subsystem. (b) Variable values related to the probability of the sensor failure detector and estimator (SFDE) satisfying reqthrottle. The last line in Figure 3b indicates that 162 of the 180 generated sample traces satisfy $req_{throttle}$ so far. That line also indicates that the probability of the SFDE satisfying $req_{throttle}$ is 0.895604.

$$req_{\mathrm{EGO}} : F^{60}\left(\begin{array}{l}\left(\begin{array}{l}\mathrm{warmup} = \mathrm{true} \rightarrow \\ \neg G^{1}\left(EGO_{\mathrm{out}} < 0 \,\|\, EGO_{\mathrm{out}} > 1\right)\end{array}\right) \wedge \\ \left(\begin{array}{l}\mathrm{warmup} = \mathrm{false} \rightarrow \\ \neg G^{25}\left(EGO_{\mathrm{out}} < 0.03 \,\|\, EGO_{\mathrm{out}} > 0.97\right)\end{array}\right)\end{array}\right),$$

$$req_{\mathrm{MAP}} \neg\left(F^{60} G^{0.1}\left(MAP_{\mathrm{out}} > 1\right)\right),$$

where $F^t f$ means that $f$ eventually occurs in $t$ seconds, and $G^t f$ means that $f$ always occurs in $t$ seconds.

**Allocating the reliability requirement.** Because all the FFCS components are combined sequentially, we can calculate the FFCS's reliability $R_T$ by multiplying the reliabilities of the components of the target $R_i'$:

$$R_T = \prod_{i=1}^{n} R_i',$$

where $n$ is a total number of components.

To satisfy the FFCS's reliability (0.9999), we allocated the components' reliability goals via discussion with FormalWorks experts: 0.99997 for the SFDE, 0.99997 for the airflow calculator, and 0.99997 for the fuel calculator.

**Calculating each component's probability.** To calculate probability, we use SMC. (We discuss this in more detail later.)

**Validating each component's reliability.** We can calculate the reliability of $R_i'$ by assigning a weight to each requirement:

$$R_i' = \sum_{req_{ij} \in REQ} \left(w_{req_{ij}} \times P\left(req_{ij}\right)\right),$$

where $w_{req_{ij}}$ is a weight value for $req_{ij}$.

Again, through discussion with the experts, we determined the weight values: $w_{throttle}$ = 0.11, $w_{speed}$ = 0.45, $w_{EGO}$ = 0.09, and $w_{MAP}$ = 0.35. This indicates that the speed and MAP sensors are more safety-critical than the throttle and EGO sensors. We will explain how to validate the reliability of the SFDE in the next section.

## SMC Experiments

We performed all experiments on a 64-bit Windows 7 Professional machine with a 3.40-GHz Intel i5 and 8 Gbytes of memory. We used a Simulink/Stateflow FFCS model in Matlab R2010a. We simulated the model using the Matlab simulator to generate sample execution traces. To validate whether the model satisfies the reliability goal (0.9999), we applied Bayesian interval estimation testing (BIET), an SMC technique.[9] To obtain a precise probability result (a goal of $1 - 10^{-4}$), we set the SMC precision parameters to $d$ = 0.00005 and $c$ = 0.9999 for BIET, where $d$ is a half-size of an estimation interval that will contain the probability result and $c$ is the coverage goal of the estimation interval.

TABLE 1

**Table 1. The statistical-model-checking results for validating the reliability of the sensor failure detector and estimator. The component's reliability was 0.999973.**

| Requirement | Weight | Probability | No. of samples | No. of failed samples | Trace generation time (hrs.) | BLTL model-checking time (hrs.)* | BIET analysis time (hrs.)* | Total verification time (hrs.) |
|---|---|---|---|---|---|---|---|---|
| $req_{throttle}$ | 0.11 | 0.999889 | 776,747 | 85 | 317.17 | 0.75 | 0.99 | 318.91 |
| $req_{speed}$ | 0.45 | 0.999989 | 92,098 | 0 | 37.99 | 0.19 | 0.26 | 38.44 |
| $req_{EGO}$ | 0.09 | 0.999933 | 533,735 | 35 | 220.91 | 0.75 | 1.32 | 222.23 |
| $req_{MAP}$ | 0.35 | 0.999989 | 92,098 | 0 | 38.01 | 0.20 | 0.26 | 38.47 |

** BLTL stands for bounded linear temporal logic; BIET stands for Bayesian interval estimation testing.*

Figure 3 shows a snapshot of an FFCS simulation running with SMC. In Figure 3a, the three component blocks correspond to the FFCS components in Figure 2 (for example, the **control_logic** block corresponds to the SFDE). The **sensors** block represents all four sensor inputs; the **fuel_rate** block represents the fuel rate output.

In Figure 3b, the SMC tool displays variable values related to the probability that the SFDE satisfies $req_{throttle}$. Specifically, $p$ is a calculated probability, $n$ is the number of sample simulation traces so far, and $x$ is the number of successful traces so far. For example, the last line in Figure 3b indicates that 162 of the 180 generated traces satisfy $req_{throttle}$. That line also indicates that the probability of the SFDE satisfying $req_{throttle}$ is 0.895604 so far.

We built a stochastic environment model that generates random faults at the sensors. We made a random-fault generator module and connected it to the sensors. The random faults are modeled by four independent Poisson processes with different arrival rates. The mean interarrival fault rate is 8 for the throttle sensor, 10 for the speed sensor, 9 for the EGO sensor, and 7 for the MAP sensor. For simplicity, we assume that all FFCS operations have the same occurrence rate. For a larger, more complex system, we would have to consider the operational profile so that the most frequently used operation would have the most testing.

We implemented the BLTL model checker (as a proof-of-concept prototype) in 500 lines of Matlab script to evaluate the eight functional-safety properties. In this case, it evaluates $req_{throttle}$, $req_{speed}$, $req_{EGO}$, and $req_{MAP}$ over Matlab/Simulink simulation traces.

We implemented the BIET statistical analyzer (http://pswlab.kaist.ac.kr/tools/SMC) in 50 lines of Matlab script. The BIET analyzer is independent from the model checker and functional-safety requirements.

We plan to implement and publicly release a general model checker that can evaluate arbitrary BLTL formulas over Matlab/Simulink simulation traces. The BLTL model checker and the BIET analyzer will be reusable for other target systems without modification.

## Experiment Results

Table 1 lists the results of applying SMC to the SFDE. On the basis of the probabilities and weight values in the table, we calculate $R_i'$ as

$$R_i' = 0.11 \times 0.999889 + 0.45 \times 0.999989 + 0.09 \times 0.999933 + 0.35 \times 0.999989 \doteqdot 0.999973 .$$

Because the calculated reliability is higher than the goal (0.99997), we conclude that the SFDE satisfies the goal. In total, the experiments consumed approximately 377 Mbytes for simulating the FFCS and 5 Mbytes for BLTL trace checking and BIET analysis.

Generating trace samples consumes 99 percent of the total verification time (for example, 317.17 out of 318.91 hrs. for $req_{throttle}$). So, we can significantly reduce the verification time by generating sample traces in parallel. Because the generated random samples are independent from each other (that is, Bernoulli-independent, identically distributed random samples), we can run multiple simulators on multiple machines to accelerate trace generation. This lets us assess a target component's reliability within a modest time frame by running hundreds of simulators on a cloud computing platform such as Amazon EC2 (Elastic Compute Cloud). For example, with 100 machines, we can calculate a probability for $req_{throttle}$ in approximately five hours (317.17/100 + 0.75 + 0.99).

To further reduce verification time, we plan to apply hybrid SMC techniques that are faster than BIET.[10]

## ABOUT THE AUTHORS

**YOUNGJOO KIM** is a full-time researcher at S-Core. Her research interests include automated software testing and statistical model checking. Kim received an MS in computer science from the Korea Advanced Institute of Science and Technology. Contact her at jerry88.kim@gmail.com.

**OKJOO CHOI** is a research assistant professor at the Korea Advanced Institute of Science and Technology's Department of Computer Science. Her research interests include software process, software safety, and reliability. Choi received a PhD in computer science from Sookmyung Women's University. Contact her at okjoo.choi@kaist.ac.kr.

**MOONZOO KIM** is an associate professor at the Korea Advanced Institute of Science and Technology's Department of Computer Science. His research interests include automated software testing and verification, concurrent program testing, and formal analysis of embedded software. Kim received a PhD in computer and information science from the University of Pennsylvania. He's a member of IEEE and ACM. Contact him at moonzoo@cs.kaist.ac.kr.

**JONGMOON BAIK** is an associate professor at the Korea Advanced Institute of Science and Technology's Department of Computer Science. His research interests include software process modeling, software economics, software reliability engineering, and software Six Sigma. Baik received a PhD in computer science from the University of Southern California. He's a member of IEEE and ACM. Contact him at jbaik@kaist.ac.kr.

**TAI-HYO KIM** is the CEO of FormalWorks. His research interests include formal methods and worst-case execution time analysis. Kim received a PhD in computer science from the Korea Advanced Institute of Science and Technology. Contact him at taihyo.kim@formalworks.com.



Many safety-critical system domains, such as the automotive or avionics domains, have adopted model-driven development. So, industries in those domains can incorporate our framework seamlessly. Adopting our framework will increase system reliability and decrease development costs through early detection of design faults or incorrect reliability allocation.

### Acknowledgments

National Research Foundation of Korea grants 2012046172 and 2010-0014375, Ministry of Knowledge Economy/Korea Evaluation Institute of Industrial Technology grant 10041752, and Dual-Use Technology Program grant UM11014RD1 in Korea supported this research.

### References

1. D.S. Herrmann, *Software Safety and Reliability*, IEEE CS, 1999.
2. *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, Int'l Electrotechnical Commission, 2003.
3. *ISO 26262, Road Vehicles—Functional Safety*, Int'l Org. for Standardization, 2011.
4. *System Reliability Toolkit*, Reliability Information Analysis Center, 2005.
5. J. Lauber, T.M. Guerra, and M. Dambrine, "Air-Fuel Ratio Control in a Gasoline Engine," *Int'l J. Systems Science*, vol. 42, no. 2, 2011, pp. 277–286.
6. M.H. Halstead, *Elements of Software Science*, Elsevier, 1977.
7. A. Stellman and J. Greene, *Applied Software Project Management*, O'Reilly Media, 2005.
8. W.S. Humphrey, *PSP: A Self-Improvement Process for Software Engineers*, Addison-Wesley Professional, 2005.
9. P. Zuliani, A. Platzer, and E.M. Clarke, "Bayesian Statistical Model Checking with Application to Stateflow/Simulink Verification," *Proc. 13th ACM Int'l Conf. Hybrid Systems: Computation and Control* (HSCC 10), ACM, 2010, pp. 243–252.
10. Y. Kim and M. Kim, "Hybrid Statistical Model Checking Technique for Reliable Safety Critical Systems," *Proc. IEEE Int'l Symp. Software Reliability Eng.* (ISSRE 12), IEEE CS, 2012; http://pswlab.kaist.ac.kr/publications/issre2012_yjkim.pdf.

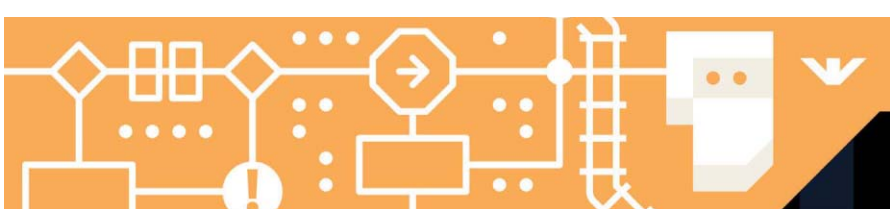
# Engineering Air Traffic Control Systems with a Model-Driven Approach

**Gabriella Carrozza**, SESM

**Mauro Faella**, Critiware

**Francesco Fucci, Roberto Pietrantuono, and Stefano Russo**, Federico II University of Naples

*// Testing software in air traffic control systems costs much more than building them. Software engineers strive to find methodological and process-level solutions to balance costs and to better distribute verification efforts among all development phases. Model-driven approaches could provide a solution. //*



**ONE OF THE FUNDAMENTAL** pillars of air traffic management (ATM) is air traffic control (ATC). ATC systems are software-intensive critical systems that assure that aircraft are safely separated in the sky when they fly and at airports when they land and take off (www.eurocontrol.int/articles/what-air-traffic-management). An ATC system manages all ground and en route flight operations, with the aim of preventing collisions and organizing traffic flow.

To build software for ATC systems, the most consolidated development process model is by far the V-model. Its key benefit is that it accounts for verification and validation (V&V) at early stages—as soon as requirements are elicited—which allows development and V&V activities to occur in parallel flows. The V-model defines criteria for testing on the basis of what will actually be produced, not on what was already produced.

However, market pressures require increasingly time- and cost-effective ways to produce and assess software. When talking about foes of software production effectiveness, the prime suspect is usually testing, especially for critical systems. Thinking about testing as requirements are available is certainly important, but it no longer seems sufficient. Testing and on-site maintenance costs are still a relevant concern for manufacturers and system integrators.

As part of a public-private collaboration between the University of Napoli and the Finmeccanica companies Selex Electronic Systems and SESM, we're jointly looking for process-level solutions that can improve how engineers build high-quality software for ATC systems. We've focused in particular on model-driven approaches.

## Looking at the Model-Driven Approach

The main source of cost happens on the left side of the V (see Figure 1), where early verification still isn't well supported by methodologies and tools. A better cost-quality balance requires improvements not only from the testing perspective: design- and process-level reasoning are key issues in optimizing testing efforts, and they have costs as well. Resources required in terms of personnel



0740-7459/13/$31.00 © 2013 IEEE

and skills, poor communication within the team, and minor involvement of end users are V-model deficiencies that affect quality and cost management.

Figure 1 shows the current V-model process (labeled artifacts comply with the MIL-STD-498 standard[1]). The objective of the collaboration team was to improve cost-quality trade-offs without impacting the current well-proven practices in such a process. This translates into finding a solution that can detect more specification and design errors earlier and find inconsistencies among artifacts; it shouldn't alter the main flow of the V-model (with roles and responsibilities); but it should scale with respect to systems complexity.

For these requirements, a model-driven approach seems attractive. We focused specifically on the model-driven architecture (MDA) development paradigm.[2] Besides the claimed advantages in terms of interoperability, portability, and reusability, we're interested in several other key features:

- manual activity in repetitive error-prone tasks is minimized;
- redundant descriptions at different stages of software behavior are avoided by automatic transformations, minimizing inconsistencies;
- early V&V of design artifacts is aided by tools and favored by modeling notation and rules;
- design-oriented flow helps optimize testing effort;
- code can be generated automatically (which is definitely the most striking feature);
- maintenance cost is also reduced because the effort of introducing a change at the upper level can be minimized by automatic transformations model to model and model to code; and
- compared to pure text, models are less prone to misinterpretation because they dramatically reduce the possibility of misunderstandings on artifacts between different teams and stakeholders.

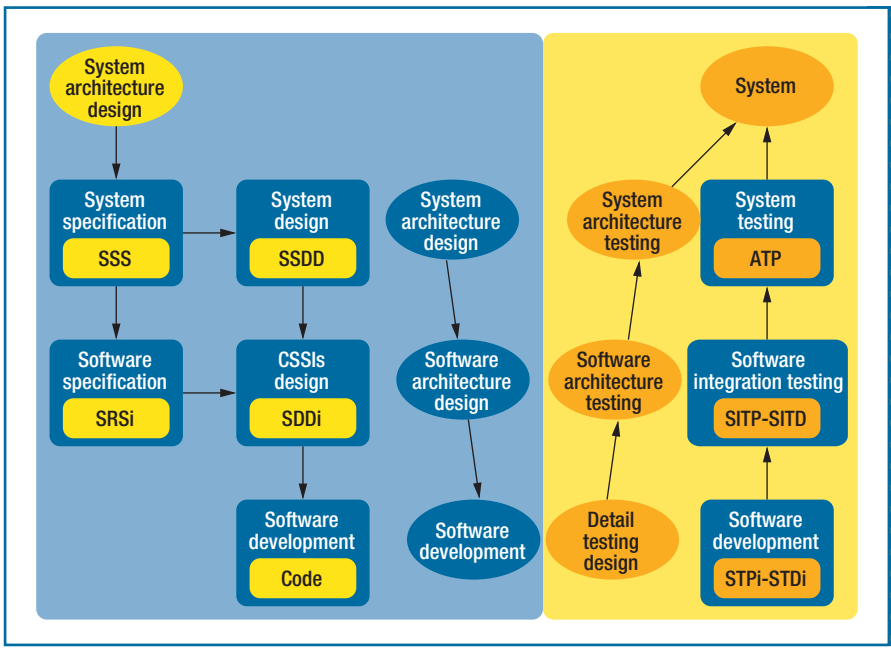



**FIGURE 1.** The reference V-model process with phases of the development process and the associated documents to produce (compliant with the MIL-STD-498 standard). The left side reports the design and coding phases, and the right side, the corresponding testing phases.

We can distinguish two major benefits in a possible integration of MDA into the V-model:

- direct testing and maintenance cost reduction through early defect detection, because the idea of the V-model verifying correctness and consistency at each stage would be enforced, and
- further cost reduction coming from the possibility of generating code automatically, favoring reuse, and easing updates and maintenance actions during operation.

These benefits don't contradict the V-model; rather, they improve and refine its pillars. So, integrating MDA into the adopted V-model was the next step for us. But, again, MDA alone doesn't suffice, and what it can't cover requires integration.

## MDA and Model-Driven Testing in a V-Model

Incorporating a model-driven way of thinking in a full development cycle won't be accomplished by simply placing MDA steps in the design or coding phase. If we want real benefits, we must address how to deal with phases not covered by MDA and how existing, well-proven activities will interact with those of MDA.

In ATC systems engineering, it's important to optimize testing activity. MDA primarily focuses on the development side. Verification is basically supported as cross-checking for design

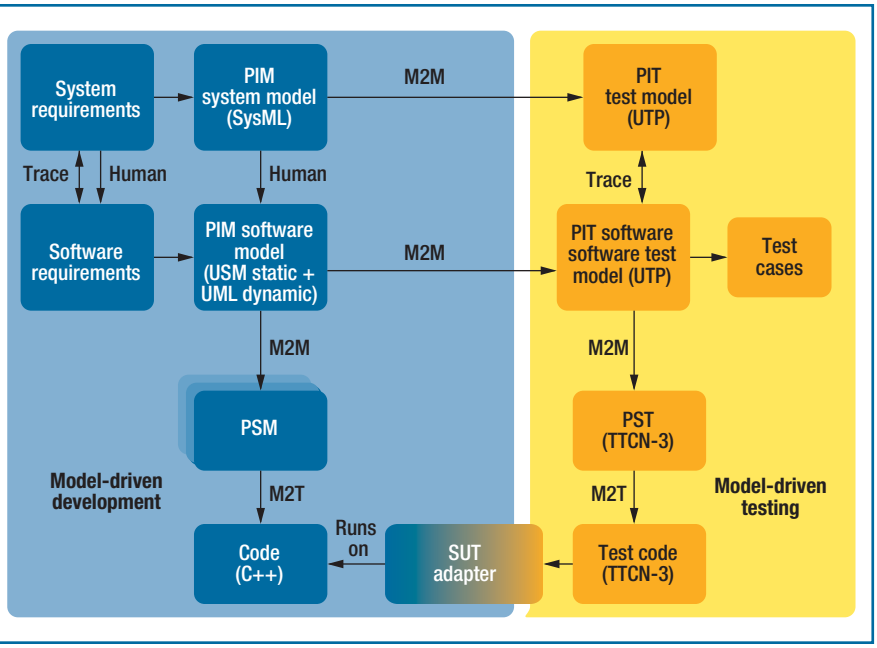



**FIGURE 2.** An overview of our proposed process. The left side reports the phases from requirements to code; the right side reports the testing activities. The figure highlights the human-made ("human") and automatic (M2M and M2T) development phases.

artifacts' consistency but is typically neglected. Model-driven testing (MDT) addresses this type of problem by shifting MDA concepts into testing.[3] Nevertheless, these two practices aren't fully integrated and aren't seen under the same umbrella in everyday work. Like MDA, MDT also proposes platform-independent and platform-specific models (PIT and PST, respectively, where T stand for "test"). And just as MDA does, MDT can reduce testing cost by deriving test cases automatically from these models.[3,4]

Today, the few companies investing in MDT don't usually manage the whole process automatically; rather, they create models manually or by partially reusing MDA design models (for example, by adding stereotypes or profiles to UML models, such as the UML2 Testing Profile[5]). Our solution lets MDA and MDT flow in parallel along the entire process, with model-to-model (M2M) transformations generating PIT and PST software automatically from design models. (The design models can be platform-independent models [PIMs] and PIM software and platform-specific models [PSMs].[2]) Figure 2 shows the implemented links between MDA and MDT in our proposed solution.

## The Integrated Process Model

The opening step of the defined development process involves a requirements analysis performed by domain experts. Then, two activities run in parallel: PIM creation and software requirement specification.

Both the PIM and PIM software have two complementary views: the *static view* describes entities and their structural relationships, and the *dynamic view* describes runtime behavior. The system-level PIM is described in SysML diagrams (for example, through requirement, block, or state machine diagrams) and transformed into PIM software via software requirements. The PIM software is described in UML2 and primarily focuses on *component diagrams*, modeling the relationships among components; *state machine diagrams*, describing the behavior of components in terms of finite-state machines; and *data model diagrams* describing the data managed by the system. These latter diagrams can be external data (exchanged with external actors) or internal data (exchanged among subsystems).

The horizontal M2M transformations use the static view from the PIM and PIM software to generate the PIT and PIT software. The PIT and PIT software are described in the UML Testing Profile (UTP),[5] a standard for defining and specifying test suites in a given domain. The dynamic view helps generate the actual test cases through model-based coverage criteria (for example, algorithms for specific coverage criteria of behavioral diagrams, such as state/transition coverage).

On the left side of the V, we can generate the PSMs from the PIM software by using the correct set of M2M transformation rules, depending on the selected platform. On the right side, the PST is generated in TTCN-3[6] notation through an additional M2M transformation. We chose TTCN-3 because it is a standard and environment-independent notation; in this way, we can reuse a PST across different PSMs.

The last part of the process concerns the M2T transformations of PSMs into source code and of PST into TTCN-3 scripts, and the manual creation of SUT (software under test) adapters, one for each specific implementation.

Finally, the tester executes the generated *test suite* on the SUT. This is done in a specific TTCN-3 runtime environment via the SUT adapter. All the artifacts except software requirements, software models, and the SUT adapter are generated automatically.

### Challenges of Integration

To implement the outlined process, we attempted to find commercial off-the-shelf support tools (for example, IBM Rational Rhapsody), but we couldn't find a complete tool chain able to support the whole integration and generation process. Although several existing tools can cover many steps of the process, they're hard to integrate either because they're produced by different vendors or owing to the different hardware and software platforms they target. Accordingly, we're working on building our own tool chain.

However, besides these technicalities, the trickiest part is to make MDA/MDT flow well enough for building critical software. Indeed, the MDA/MDT paradigm still has several deficiencies. For coping with them, we need to link well-proven V-model activities to MDA/MDT ones.

An inherent problem of MDA/MDT is again about testing, which is right where we expect major benefits. Indeed, testing automation is MDT's most substantial contribution because the testing model contains the static and dynamic view as well. But test automation, and MDT more generally, doesn't necessarily imply test-suite cost-effectiveness.

Through the criteria of the model-based coverage achieved through the dynamic view, MDT automatically creates test cases from the test model; the latter is derived from the design model through the described M2M transformation (from PIM software to PIT software). Thus, test cases are indirectly linked to the design model. This is fine for testing what the system is expected to do against what's specified at the design stage, but it presents some problems:

- In large-scale complex systems, such as the ones we deal with, exercising all the produced test cases isn't feasible.
- In critical systems, conformance to certification standards, and the consequent best practices taken for quality assurance, already provide a certain degree of confidence in functional behavior. The missing link is the fulfillment of nonfunctional requirements. Standards require evidence of quality assurance—the coverage level of functional behavior, RAMS (reliability, availability, maintainability, safety), robustness, and, more generally, satisfaction of dependability requirements still must be demonstrated.

To tackle these issues, we use MDA/MDT for functional test case production. We address nonfunctional testing via consolidated RAMS analysis steps at each stage.

To prevent the number of functional test cases from exploding, we need adequacy criteria and test case selection techniques, so we're exploring solutions that pursue high coverage at a low cost. Along with implementing several coverage criteria for test suites generated from state machines, we're also focusing on similarity-based test case selection techniques.[7]

We use RAMS analysis to identify the most critical software components—in terms of time and budget—for nonfunctional test cases. Post analysis, we can generate test cases to prove software robustness, and/or to run stress and performance tests. Although UTP provides some support for this task, we can exploit a much lower degree of automation compared to functional test case generation.

MDA doesn't cover the uppermost part of the V—that is, from requirements to high-level design. Certification standards of interest (such as DO-178B/DO-178BC/DO-248) deem *requirement management* as a crucial life-cycle activity. Even if MDA provides great support facilities in designing and checking conformance to requirements, it can still be improved through an integrated MDA/MDT approach: executable design models let you exercise them against requirements, and looking at the generated test models helps identify discrepancies between the corresponding design model and requirements.

However, this still isn't enough to cover everything needed in practice, especially from the certification perspective. Requirement completeness, correctness, and traceability among requirements at different levels of abstraction must still be verified via static manual analysis (for example, inspection, checklist, walkthroughs/design reviews) and requirements engineering techniques.[8] For validation, the V-model includes acceptance tests, which give us feedback about user needs and about what's important to prove in terms of system performance.

> Requirement completeness, correctness, and traceability among requirements must still be verified via static manual analysis

A further concern is integration with off-the-shelf components and/or legacy code; this is a common way of developing large systems for ATC, for which MDA/MDT has limited support. The defined flow supports only test case creation for off-the-shelf

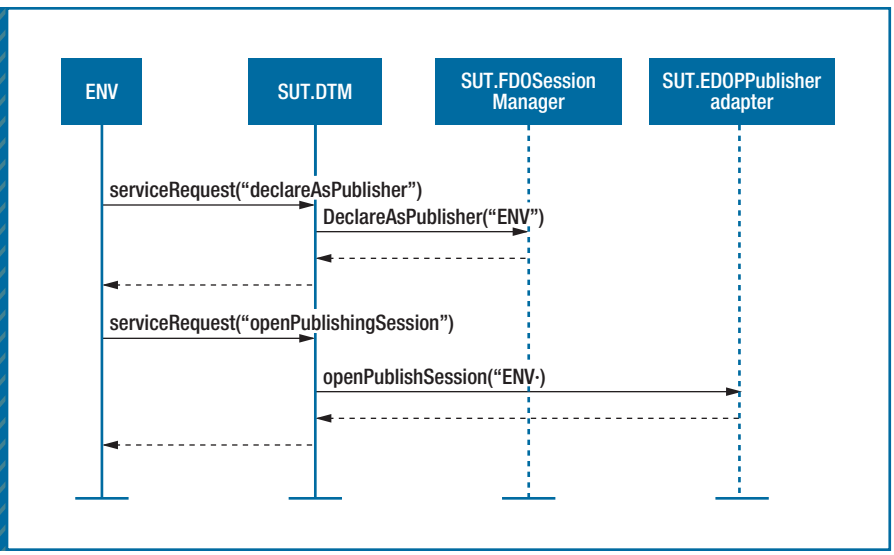



**FIGURE 3.** A test case example. The test case is described by a sequence diagram modeling the interaction among three components of the system under test (SUT).

components (with some documentation support) at the unit level and for their interaction with others in the architecture. This simplifies one task, but the rest of the integration cycle—namely, off-the-shelf search, interface matching, adaptation, and integration strategy—must be managed separately in our V-model.

## An Example

To demonstrate the applicability of the approach, let's look at a model instantiation developed in the context of our industry-university partnership.

The industrial partner is currently developing a project aimed at designing a new generation of ATM and ATC systems. Its goals include optimizing system deployment and maintenance, achieving the performance required to manage increased traffic, and converging toward interoperability with other European ATM systems as required by the Single European Sky ATM Research project (www.sesarju.eu).

The industrial partner designed the ATC system subject of our case study with a component-based approach. The system has tens of thousands of requirements and consists of many interacting deployable components, known as CSCIs (computer software configuration items). Here, we describe the application of our approach to a sub-CSCI of the system *controller working position* component, named *data manager* (DTM). The DTM is our SUT and is responsible for

- managing the transition of flight data objects (FDOs) from external source to the GUI (FDOs include flights and air traffic data such as weather information, altitude, and flight coordinates;
- converting data in different standard formats and storing them into a database; and
- offering publish and subscribe services for FDOs.

DTM has approximately 70 requirements and is meant to be used by other components.

For DTM development, we implemented MDA/MDT in the V-model as shown in Figure 1. We started from an available PIM that we transformed in PIT through M2M translation rules provided by Test Conductor, a commercial plug-in from IBM Rational Rhapsody.

We designed a PIM software with UML2 based on the software requirements specification. The high-level architecture (that is, the static view) consists of six components:

- the FDOStorageManager manages the format conversion and the persistent storage of FDOs in a database;
- the FDOWriterAdapter manages the services to modify the FDOs during a writing session and uses the FDOStorageManager to do so;
- the FDOPublisherAdapter manages the services to publish new FDOs during a publishing session and uses the FDOStorageManager to do so;
- the FDOReaderAdapter provides services to read FDOs during a reading session, using the FDOStorageManager to retrieve the requested data;
- the FDOSessionManager manages three kinds of sessions for external components to manipulate FDOs, namely, writing, publishing, and reading; and
- the FDOChangeNotificationCenter plays the role of message broker, requesting the FDOStorageManager to store FDOs posted by publishers and notifies subscribers about FDO changes.

The dynamic view is described by UML2 statechart diagrams, manually verified against software requirements. At a high level, the DTM component starts in an idle state, waiting for a service request that activates the transition to the busy state. When the requested service is performed without anomalies, it comes back into the idle

state; otherwise, it transits into the error state. When recovery activities are performed, the DTM restarts, resuming to idle.

Test conductor transformation rules automatically generate the PIT software from the static DTM view. The dynamic view helps generate the test cases with the criterion of covering all the states. Figure 3 shows a very simple example of generated test cases; as we go deeper, test cases become more complex.

On the left side of the V, we used the Rhapsody translation rules to transform the PIM software into a PSM, with "platform specific" relating to the specific implementation language, C++, and then to C++ source code. On the right side, we used the ConformiQ tool to generate TTCN-3 scripts from test models; see Figure 4 for an example. These kinds of scripts are executed through Elvior TestCast, which uses a SUT adapter for specific APIs provided by the TTCN-3 execution environment. The SUT adapter that we implemented in Java handles communication between TTCN-3 scripts and the SUT's C++ implementation.

```
testcase State_DTM_Idle_to_WritingSession() runs on Tester system SUT_adapter
{
 var float oldtimer := 0.0;
 var default default_behaviour_ref;
 start_test_case();
 default_behaviour_ref := activate(testerDefaultBehaviour());
 send_ServiceRequest_to_input(DeclareAsPublisherTemplate1);
 oldtimer := 0.0;
 timeoutTimer.start(10.0 - oldtimer);
 alt
 {
  [] timeoutTimer.timeout {}
 }
 timeoutTimer.stop;
 send_ServiceRequest_to_input(OpenPublishingSessionTemplate1);
 setverdict(pass);
 deactivate(default_behaviour_ref);
 end_test_case();
}
```

**FIGURE 4.** A generated TTCN-3 script example. This script implements the diagram from Figure 3.

Model-driven flow is essential on both sides of the V: it allows for parallel evolution of artifacts and favors cross-checking between corresponding activities at any given level of abstraction.

Procedures for integrating MDD into customized processes can bring significant benefits,[9] but it's important not to underestimate the effort needed to set them up or the fact that they might need to be tailored for different systems.

There's still poor interoperability among available tools: a one-size-fits-all tool chain doesn't exist yet. The Rational Rhapsody tool and related plugins cover a relevant slice of our model, but part of the right side of the V (transformation into TTCN-3, to TTCN-3 test scripts, and the TTCN-3 execution environment) must be implemented with other tools. Open source integrated alternatives—for example, based on languages or tools in the Eclipse environment—would be desirable.

One last consideration is the opportunity for radical changes in this area. Besides social, cultural, and economic hurdles,[10] we believe that industry and academia still aren't ready for exploiting MDE benefits systematically. Industry (with reason) is firmly anchored to consolidated processes and practices, which work well even if dated and not in line with modern technologies and paradigms. Academia, on the other hand, misses real-world application scenarios to make research real and to practically assess methodologies and approaches.[11] Concrete experiences in industrial settings are the missing link. Only through more industrial examples will we convince people that certain changes are possible and are worth consideration for improving the quality of delivered critical, large-scale software systems.

## Acknowledgments

MIUR (Ministry of Education, University, and Research) under project PON02_00485_3487758 "SVEVIA" of the public-private laboratory "COSMIC" (PON02_00669) and Finmeccanica under the "Iniziativa Software" project have supported this work. The project Embedded Systems in Critical Domains (CUP B25B09000100007) within the framework of POR Campania FSE 2007-2013 supported authors Roberto Pietrantuono and Francesco Fucci.

## References

1. *MIL-STD-498, Overview and Tailoring Guidebook*, US Dept. of Defense, 1996.

2. J. Miller and J. Mukerji, *MDA Guide Version 1.0.1*, 2003; www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.
3. P.Baker et al., *Model Driven Testing: Using the UML Testing Profile*, Springer, 2010.
4. M. Mussa et al., "A Survey of Model-Driven Testing Techniques," *Proc. Int'l Conf. Quality Software*, IEEE CS, 2009, pp. 167–172.
5. *UML Testing Profile (UTP)*, OMG, 2012.
6. C. Willcock et al., *An Introduction to TTCN-3*, John Wiley & Sons, 2011.
7. H.Hemmati, A.Arcuri, and L.Briand, "Achieving Scalable Model-Based Testing through Test Case Diversity," *ACM Trans. Software Eng. and Methodology*, vol. 22, no.1, 2013, article 6.
8. E. Hull, K. Jackson, and J. Dick, *Requirements Engineering*, Springer, 2010.
9. J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-Driven Engineering Practices in Industry," *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2011, pp. 633–642.
10. B. Selic, "What Will It Take? A View on Adoption of Model-Based Methods in Practice," *Software & Systems Modeling*, Springer, vol. 11, no. 4, 2012, pp. 513–526.
11. L. Briand, "Embracing the Engineering Side of Software Engineering," *IEEE Software*, vol. 29, no. 4, 2012, p. 96.


## ABOUT THE AUTHORS

**GABRIELLA CARROZZA** leads the verification and validation team at SESM. Her research interests include dependability evaluation and assessment of complex software systems, as well as the verification and validation of large critical systems. Carrozza received a PhD in computer and automation engineering from the Federico II University of Naples. Contact her at gcarrozza@sesm.it.

**MAURO FAELLA** is an R&D software engineer at Critiware. His research interests include model-driven approaches and practices in testing activities of critical systems. Faella received an MS in computer engineering from the Federico II University of Naples, Italy. Contact him at mauro.faella@critiware.com.

**FRANCESCO FUCCI** is a PhD student in computer and automation engineering at the Federico II University of Naples. His research interests include verification and validation of complex software systems and fault-tolerance techniques. Fucci received an MSc in computer engineering from the Federico II University of Naples. Contact him at francesco.fucci@unina.it.

**ROBERTO PIETRANTUONO** is a post-doctoral researcher at the Federico II University of Naples. His research interests include software engineering, particularly in the software verification of critical systems, software testing, and software reliability. Pietrantuono received a PhD in computer and automation engineering from the Federico II University of Naples, Italy. He's a member of IEEE. Contact him at roberto.pietrantuono@unina.it.

**STEFANO RUSSO** is professor and deputy head in the Department of Computer and Systems Engineering at the Federico II University of Naples, where he's chairman of the curriculum in computer engineering, and director of the "C. Savy" Laboratory of the National Inter-Universities Consortium for Informatics. His research interests include distributed software engineering, middleware technologies, and dependable software systems. Contact him at stefano.russo@unina.it.

# 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**

IEEE computer society | 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.

FOCUS: SAFETY-CRITICAL SOFTWARE

# Testing or Formal Verification:

## DO-178C Alternatives and Industrial Experience

**Yannick Moy**, AdaCore

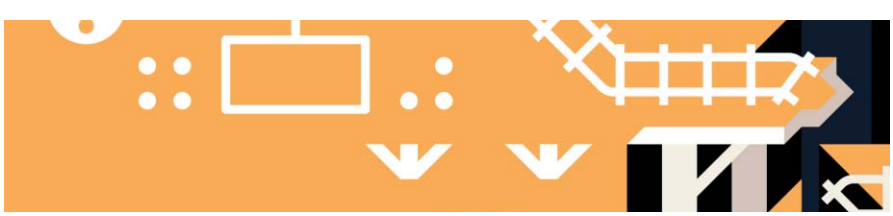
**Emmanuel Ledinot**, Dassault-Aviation

**Hervé Delseny**, Airbus

**Virginie Wiels**, ONERA

**Benjamin Monate**, TrustMySoft

// *Software for commercial aircraft is subject to stringent certification processes described in the DO-178B standard,* Software Considerations in Airborne Systems and Equipment Certification. *Issued in late 2011, DO-178C allows formal verification to replace certain forms of testing. Dassault-Aviation and Airbus have successfully applied formal verification early on as a cost-effective alternative to testing.* //



**AVIONICS IS THE** canonical example of safety-critical embedded software, where an error could kill hundreds of people. To prevent such catastrophic events, the avionics industry and regulatory authorities have defined a stringent certification standard for avionics software, DO-178 and its equivalent in Europe, ED-12, which are known generically as DO-178. The standard provides guidance—objectives as well as associated activities and data—concerning various software life-cycle processes, with a strong emphasis on verification.

The current version, called DO-178B,[1] has been quite successful, with no fatalities attributed to faulty implementation of software requirements since the standard's introduction in 1992. However, the cost of complying with it is significant: projects can spend up to seven times more on verification than on other development activities.[2] The complexity of avionics software has also increased to the point where many doubt that current verification techniques based on testing will be sufficient in the future.[3] This led the avionics industry to consider alternative means of verification during the DO-178B revision process. The new standard, DO-178C,[1] includes a supplement on formal methods (see the "What Are Formal Methods?" sidebar), known as DO-333[4], which states the following:

> Formal methods might be used in a very selective manner to partially address a small set of objectives, or might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.

Although this permission to replace part of testing with formal verification is quite new, we've successfully applied this new guidance into a production-like environment at Dassault-Aviation and Airbus. The use of formal verification for activities previously done by testing has been cost-effective for both companies, by facilitating maintenance leading to gains in time on repeated activities.

### Formal Verification at the Source-Code Level

DO-178 requires verification activities to show that a program in executable form satisfies its requirements (see Figure 1). For some requirements, verification, which can include formal analysis, can be conducted directly on the binary



0740-7459/13/$31.00 © 2013 IEEE

representation. For example, Airbus uses formal analysis tools to compute the worst case execution time (WCET) and maximum stack usage of executables.[5] For many other requirements, such as dataflow and functional properties, formal verification is only feasible via the source-code representation. DO-178 allows this approach, provided the user can demonstrate that properties established at the source level still hold at the binary level. The natural way to fulfill this objective is to show that requirements at source-code level are traceable down to the object-code level.[6,7] Demonstrating traceability between source and object code is greatly

## WHAT ARE FORMAL METHODS?

According to RTCA DO-333, formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems. The first work on formal methods dates back to the 1960s, when engineers needed to prove the correctness of programs. The technology has evolved steadily since then, exploiting computing power that has increased exponentially. In DO-333, a formal method is defined as "a formal model combined with a formal analysis." A model is formal if it has unambiguous, mathematically defined syntax and semantics. This allows automated and exhaustive verification of properties using formal analysis techniques, which DO-333 separates into three categories: deductive methods such as theorem proving, model checking, and abstract interpretation. Today, formal methods are used in a wide range of application domains including hardware, railway, and aeronautics.

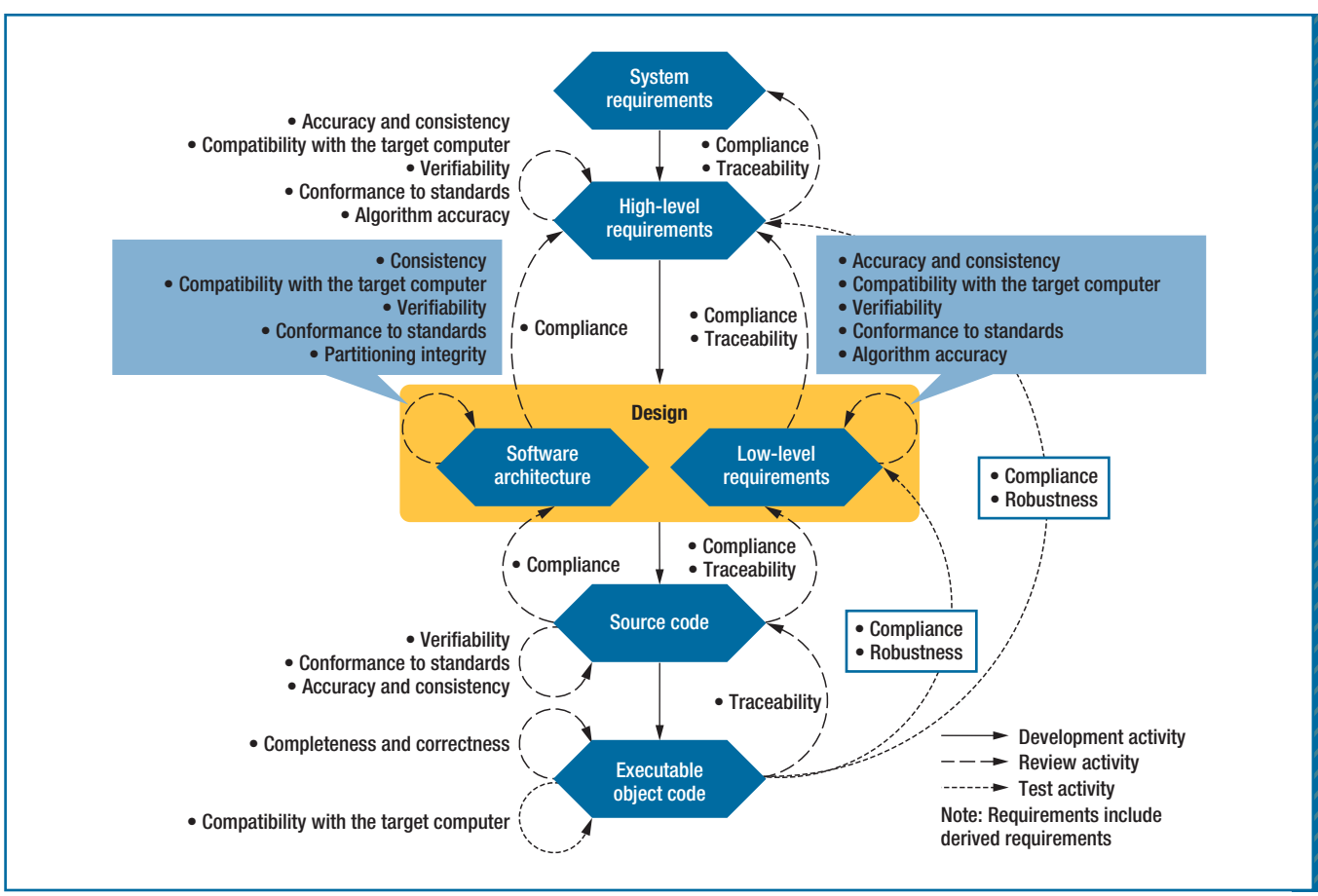



**FIGURE 1.** Activities mandated by DO-178C to fulfill objectives (the labels on the arcs). Verification against requirements is shown in two white boxes with blue borders. (Note that the legend says "Test activity," but DO-333 allows formal verification to replace these testing activities; artwork reproduced with permission of RTCA/EUROCAE.)

## WHAT ARE FUNCTION CONTRACTS?

The concept of program contracts was invented by the researcher C.A.R. Hoare in 1969 in the context of reasoning about programs. In the mid-1980s, another researcher, Bertrand Meyer, introduced the modern function contract in the Eiffel programming language. In its simplest formulation, a function contract consists of two Boolean expressions: a precondition to specify input constraints and a postcondition to specify output constraints. Function contracts have subsequently been included in many other languages, either as part of the language (such as CodeContracts for .NET or contracts for Ada 2012) or as an annotation language (such as JML for Java or ACSL for C). Contracts can be executed as runtime assertions, interpreted as logic formulas by analysis tools, or both.

facilitated by using qualified tools for purposes such as enforcing coding restrictions against features that would complicate traceability, by applying appropriate compiler options to preserve control flow, and by using code traceability analyses prepared by compiler vendors.

Assuring the correctness of the compiler's translation of source code into object code is, of course, important. Trust can be based on examination of the compiler itself (the tool qualification process) or the compiler's output. The former approach (qualifying the compiler) is rare because of the effort involved. The latter approach provides the relevant degree of assurance through the multiple and overlapping activities required by DO-178, including the hardware/software integration testing and the verification of untraceable object code.

The form of verification required by DO-178 is mostly based on requirements, both for verifying high-level requirements, such as "HLR1: the program is never in error state *E1*," and for verifying low-level requirements, such as "LLR1: function *F* computes outputs *O1*, ..., *On* from inputs *I1*, ... *Im*." For both HLRs and LLRs, the DO-178 guidance requires in-range (compliance) and out-of-range (robustness) verification, either by testing or by formal verification.

Compliance requirements focus on a program's intended nominal behaviors. To use formal verification for these requirements, you first express the requirement in a formal language—for example, HLR1 can be expressed as a temporal logic formula on traces of execution or as an observer program that checks the error state is never reached. Then, you can use symbolic execution techniques to check that the requirement is respected. The Java PathFinder tool used at NASA and the Aoraï plug-in of Frama-C implement this technique.[8] As another example, you can express LLR1 as a logic function contract (see the "What Are Function Contracts?" sidebar). Then, you use various formal analyses to check that the code implements these formal contracts, although deductive methods typically perform better here, as demonstrated by the operational deployment of tools such as Caveat/Frama-C[5,8] and SPARK.[9]

Robustness requirements focus on a program's behaviors outside its nominal use cases. A particularly important robustness requirement is that programs are free from runtime errors, such as reading uninitialized data, accessing out-of-bounds array elements, dereferencing null pointers, generating numeric overflows, and so on, which might be manifest at runtime by an exception or by the program silently going wrong. Formal analyses can help check for the absence of runtime errors. Model checking and abstract interpretation are attractive options because they don't require the user to write contracts, but they usually suffer from state explosion problems (meaning the tool doesn't terminate) or they generate too many false alarms (meaning the tool warns about possible problems that aren't genuine). A successful example of such a tool is Astrée,[5] which was specifically crafted to address this requirement on a restricted domain-specific software. Deductive verification techniques require user-written function contracts instead of domain-specific tools and don't suffer from termination problems or too many false alarms. These techniques are available in Caveat,[5] Frama-C,[8] and SPARK.[9]

## Replacing Coverage with Alternative Objectives

To increase confidence in the comprehensiveness of testing-based verification activities, DO-178 requires coverage analysis. Test coverage analysis is a two-step process that involves requirements-based and structural coverage analyses. Requirements-based coverage establishes that verification evidence exists for all of the software's requirements—that is, that all the requirements have been met. This also applies to formal verification. Structural coverage analysis during testing (for example, statement coverage) aims to detect shortcomings in test cases, inadequacies in requirements, or extraneous code.

Structural coverage analysis doesn't apply to formal verification. Instead, DO-178C's supplement on formal methods, DO-333, defines four alternative activities to reach the structural coverage goals when using formal

verification:[6,7] *cover*, *complete*, *dataflow*, and *extraneous*. The four alternative activities aim to achieve the same three goals, substituting verification cases for test cases in the first one.

### Cover: Detect Missing Verification Evidence

Unlike testing, formal verification can provide complete coverage with respect to a given requirement: it ensures that each requirement has been sufficiently—in other words, mathematically—verified. But unlike testing, formal verification results depend on assumptions, typically constraints on the running environment, such as the range of values from a sensor. Thus, all assumptions should be known, understood, and justified.

### Complete: Detect Missing or Incomplete Requirements

Formal verification is complete with respect to any given requirement. However, additional activities are necessary to ensure that all requirements have been expressed—that is, all admissible behaviors of the software have been specified. This activity states that the completeness of the set of requirements should be demonstrated with respect to the intended function:

- "For all input conditions, the required output has been specified."
- "For all outputs, the required input conditions have been specified."

Checking that the cases don't overlap and that they cover all input conditions is sufficient for demonstrating the first bullet point. Furthermore, it's easy to detect obvious violations of the second point by checking syntactically that each case explicitly mentions each output. A manual review completes this verification. Note that formal methods can't handle the more general problem of detecting all missing requirements.

### Dataflow: Detect Unintended Dataflow

To show that the coding phase didn't introduce undesired functionality, the absence of unintended dependencies between the source code's inputs and outputs must be demonstrated. You can use formal analysis to achieve this objective. Formal notations exist to specify dataflows, such as the SPARK dataflow contracts[9] or the Fan-C notation in Frama-C,[8] and associated tools automate the analysis.

### Extraneous: Detect Code That Doesn't Correspond to a Requirement

DO-178C requires demonstrating the absence of "extraneous code": any code that can't be traced to a requirement. This includes "dead code" as defined in DO-178C: code that's present by error and unreachable. The relevant section of DO-333 explicitly states that detection of extraneous code should be achieved by "review or analysis (other than formal)." Although formal analysis might detect some such code, computability theory tells us that any practical formal analysis tool (which doesn't generate so many false alarms that it's useless in practice) will be unsound, meaning it will fail to detect some instances of extraneous code. DO-178C doesn't allow unsound tools.

The effort required by this review or analysis depends chiefly on the degree of confidence obtained after completing the previous activities (cover, complete, and dataflow). Testing detects extraneous code as code that isn't executed at runtime. This step detects both unreachable code that can never be executed and unintended functionalities—those that could be executed but aren't triggered by the tests derived from requirements. When you use formal analysis, the previous activities give some degree of confidence that unintended functionalities can be detected.

> Unit proof has replaced some of the testing activities at Airbus on the A400M military aircraft and the A380 and A350 commercial aircraft.

It only remains to detect by review or analysis the unreachable code. Because this is a manual activity, its details vary from project to project.

## Formal Verification of Functional Properties: Airbus

Since 2001, a group at Airbus has transferred formal verification technology—tools and associated methods—from research projects to operational teams who develop avionics software.[5] The technology for verifying nonfunctional properties such as stack consumption analysis, WCET assessment, absence of runtime errors, and floating-point accuracy isn't seen as an alternative to testing and won't be discussed here. Instead, we focus on unit proof,[4,10] which we developed for verifying functional properties. It has replaced some of the testing activities at Airbus for parts of critical embedded software on the A400M military aircraft and the A380 and A350 commercial aircraft.

Within the classical V-cycle development process of most safety-critical avionics programs, we use unit proof for achieving DO-178 objectives related to verifying that the executable code meets the functional LLRs. The term "unit proof" echoes the name of the classical technique it replaces: unit

testing. The use of unity proof diverged from the DO-178B standard (more accurately, it was treated as an alternative method of compliance), so we worked with the certification authorities to address and authorize this alternative. The new DO-178C standard—together with the formal methods supplement DO-333—fully supports the use of unit proof.

The technique of unit proof reduces the overall effort compared to unit testing, in particular because it facilitates maintenance.

Unit proof is a process comprising three steps:

- An engineer expresses LLRs formally as dataflow constraints between a computation's inputs and outputs, and as preconditions and postconditions in first-order logic, during the development process's detailed design activity.
- An engineer writes a module to implement the desired functionality (this is the classical coding activity). The C language is used for this purpose.
- An engineer gives the C module's formal requirements and the module itself to a proof tool. This activity is performed for each C function of each C module.

Different steps are needed when using the theorem-proving tool. An engineer first defines the proof environment, and then the tool automatically generates the data and control flows from the C code. The engineer then verifies these flows against the data and control flows defined during the design phase. Next, the tool attempts to prove that the C code correctly implements the functional properties defined during the design phase. Finally, the engineer analyzes the proof results. The theorem-proving tool is integrated into the standard process management tool, so that this proof process is entirely automated and supported during maintenance.

As discussed earlier, because we perform a verification activity at the source level instead of the binary level, we also analyze the compiler-generated object code, including the effects of the compiler options on the object code, to ensure that the compiler preserves in the object code the property proved on the source code. Within this development cycle, HLRs are expressed informally, so integration verification is done via testing, which includes verification of timing aspects and hardware-related properties. Even when taking into account these additional activities, the technique of unit proof reduces the overall effort compared to unit testing, in particular because it facilitates maintenance.

This approach satisfies the four alternative objectives to coverage:

- *Cover.* Each requirement is expressed as a property, each property is formally proved exhaustively, and every assumption made for formal verification is verified.
- *Complete.* Completeness of the set of requirements is verified by verifying that the dataflow gives evidence that the data used by the source code is conformant with decisions made during design. Based on this guarantee, the theorem-proving tool verifies that the formal contract defined in the design phase specifies a behavior for all possible inputs. Then, we manually verify the formal contracts, to determine that an accurate property exists and specifies the value of each output for each execution condition.
- *Dataflow.* The dataflow verification gives evidence that the operands used by the source code are those defined at the design level.
- *Extraneous.* Except for unreachable code (which can't be executed), all the executable code is formally verified against LLRs. Thus, the completeness of the properties and the exhaustiveness of formal proof guarantee that any code section that can be executed will have no other impact on function results than what's specified in the LLRs. Identification of unreachable code, including dead code, is achieved through an independent, focused manual review of the source code.

There are two manually intensive, low-level testing activities in DO-178: normal range testing and robustness testing. While Airbus has been using formal verification to replace both types of testing (excluding runtime errors), Dassault-Aviation has experimented with formal verification to replace the robustness testing (including runtime errors).

## Formal Verification of Robustness: Dassault-Aviation

Since 2004, a group at Dassault-Aviation has used formal verification techniques experimentally to replace integration robustness testing,[6] where robustness is defined as "the extent to which software can continue to operate correctly despite abnormal inputs and conditions."[1] We've applied these

techniques to flight control software developed following a model-based approach, specifically on the Falcon family of business jets equipped with digital flight control systems. C source code is automatically generated from a graphical model that includes a mix of dataflow and statechart diagrams. The average size of the software units verified by static analyzers is roughly 50 KLOC.

Normal conditions for this software are defined as intervals bounding the model's input variables and the permanent validity of a set of assertions stated at the model level. These assertions are assumptions expected to be met in both normal and abnormal input conditions for the model to operate properly—typically, they're range constraints on arguments to library functions at the model's leaf nodes. Apart from runtime errors, the robustness assertions amount to a few hundred properties stated at the model level and then propagated to the generated C code.

On such software, integration testing is functional, based on pilot-in-the-loop and hardware-in-the-loop activation of the flight control laws. Designing test cases to observe what might happen if some internal assertions break was determined to be costly and inconclusive, so we handle robustness by manually justifying that normal and abnormal external inputs can't lead to assertion failures. A set of design rules facilitate the checking of range properties; we apply them at the software-modeling level and use a custom checker to verify them. These rules made a manual justification possible.

We anticipated that strengthening the manual analysis of range constraints through mechanized interval propagation and abstract interpretation would be beneficial. But we couldn't compare the benefits of this process evolution on the baseline process by simply comparing past testing cost and present formal verification cost: formal verification supplements an activity that was never performed through testing, just through human analysis.

To mechanize the analysis through formal proof of the assertions, we use two static analyzers that collaborate and share results on the Frama-C platform. Approximately 85 percent of these assertions are proved by abstract interpretation using Frama-C's value-analysis plug-in, and the remaining assertions are proved by deductive verification using Frama-C's WP plug-in and a set of automated theorem provers. The value-analysis plug-in takes into account IEEE 754-compliant numerical precision; while propagating intervals, it also verifies the absence of runtime errors, in particular, the absence of overflows and underflows.

As far as the verification process is concerned, once the integrated flight control software is sufficiently stable, a static analysis expert, in cooperation with a model expert, initially performs the formal robustness verification. The critical issue is to add a few extra assertions to be conclusive about the return values for the numerically intensive library functions. Finding them requires both deep knowledge of the model and abstract interpretation expertise. It takes roughly a person-month effort to set up the Frama-C analysis script and to tune any manually added assertions. Then the model verifiers—an independent group from the model development team—can autonomously replay and update the analysis until some substantial algorithmic change in the model requires revisiting the extra assertions, possibly with some support from the formal verification expert.

Design-rule verification and manual assertion analysis is estimated to take a person-month of effort by the independent control engineers (not software engineers) in charge of model verification. This effort must be repeated for every software model release, so there's no economic gain for a single release. However, because robustness verification is a recurrent task that's automated once the setup phase is complete, this rather long preparation provides a significant competitive advantage for repetitive analyses. The gain is roughly a person-month per flight software release.

This approach satisfies the following alternative objectives to coverage:

- *Cover.* An engineer handles abnormal input conditions through larger intervals and no other assumptions. The tool performs abstract interpretation with no assumptions other than those required to ensure hardware-dependent numerical consistency.
- *Complete.* A manual peer review of the set of assertions in the libraries and in the model ensures that robustness requirements are complete. This is facilitated by the simplicity of typical assertions, 90 percent of which are interval constraints.
- *Dataflow.* An engineer formally specifies dataflows at the model level, using a dataflow formalism. Qualification of the code generator ensures no unintended dataflow

> Because robustness verification is a recurrent task, the gain is roughly a person-month per flight software release.

## ABOUT THE AUTHORS

**YANNICK MOY** is a senior engineer at AdaCore, working on static analysis and formal verification tools for Ada and SPARK programs. He previously worked on similar tools for C/C++ programs at PolySpace, INRIA research labs, and Microsoft Research. Moy received a PhD in formal program verification from Université Paris-Sud. Contact him at moy@adacore.com.

**EMMANUEL LEDINOT** is a senior expert in formal methods applied to software and system engineering at Dassault-Aviation and was Dassault's representative in the ED-12/DO-178 formal methods group. Ledinot graduated as an engineer from Centrale Paris and has an MS in theoretical computer science from the University of Paris VII. Contact him at emmanuel.ledinot@dassault-aviation.com.

**HERVÉ DELSENY** is an expert in avionic software aspects of certification at Airbus and was a member of the working group in charge of writing issue C of ED-12/DO-178. His professional interests include formal methods and promoting their use in avionics software verification. Delseny has an MS in industrial software from Tours University, France. Contact him at herve.delseny@airbus.com.

**VIRGINIE WIELS** is a research scientist at Onera. She previously worked for NASA on formal verification of the Space Shuttle's embedded software. Wiels received a PhD in formal system development and verification from Ecole Nationale Supérieure d'Aéronautique et d'Espace. Contact her at virginie.wiels@onera.fr.

**BENJAMIN MONATE** is a founder and director at TrustMySoft. He's the former leader of the Software Reliability Laboratory at CEA LIST and a senior expert in formal verification and validation. His research interests include application of formal methods to static and dynamic analysis of programs as well as their certification and methodologies of deployment. Monate has a PhD from Université Paris-Sud Orsay. Contact him at benjamin.monate@cea.fr.

relationship at the source-code level compared to the design model.

Airbus and Dassault-Aviation were early adopters of formal verification as a means to replace manually-intensive testing, at a time where the applicable standard DO-178B didn't fully recognize it. New projects can expect to get the same benefits in contexts where the new standard DO-178C explicitly supports it.

Formal methods technology has matured considerably in recent years, and it's attracting increasing interest in the domain of high-integrity systems. Airborne software is an obvious candidate, but DO-178B treated the use of formal methods for verification as an activity that could supplement but not necessarily replace the prescribed testing-based approach. The revision of DO-178B has changed this, and the new DO-178C standard together with its DO-333 supplement offer specific guidance on how formal techniques can replace, and not simply augment, testing.

Experience at Airbus and Dassault-Aviation shows that the use of formal methods in a DO-178 context isn't simply possible but also practical and cost-effective, especially when backed by automated tools. During the requirements formulation process, engineers can use formal notation to express requirements, thus avoiding the ambiguities of natural language, and formal analysis techniques can then be used to check for consistency. This is especially useful because, in practice, the errors that show up in fielded systems tend to be with requirements rather than with code. However, the correct capture of system-functional safety at the software level can't be addressed by formal methods. During the coding phase, formal verification techniques can determine that the source code complies with its requirements.

An interesting possibility that we didn't discuss here is to combine testing with formal verification. This has seen some promising research in recent years,[11] and further industrial experience in this area will no doubt prove useful.

### Acknowledgments

We thank the anonymous reviewers and Benjamin Brosgol for their helpful comments on this article, as well as Cyrille Comar for inspiring us to write it.

## References


1. RTCA DO-178, "Software Considerations in Airborne Systems and Equipment Certification," RTCA and EUROCAE, 2011.
2. NASA ARMD Research Opportunities in Aeronautics 2011 (ROA-2011), research program System-Wide Safety and Assurance Technologies Project (SSAT2), subtopic AFCS-1.3 Software Intensive Systems, p. 77; http://nspires.nasaprs.com/external/viewrepositorydocument/cmdocumentid=320108/solicitationId=%7B2344F7C4-8CF5-D17B-DB86-018B0B184C63%7D/viewSolicitationDocument=1/ROA-2011%20Amendment%208%2002May12.pdf.
3. J. Rushby, "New Challenges in Certification for Aircraft Software," *Proc. 9th ACM Int'l Conf. Embedded Software*, ACM, 2011; www.csl.sri.com/users/rushby/papers/emsoft11.pdf.
4. RTCA DO-333, *Formal Methods Supplement to DO-178C and DO-278A*, RTCA and EUROCAE, 2011.
5. J. Souyris et al., "Formal Verification of Avionics Software Products," *Proc. Formal Methods*, Springer, 2009; http://link.springer.com/chapter/10.1007%2F978-3-642-05089-3_34?LI=true.
6. E. Ledinot and D. Pariente, "Formal Methods and Compliance to the DO-178C/ED-12C Standard in Aeronautics," *Static Analysis of Software*, J.-L. Boulanger, ed., John Wiley & Sons, 2012, pp. 207–272.
7. D. Brown et al., "Guidance for Using Formal Methods in a Certification Context," *Proc. Embedded Real-Time Systems and Software*, 2010; www.open-do.org/wp-content/uploads/2013/03/ERTS2010_0038_final.pdf.
8. P. Cuoq et al., "Frama-C, A Software Analysis Perspective," *Proc. Int'l Conf. Software Eng. and Formal Methods*, Springer, 2012; www.springer.com/computer/swe/book/978-3-642-33825-0.
9. J. Barnes, *SPARK, the Proven Approach to High Integrity Software*, Altran Praxis, 2012.
10. J. Souyris and D. Favre-Félix, "Proof of Properties in Avionics," *Building the Information Society*, IFIP Int'l Federation for Information Processing, René Jacquart, ed., vol. 156, 2004, pp. 527–535.
11. C. Comar, J. Kanig, and Y. Moy, "Integrating Formal Program Verification with Testing," *Proc. Embedded Real-Time Systems and Software*, 2012; www.adacore.com/uploads_gems/Hi-Lite_ERTS-2012.pdf.


See www.computer.org/software-multimedia for multimedia content related to this article.

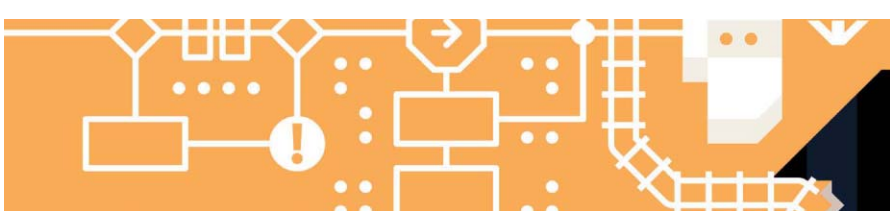
# Strategic Traceability for Safety-Critical Projects

**Patrick Mäder**, Ilmenau Technical University

**Paul L. Jones and Yi Zhang**, US Food and Drug Administration

**Jane Cleland-Huang**, DePaul University

*// An evaluation of traceability information for 10 submissions prepared by manufacturers for review at the US Food and Drug Administration identifies nine widespread traceability problems that affected regulators' ability to evaluate products safety in a timely manner. //*



**FAILURE OF SAFETY-CRITICAL** software systems to operate correctly can cause serious harm to the public—consider devices such as pacemakers, nuclear power systems, and train signals, all of which run on safety-critical software. Therefore, teams building safety-critical software products must perform rigorous risk analyses to identify potentially unsafe conditions and their contributing factors. Many projects conduct this process using techniques such as failure modes and effects analysis, fault tree analysis, and hazard and operability studies. The risk analysis produces a set of system-level requirements specifically designed to mitigate or eliminate faults and reduce the likelihood of accidents.[1] These requirements relate to a broad range of factors including training, testing, process improvements, hardware, human factors, and software design constraints.

In this article, we focus on traceability's role in establishing evidence that device specifications and implementations address identified hazards and their risk control measures (see the "Traceability Standards in Safety-Critical Projects" sidebar).[2] Creating and maintaining trace links can be an arduous, error-prone, and costly process that can have a significant effect on the overall costs and time-to-market for a product.[3–5] Traceability practices, therefore, need to be strategically planned and carefully implemented to provide cost-effective support for evaluating and demonstrating a specific system's safety and security.[6] When traceability isn't implemented strategically, individual stakeholders might create traces that they personally consider to be important or attempt to provide complete trace coverage without considering how the resulting trace links will be used. A brute-force approach to traceability has been shown in practice to be difficult to implement, almost impossible to maintain, and not particularly helpful for providing evidence that a system or device is safe for its intended use.

We present six practices for strategic traceability, derived from our own observations of effective traceability in industrial projects and supported by current literature.[3–6] We also identify nine recurring problems, each of which reduces the effectiveness of traceability verification efforts and increases the difficulty experienced by regulators in evaluating product safety. All the observations in this article are based on actual observations, but the illustrative examples are either fictitious or built on obfuscated data.

## Effective Practices for Tracing in Safety-Critical Projects

Although all the cases reported in this article are safety-critical in nature, many of the problems that we discuss are also applicable to software and



0740-7459/13/$31.00 © 2013 IEEE

## TRACEABILITY STANDARDS IN SAFETY-CRITICAL PROJECTS

Traceability is an established tenet in the software engineering community and is essential for assuring that software is safe for use. Many regulatory agencies of various industry sectors have recognized its importance and have subsequently incorporated it into various standards and guidelines. For example, the Federal Aviation Administration DO-178C standard specifies that at each stage of development, "software developers must be able to demonstrate traceability of designs against requirements."[1] The automotive safety standard ISO 26262:2011 dedicates an entire section to requirements management and states, for example, stating that "safety requirements shall be traceable … to: each source of a safety requirement at the upper hierarchical level, each derived safety requirement at a lower hierarchical level, or to its realization in the design, and the specification of verification."[2]

The ANSI/AAMI/IEC 62304:2006 standard addresses the medical device software development life-cycle processes, requiring "traceability between system requirements, software requirements, software system test, and risk control measures implemented in the software" and that "the manufacturer shall verify and document that the software requirements are traceable to the system requirements or other source."[3] Similarly, the US Food and Drug Administration (FDA) states that traceability analysis must be used to verify that the software design of a medical device implements the specified software requirements, that all aspects of the design are traceable to software requirements, and that all code is linked to established specifications and test procedures.[4] Fergal Mc Caffery and his colleagues provide a comprehensive discussion of traceability requirements for medical device software development.[5]

### References


1. *DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, 2011.
2. *ISO DIS 26262:2011, Road Vehicles—Functional Safety, International Organization for Standardization*, 2011.
3. *ANSI/AAMI/IEC 62304:2006, Medical Device Software—Software Life Cycle Processes*, Assoc. Advancement Medical Instrumentation, 2006.
4. *Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices*, US Food and Drug Administration, 2005.
5. F. Mc Caffery et al., "Medical Device Software Traceability," *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and Andrea Zisman, eds., Springer, 2011, pp. 321–339.


systems traceability in general product development efforts. The following practices can be used to establish traceability that's cost-effective and that provides effective support for constructing a safety-critical system and assessing its safety. We present the practices in the order in which we might expect them to be adopted. In some cases, higher-level practices are dependent on lower-level ones.

### Practice 1: Plan Your Traceability

Project managers should strategically plan traceability in a project's early phases and document it using a traceability information model (TIM).[7] A TIM models the traceable artifact types (requirements, design, code, and so on) and their permitted trace links as a Unified Modeling Language (UML) class diagram. Figure 1 depicts a TIM for a safety-critical project. Artifact

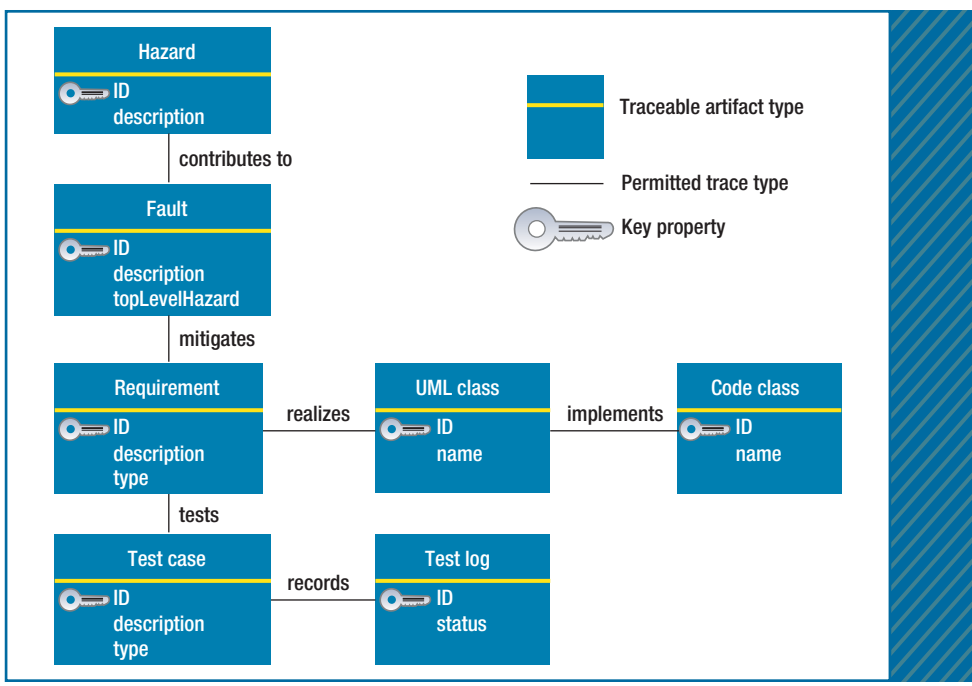



**FIGURE 1.** A typical traceability information model (TIM) for a safety-critical system. The TIM depicts the planned trace paths among development life-cycle artifacts such as hazards, requirements, and design.

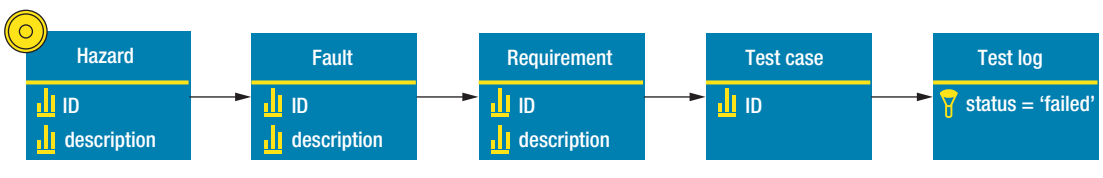



**FIGURE 2.** A Visual Trace Modeling Language (VTML) query modeled over the TIM in Figure 1. The query retrieves, for a given hazard, the ID and description of related faults and requirements that have assigned failed test cases.

types include hazards, faults, requirements, UML classes, code classes, test cases, and a test log. Trace links are only permitted along the specified paths. For example, requirements can be traced to faults and UML classes to requirements. Furthermore, each traceable artifact type is characterized by one or more properties, such as ID, description, or type, used to generate trace queries and that might also be included in trace query results.

### Practice 2: Offer Traceability Tool Support

Creating, maintaining, and using trace links can be time-consuming and arduous. Tracing should therefore be supported using any tool, such as Rational DOORS or Rational RequisitePro, that provides features for establishing, maintaining, and navigating trace links and has the ability to display trace information in formats such as matrices or trace slices. The project environment can also be instrumented to include semiautomated approaches that use information retrieval methods to dynamically generate candidate trace links[5,6] or to infer relationships by analyzing change management systems' commit logs.

### Practice 3: Create Traces Incrementally

In practice, the task of creating, evaluating, and approving traceability links is frequently deferred until very late in the project, at which point it's often conducted by people other than the original developers, testers, and requirements engineers. Consequently, trace links are often incomplete and inaccurate[3,6] and aren't available throughout the project to support development. Instrumenting the environment with tracing tools empowers knowledgeable project stakeholders to create trace links incrementally within the context of their daily work. This reduces the likelihood that trace links will be created solely for approval purposes and allows project stakeholders to benefit from traceability knowledge throughout the project.

### Practice 4: Model Traceability Queries

Traceability queries cover basic lifecycle activities such as finding all requirements associated with currently failed test cases or listing all mitigating requirements associated with a given hazard. Trace queries can be defined in several ways, for example, by using the Visual Trace Modeling Language (VTML), which represents queries as a set of filters applied to the TIM.[7] These filters eliminate unwanted artifacts and define data to be returned by the trace query. Figure 2 shows a VTML query.

### Practice 5: Visualize Trace Slices

In safety-critical systems, trace links established among hazards, faults, mitigating requirements, design, implementations, and test cases are of particular importance.[8] Therefore, instead of presenting traceability material in the form of trace matrices, generate trace slice visualizations in which the hazard is the root node and all direct and indirectly traced artifacts that contribute to mitigating the hazard are shown as a tree. Figure 3 illustrates this with a trace slice for one specific hazard. These slices support safety-related tasks such as helping a regulator to understand how a specific hazard has been addressed in the final system.

### Practice 6: Evaluate Traces Continually

One challenge of implementing a traceability process is that the people performing the tracing tasks often don't directly realize tracing benefits. Furthermore, the current status of the traceability effort is often not visible to individual stakeholders or the project manager. A dashboard that displays the tracing progress for a project can be effective for tracking and managing the project's tracing goals and also for motivating team members to create appropriate trace links. The dashboard can display useful information such as burn down charts showing the percentage of hazards that don't have mitigating requirements, or the percentage of mitigating requirements without passed test cases. This information is generated via trace queries. Personalized views can be created for individual project members.

## Observed Traceability Problems

Unfortunately, current traceability practices often fall far short of accepted principles in software engineering for developing safety-critical systems. Our two US Food and Drug Administration (FDA) research team members systematically evaluated the traceability documentation presented in 10 submissions

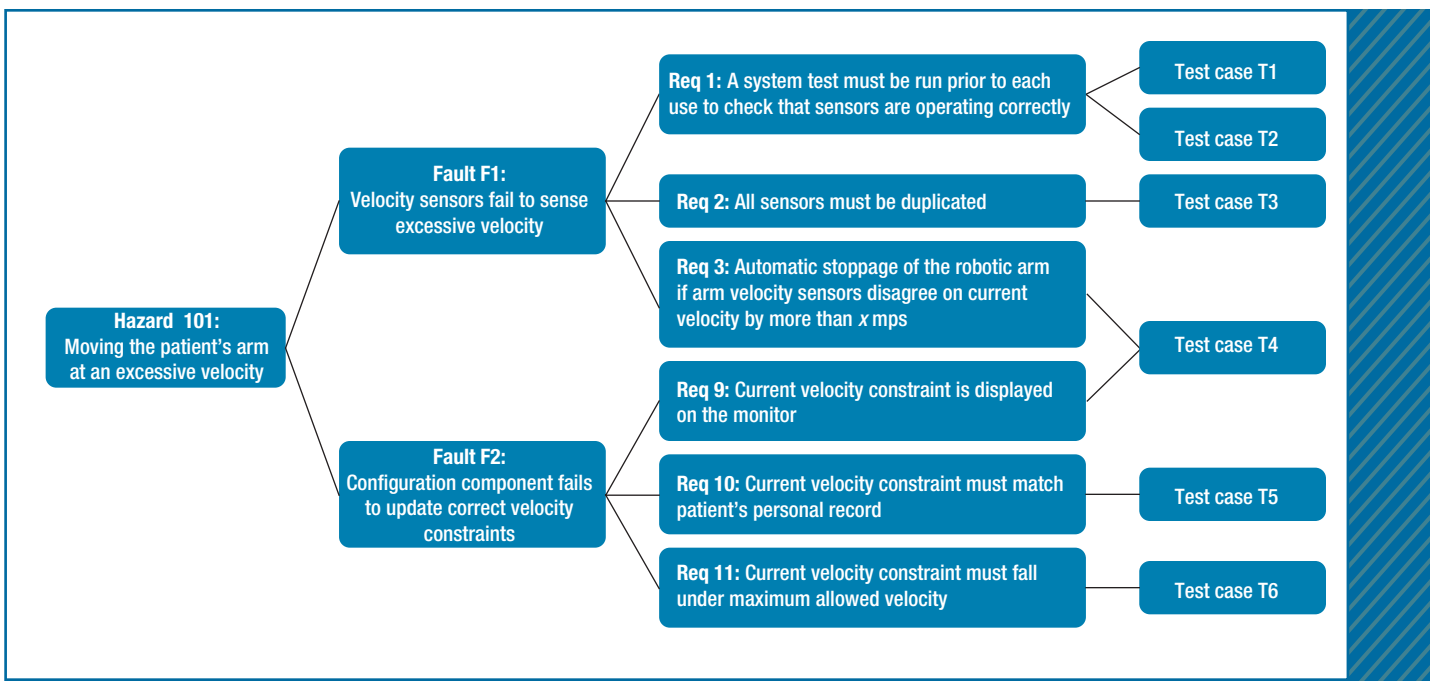



**FIGURE 3.** A simplistic trace slice showing test cases, requirements, and faults associated with a hazard. The information can be retrieved via the VTML query from Figure 2. (Example taken from a therapeutic robotic arm case study.[9])

for FDA medical device approval. Their analysis of submissions revealed several issues from which we identified nine distinct problems, specified in terms of definition, trace instance, and presentation problems.

## Definition Problems

We identified three types of problems in the area of defining trace strategies. Definition problems appeared to have far-reaching impacts on the tracing process and resulted in ad hoc approaches to traceability and uneven, inconsistent coverage among design artifacts.

**Problem 1: Failure to explicitly model a TIM.** Without an explicit and documented trace strategy, developers often expend valuable effort in the wrong places while important traces required for demonstrating or arguing product safety are missing.

The lack of traceability planning introduces numerous issues, such as the problem depicted in Figure 4 in which traces are established directly from design requirements to hazards without any intermediate artifacts. This results in high fan-in—for example, in one case we found 15 requirements mitigating a single hazard—and makes it difficult to understand why a particular requirement is linked to a hazard. A better solution would be to decompose the hazards into contributing factors derived from the initial risk analysis, then to create trace links from risk control measures to contributing factors and contributing factors to hazards.

**Remedy 1.1.** Create a TIM early in the project and assign responsibility to the project manager to ensure that it's followed consistently throughout the development process.

**Remedy 1.2.** Use the TIM to specify how links will be created. To reduce effort, create manual links only for critical requirements, and address other traceability needs through automated techniques,[10] which use information retrieval methods to generate traces on a just-in-time basis.

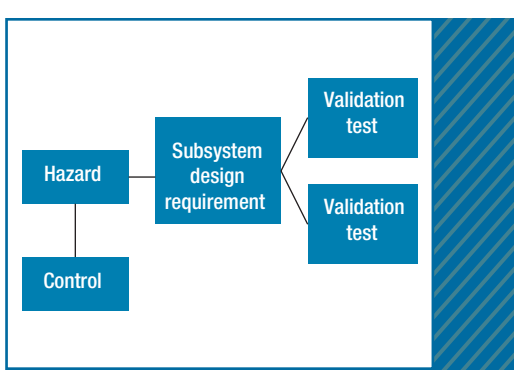



**FIGURE 4.** A TIM that shows a trace path directly from requirements to hazards, missing the important intermediate step of tracing through contributing faults. In this example, hazards were traced directly up to 15 subsystem design requirements in the worst case.

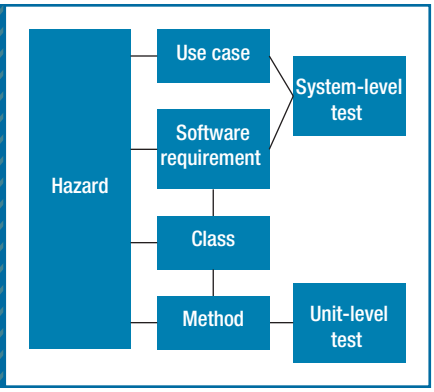



**FIGURE 5.** A TIM that shows redundant trace paths between sets of artifacts. This TIM suffers from multiple granularity problems.

**Problem 2: Trace granularity not clearly defined.** Ill-defined trace granularity leads to unacceptably high, unacceptably low, or mismatched links, making it difficult to determine whether hazards and faults are fully addressed. Our study identified three different kinds of granularity issues:

- *Trace links that are too coarse-grained*. The links don't reference artifacts of interest, but instead reference higher-level artifacts—for example, links established between large abstract sections of the requirements specification and test cases. Traces are too coarse-grained when they don't link the test case to the specific requirement being addressed.
- *Trace links that are too fine-grained*. The links reference low-level parts of an artifact, distracting the stakeholder from the fact that the relation to the higher-level, more coarse-grained artifact is important—for example, we observed multiple individual trace links between a requirement and every single step of a test procedure, introducing unnecessary effort to create and maintain so many links. In this case, the entire set of individual links should be replaced by a single link to the test case.
- *Inconsistent granularity*. A set of trace links are inconsistent in the level of artifacts being traced—for example, we observed a single trace matrix, which includes whole sections of the software requirements specification, traced to a single test case, and then, in the same matrix, trace links from individual requirements to multiple test suites. The mismatch of granularity in the trace matrix makes any degree of automated reasoning concerning test coverage very difficult and also complicates the human reviewer's task. (Defining inconsistent granularity at the link level contributes to the inconsistent link problem described in Problem 5.)

**Remedy 2.** Define trace granularity clearly in the TIM and evaluate trace matrices periodically to ensure that traces are created at the correct granularity. When necessary, you can apply different trace granularities to different subsets of a particular artifact type; however, in this case, the trace matrices should be separated.

**Problem 3: Redundant traceability paths.** Redundant traceability paths defined in the TIM lead to extraneous and possibly diverging traceability matrices. A TIM includes a redundant path if there's more than one way to trace from one artifact type to another. In some cases, redundant paths might be necessary—for example, it's possible that some requirements are explicitly realized in a UML system design while the remaining ones are directly implemented in the source code. In these cases, use redundant paths judiciously. We observed several cases of redundant traceability paths—for example, the project depicted in Figure 5. If redundant links are stored in different traceability matrices and maintained by different stakeholders, there's a high risk of inconsistency.

In one submission, we observed that test cases traced directly to both hazards and to mitigating requirements. The mitigating requirements then traced to hazards, creating a second indirect trace path from test cases, via mitigating requirements, to hazards. After comparing the trace links along both paths, we found many inconsistencies.

**Remedy 3.** Minimize and preferably remove redundant traceability paths in the TIM. If required, inform traceability stakeholders about the purpose of each trace route and ensure that each artifact is traced along one path only. Use trace queries to find redundant traces and eliminate them.

## Trace Instance Problems

We identified three types of problem at the level of individual trace links. These problems often stemmed from definition problems or from ill-defined day-to-day tracing processes. They affect the reviewer's ability to understand relationships between specific artifacts or to perform a complete coverage analysis of a specific hazard.

**Problem 4: Failure to provide unique IDs across the project.** Artifacts can lack unique IDs or names. Moreover, IDs might not be used consistently, in which case traceability information exists but is not useful.

A fundamental principle of traceability is that each traceable artifact must have a unique identifier. Furthermore, prefixes used to distinguish artifact types should be unique across the project as well as intuitive to stakeholders. We didn't see this fundamental principle in many of the submissions we observed. For example, requirements

identified as SYRS25.01 – SYRS25.xx that trace to tests identified as SYRS01 – SYRSxx are hard to distinguish because the requirements and test all share the single SYRS prefix. This introduces unnecessary communication problems.

In a second example (see Figure 6), one requirement specification consisted of paragraphs containing multiple requirements. IDs were directly embedded in the text and weren't unique across the project. A precise, tool-supported evaluation of the existing traces was almost impossible. Changing existing requirements or adding new ones could lead to inconsistent or illogically ordered labels and section numbers and would require significant effort that would likely introduce labeling mistakes.

A further example illustrates why section headings should not be used in lieu of IDs for tracing purposes. A requirements specification contained a table with alarm and alert definitions. These alarms were validated by separate test cases, giving rise to the need to trace an alarm definition to a test case.

Developers created traces by referring to row numbers provided in the first column of the table. The trace R25.6.2-19 –-> RF-0282 defined in Table 1 refers to row 19 in the table and related it to test RF-0282. Although this scheme is logical, it's also highly vulnerable to changes in the requirements specification. Any changes made to sections, tables, or table rows could break existing traces.

...
During the [...], the timeout SHALL (5.5a) be set to 60 seconds. Upon completion of the [...], the default SHALL (5.5b) be set to 30 seconds.
After the specified interval [...], the [...] SHALL (5.5c) turn off and [...].
...

**FIGURE 6.** Power-off timeout from our observed examples. The sequential and embedded nature of the IDs makes it difficult to add new requirements without reassigning IDs, resulting in possible synchronization issues among other design artifacts. For trace purposes, IDs must be assigned permanently.

**Remedy 4.** Artifact IDs are essential for traceability. Carefully define them up front and then consistently use them across all trace links. ID prefixes should allow for an intuitive association with artifact types in a project and should be clearly distinguishable.

**Problem 5: Redundant trace information.** Duplicated trace information occurs in two different forms. In the first case, identical links are included multiple times in the trace matrix, which will lead to future maintenance problems. In one case we observed, 247 of 2,789 traces were redundant. These 247 traces duplicated 167 unique traces, in some cases, up to six times.

In the second case, a complex form of redundancy occurs when similar traces are established at different levels of granularity. For example, a section in an SRS is traced to a part of the systems' design, but also all requirements in that section are individually traced to the same part. This kind of redundancy is difficult to find, and such links are almost impossible to maintain. We observed multilevel redundant traces in almost all of the documents that we reviewed. The TIM depicted in Figure 5 shows code represented at both the class and method levels, which introduces the possibility of multilevel redundancy problems.

**Remedy 5.1.** Prevent duplicated links by storing them in a database-like repository. Either define constraints that prevent redundant links from being created or regularly execute trace queries to find duplicated links and remove them.

**Remedy 5.2.** Whenever possible, avoid modeling multiple levels of a single artifact type in the TIM—for instance, model code at either the class or the method level, but not both. When this is unavoidable because different artifact types must be traced to different levels, avoid tracing a single artifact to

TABLE 1

Example from a requirements specification.*

| | Condition | Type | Requirements | | |
|---|---|---|---|---|---|
| ... | | | | | |
| 19 | [...] Canceled | Always on | Pump | MP-6 | MP-0 |
| ... | | | | | |

* The table appears in Section 25.6.2 of the specification and provides alarm and alert definitions. Traces refer to the section number followed by the row number in the table. Definitions in rows should carry a unique identifier.

```
APSYRS DOORS
|– 4.1.1.0-1 (APSYRS2825) [Name of the requirement]
…
|– 4.15.2.0-4 (APSYRS3968) [Name of the requirement]
    |– AP-SYTPS0023-12000
    |– AP-SYTPS0023-12035
    |– AP-SYTPS0023-12400
|– 4.15.3.0-1 (APSYRS4153) [Name of the requirement]
…
```

**FIGURE 7.** A small excerpt of unprocessed trace information produced by a requirements management tool. This kind of megatable data doesn't provide sufficient information to allow reviewers to directly evaluate product safety.

multiple levels of the same target artifact; that is, trace an individual requirement to either the class level or to the method level, but not to both.

**Problem 6: Important links missing.** Most certifying or approving organizations expect all hazards to be fully covered through trace links from requirements to code and test cases. Missing links indicate insufficient evidence to evaluate whether a hazard has been fully mitigated. By comparing a project's traces and artifacts, it's possible to identify critical artifacts with no associated links. We found examples of untraced mitigating requirements (widows) and untraced test cases (orphans) in all of the cases we observed. We also observed a case in which an important test was listed as passed, even though it neither appeared with the other tests in the test specification nor in any of the traceability matrices.

**Remedy 6.** Use trace queries to perform completeness and coverage analysis of project artifacts to ensure that all critical artifacts are traced.

## Presentation Problems

We identified three problems in the way traceability data was packaged and presented to FDA reviewers. These problems were pervasive across nearly all of the documents we studied and severely reduced the potential benefits that the traceability information could bring to the assessment process.

**Problem 7: TIMs not included in documentation.** It isn't sufficient to have a traceability strategy for a project; it's also important to communicate that strategy to all stakeholders, including external assessors. Without the TIM, an assessor or reviewer must invest considerable time to understand the way artifacts are structured and labeled in the project before he or she can start the core assessment task. A TIM provided as part of a project's documentation enables external reviewers to quickly gain an understanding of the development process. (Only one of the documents we observed contained a TIM.)

**Remedy 7.** Include a TIM in the submitted project documentation so that reviewers can understand artifact naming conventions as well as the traceability paths used throughout the submission.

**Problem 8: Traceability links might be presented in megatables.** It's relatively easy to generate a megatable of trace links from a database or a requirements management tool, but such tables often fail to include sufficient information about the artifacts, and therefore fail to provide adequate support for claims of product safety. Furthermore, this makes reading and comprehending traces in printed reports almost impossible for the reviewers.

We found several examples of trace matrices spanning multiple columns and tens of pages that included only source and target IDs. Although these traces might be technically correct, their usefulness to someone reading the document is limited. If, for example, a reviewer wants to trace a requirement to related test cases, he or she must find the requirement ID within the (potentially unsorted) megatables, retrieve and remember the related test IDs, and then manually browse the table to find the relevant test cases.

One case presented a barely readable screenshot from IBM's Rational DOORS extending over more than 10 pages, and it didn't provide sufficient information to enable interpretation of the traces (see Figure 7).

**Remedy 8.** Maintain traces in a table format, but generate useful views, such as trace slices, that support safety inspections. Utilize tracing tools, such as Rational DOORS, that have the ability to generate and print such views.

**Problem 9: Traceability as an afterthought.** Constructing trace links to merely give the appearance of meeting a regulatory expectation is counterproductive and, if apparent to the reviewer, will diminish confidence in the quality of the development process and the subsequent safety of the delivered system. Furthermore, performing traceability in an ad hoc, after-the-fact fashion means that organizations incur all the costs of creating trace links without experiencing any of its benefits. In several of the submissions we observed, the incompleteness of the trace links and the haphazard effort to document them gave the appearance that traceability had been conducted at the end of the project solely for approval purposes.

**Remedy 9.** Establish tracing processes and instrument the project environment so that traces are created incrementally and accurately maintained throughout a project's lifetime.

- All traceable artifacts and permitted links are clearly defined in a TIM (Problem 1; Remedies 1.1 and 1.2).
- The granularity of each link is clearly defined. Redundant paths are prevented where possible (Remedies 2, 3, and 5.2).
- All traceable artifacts have been assigned meaningful, unique IDs (Remedy 4).
- Traceability is supported by tools (Problem 2; Remedy 5.1).
- Traces are created throughout the project, rather than after the fact (Problem 3; Remedy 9).
- Traces comply with the TIM (Problem 4; Remedy 5.1).
- Traces are used to perform completeness and mitigation analysis on critical artifacts before submission (Remedies 6 and 9).
- Project documentation and submission contains the TIM, all traced artifacts and all traces (Remedies 7 and 9).
- Traces are reported and submitted as useable views and slices (Problem 5; Remedy 8).
- A project dashboard shows the project state by aggregating trace metrics (Problem 6).

**FIGURE 8.** A quick traceability checklist.

These practices and remedies highlight the importance of using traceability strategically in safety-critical projects to systematically build a case for product safety and support the assessment process. We're aware that our advice is contrary to some previous papers that advocate a more brute-force approach in which all requirements are thoroughly traced across the life cycle.[3,4] Although we certainly don't discourage complete traceability coverage, we take a pragmatic approach that focuses efforts on creating the trace links needed to support safety analysis. Emerging technologies that use automated methods to dynamically generate just-in-time trace links can be relied on for other less critical tracing needs.

Figure 8 provides a checklist that summarizes some of the main findings of our study and can be used to help a manufacturer to implement good tracing that serves to support product safety claims.

Additional information, tools, and support for performing tracing in safety-critical projects can be found at the Center of Excellence for Software and Systems Traceability at www.coest.org.

## Acknowledgments

The work in this article was partially funded by the US National Science Foundation grant

## ABOUT THE AUTHORS

**PATRICK MÄDER** is a researcher at the Ilmenau Technical University. His research interests include software engineering with a focus on requirements traceability, requirements engineering, and object-oriented analysis and design. Mäder received a PhD in computer science from the Ilmenau Technical University. Contact him at patrick.maeder@tu-ilmenau.de.

**PAUL L. JONES** is a senior systems/software engineer in the Office of Science and Engineering Laboratories at the US Food and Drug Administration. His research interests include systems and software engineering, safety, and risk management. Jones received an MS in computer engineering from Loyola University. Contact him at paul.jones@fda.hhs.gov.

**YI ZHANG** is a visiting scientist in the Office of Science and Engineering Laboratories at the US Food and Drug Administration. His research interests include formal methods (especially model-based engineering and software static analysis), software testing, software engineering, and cybersecurity, with an emphasis on introducing research advances in these areas to promote the safety and effectiveness of medical devices. Zhang received a PhD in computer science from North Carolina State University. Contact him at yi.zhang2@fda.hhs.gov.

**JANE CLELAND-HUANG** is an associate professor in the School of Computing at DePaul University. She also serves as the North American Director of the International Center of Excellence for Software Traceability. Her research interests include the application of machine learning and information retrieval methods to tackle large-scale and safety-critical software engineering problems, especially in the area of software traceability. Cleland-Huang received a PhD in computer science from the University of Illinois at Chicago. She serves on the editorial board for the *Requirements Engineering Journal* and *IEEE Software* and as associate editor for *IEEE Transactions on Software Engineering*. Contact her at jhuang@cs.depaul.edu.

#CCF-0810924, the Austrian Science Fund (FWF): M1268-N23, and the German Research Foundation (DFG): Ph49/8-1.

The FDA does not endorse any tools or techniques mentioned in this article.

## References

1. P. Bishop and R. Bloomfield, "A Methodology for Safety Case Development," *Proc. 6th Safety-Critical Systems Symp.*, F. Redmill and T. Anderson, eds., Springer, 1998, pp. 194–203.
2. J. Cleland-Huang et al., "Trace Queries for Safety Requirements in High Assurance Systems," *Proc. 18th Int'l Conf. Requirements Eng.: Foundation for Software Quality* (REFSQ 12), Springer, 2012, pp. 179–193.
3. O. Gotel and C. Finkelstein, "An Analysis of the Requirements Traceability Problem," *Proc. 1st Int'l Conf. Requirements Eng.*, IEEE CS, 1994, pp. 94–101.
4. B. Ramesh and M. Jarke, "Toward Reference Models of Requirements Traceability," *IEEE Trans. Software Engineering*, vol. 27, no. 1, 2001, pp. 58–93.
5. J. Cleland-Huang, O. Gotel, and A. Zisman, "Software and Systems Traceability," Springer, 2011; doi:10.1007/978-1-4471-2239-5.
6. P. Mäder, O. Gotel, and I. Philippow, "Motivation Matters in the Traceability Trenches," *Proc. 17th Int'l Conf. Requirements Eng.* (RE 09), IEEE CS, 2009, pp. 143–148.
7. P. Mäder and J. Cleland-Huang, "A Visual Language for Modeling and Executing Traceability Queries," *J. Software and Systems Modeling*, Apr. 2012; doi:10.1007/s10270-012-0237-0.
8. *ANSI/AAMI/IEC 62304:2006, Medical Device Software—Software Life Cycle Processes*, Assoc. Advancement Medical Instrumentation, 2006.
9. A. Frisoli et al., "Arm Rehabilitation with a Robotic Exoskeleleton in Virtual Reality," *Proc. IEEE 10th Int'l Conf. Rehabilitation Robotics* (ICORR 07), IEEE, 2007, pp. 631–642.
10. J. Cleland-Huang et al., "Best Practices for Automated Traceability," *Computer*, vol. 40, no. 6, 2007, pp. 27–35.

# Flight Control Software: Mistakes Made and Lessons Learned

**Yogananda Jeppu**, Moog India Technology Centre

*// Aerospace or flight control systems software development follows a rigorous process, yet software errors still occur. A review of mistakes found during flight control test activities spanning 23 years reveals that the same mistakes recur repeatedly. //*

**FLIGHT SOFTWARE CAN** be developed with strict guidance on development and testing relevant to its safety-critical nature and still fail. The software isn't completely blameless for these mishaps. A root cause might lie in a wrongly entered constant for a filter coefficient, as in the case of the Milstar satellite in 1999.[1] In that case, –1.992476 was entered as –0.1992476, resulting in the loss of approximately US$1 billion. The implementation of imperial units instead of metric units caused the 1998 loss of the Mars Climate Orbiter. The 1996 Ariane 5 failure is an oft-quoted example of an overflow fault causing a mission failure. Thomas Huckle reports additional errors at http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html. Many of these mistakes have similar roots and appear to be universal.

My association with safety-critical flight control software faults started in 1988. In those days, the Honeywell Bull mainframe computer was popular, Matlab was just in, and the coding language was Fortran. We discovered that legacy simulation packages with differential equations, control laws, and guidance models had errors. We were experimenting with Matlab, giving all sorts of input waveforms, and finding out that the Fortran code didn't match the Matlab code. This was the beginning of model-based testing for us. The fact that senior, experienced scientists had been using the software and making mistakes for ages without realizing it was to lead some of us into verification-and-validation careers.

You might say I was fortunate that I didn't witness any major failures in my career in flight controls, but the near misses were memorable. (Looking at the telemetry screen with a red light set in all four channels of the quadruplex system and hearing a pilot's voice on the headphones say "I am experiencing a slat failure warning" is very memorable.) Here, I share what I learned from such mistakes. These lessons come from my experiences with safety-critical software in various organizations that I've worked with: Defense R&D Lab (1988–1995), Aeronautical Development Agency (1995–2007), and, currently, Moog India. I include the year of these instances because the same mistakes tend to recur.

## Incorrect Filter Implementation (1989)

Digital filters are the most common control system element blocks in flight software. When implemented as notch filters, they remove specific frequency components from signals. This removes structural vibrations and prevents such vibrations from entering the control system. When implemented as phase advance filters, they help stabilize the closed-loop system.

One scenario involved a violent oscillation, called *limit cycling*, in the aerospace vehicle, which broke apart. The Fortran six-degree-of-freedom mathematical model of the vehicle

0740-7459/13/$31.00 © 2013 IEEE

```
O = input * numc1 + Prev_inp * numc2 – O * denc1;
Prev_inp = input;

If O > upperlimit
O = upperlimit;
Elseif O < lowerlimit
O = lowerlimit;
End
```
**(a)**

```
If O > upperlimit
Olimit = upperlimit;
Elseif O < lowerlimit
Olimit = lowerlimit;
Else
Olimit = 0;
End
```
**(b)**

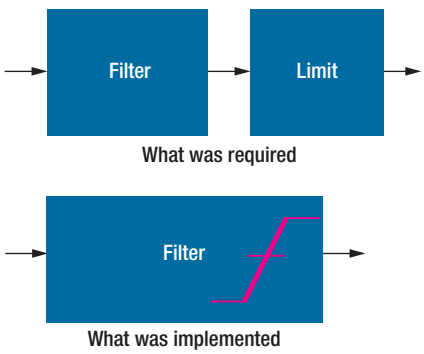



**FIGURE 1.** A filter implementation error. (a) Here, the filter code segment computes the filter output **O** using the variables **numc1**, **numc2**, and **denc1**, which are the digital coefficients. The variable **O** is also used as a state. (b) However, the limit should have used a different variable, **Olimit**, shown here.

showed no oscillation. However, a simulation on the Iron Bird hardware-in-the-loop simulator, using the actual onboard computer, showed these oscillations. We recorded all the signals to and from the controller and compared the Fortran and onboard C code output. We found that a filter implementation error caused the limit cycling.

We attributed the error to reuse of the same variable for two block outputs. The filter was followed by a limiter that restricted the output (see Figure 1). The filter output, represented as the variable **O** in the code, served as a previous value or state in the next computation cycle. **O** also represented the limiter's output. In the next iteration, the filter used the limited value of the state **O**. This caused the limit cycling and system failure.

The Iron Bird simulation didn't trap this error because the testing employed static inputs to verify scaling and connectivity. The closed-loop simulation didn't simulate the actual violent flight conditions, so the limits weren't exercised.

The most interesting part of the whole exercise was that a second, very similar failure occurred after we corrected the filter. We pondered over the wrong diagnosis for some time, until we discovered that a filter in the other control loop was implemented incorrectly. "You didn't tell us there was a problem with this filter, too," said the coders.

I learned four lessons from this experience. First, static tests just check for scaling and connectivity. To test filters, we also needed dynamic tests with specific signals, such as sinusoidal sweeps, steps, and doublets with negative and positive excursions. We should have selected a frequency and amplitude that would make any error observable at the output. Too high a frequency signal or too low an amplitude can't excite a low-pass filter. The filter will remove these components, and the output will be near zero.

Second, any error in the filter should be propagated to the output, which will ensure that the filter computation's effect is observable at the output. What we learned from this incident resulted in the delta model concept used for generating control-law tests for the Indian Light Combat Aircraft (LCA) program nine years later. Designers coded the LCA control law in Fortran for development and test activities. The delta model was Fortran code with the filter algorithm or coefficient perturbed. We changed the third decimal place value by adding 0.001 to the filter coefficient. We considered a test case suitable if the actual model output, when compared with the delta model, brought out the seeded error. We seeded only one error into the delta model at a time.[2]

Third, we extended the delta model to perturb the IEEE 754 floating-point representation of the filter coefficient at the 18-bit position in the mantissa. Figure 2 shows the error at the controller output for various bit perturbations in the filter coefficient in the delta model. This, we felt, was a better representation of error instead of the ad hoc 0.001 value we used earlier. (We learned to be more consistent mathematically.)

Finally, the testing errors, which failed to uncover the limit error, were similar to the case Thomas Huckle mentions on his website in which the break statement wasn't tested. The RTCA DO-178C standard (*Software Considerations in Airborne Systems and Equipment Certification*) insists on complete code coverage, along with other coverage metrics based on the criticality level.[3] But the indicated code coverage doesn't always mean that the object code executable on the target board behaves in the same manner. We'll see this in the next example.

## The Case of the Missing Variable (1998)

In the LCA program, the flight control laws and air-data algorithm are coded in Ada and compiled for an i960 processor using a qualified compiler in optimization mode. In 1998, one build had passed the final tests on Iron Bird. The flight control laws were undergoing non-real-time (NRT) testing[2] on a single-board computer. The air-data system code was new and was

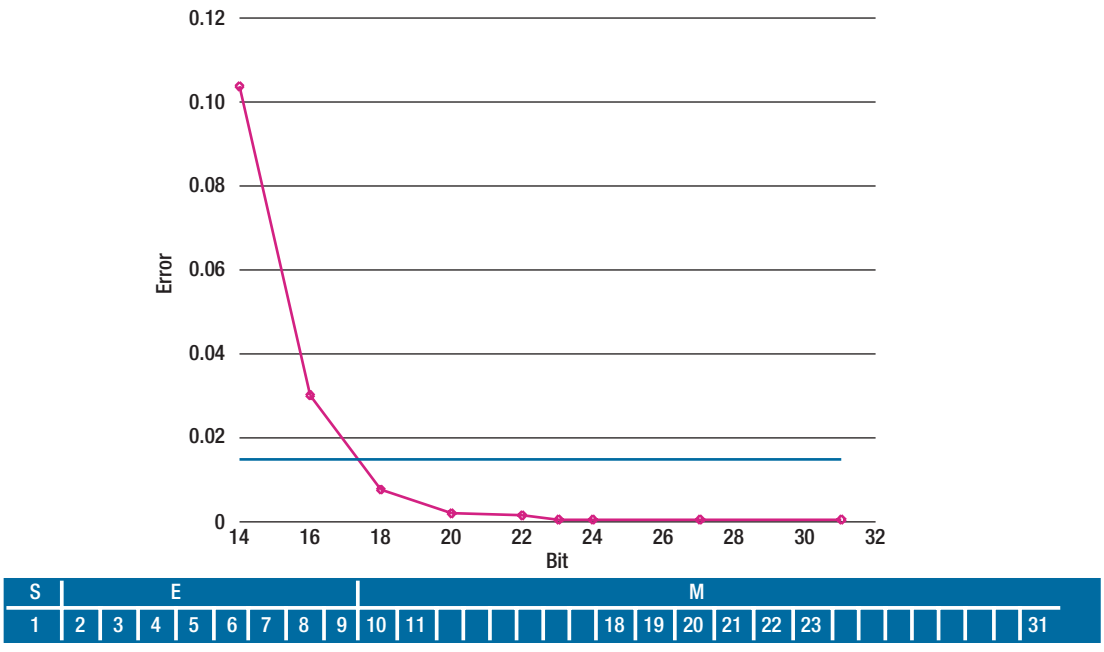



**FIGURE 2.** IEEE 754 implementation of a filter coefficient and the error produced at the output by toggling one bit at the different locations in the mantissa. The lower part of the figure represents the IEEE 754 format, showing the sign bit, exponent, and mantissa.

still being worked on to be tested on a single-board computer. Another group was looking at object code verification. They found a problem in the object code but couldn't figure out the trace to the code. This pressured us to speed up NRT testing for the air-data system. We could isolate the function in which the model and code differed. The error, however, wasn't obvious.

We found that a function was supposed to have two variables—the Mach number and angle of attack—passed in as parameters. In this case, the function was called using the Mach number passed in twice for both the variables: `O = compute_table(Mach_Number, Mach_Number)`.

The optimizing compiler thought the other variable wasn't necessary and removed the code segments associated with the angle of attack. The object code analysis team noticed this, but the code still passed the Iron Bird tests. A successful Iron Bird test means that the digital flight control computer is cleared for flight tests.

Here, I learned three things. First, the compiled code, especially under optimization, differed from the written code. DO-178B requires showing correctness-testing results on the actual hardware with the compiled executable, but showing this at the system level is difficult. Remember, the software had already passed the Iron Bird tests.

Second, a board-level test in NRT mode can reveal many errors in the safety-critical control-law algorithm. Such testing won't reveal timing issues. However, a comparison with a model at every frame with a very low pass/fail threshold value, perhaps around 0.0002, is a powerful tool for control-law tests. This threshold is very low compared to the 1–2 percent of the full-scale range at the system-level tests. Such tests (NRT methods) are mandatory for the LCA program. In 2011, using this NRT test methodology, we cleared four US commercial aircraft flight control laws, which were DO-178B level-A projects (the highest design assurance level).

Finally, you can't reduce the rigor of testing and reviews. There's a tendency to say, "We've tested enough; do we need object code reviews?" In this case, the object code reviews captured the error. The code reviews and unit tests had passed the code. We learned that we should use all available resources to certify safety-critical code.

## A Fader Logic Anomaly (1999)

Often, flight control laws require bringing in one signal and fading out another. Fader logic circuits bring in one signal gradually while freezing and fading out the other signal that could have failed. The fading happens in a

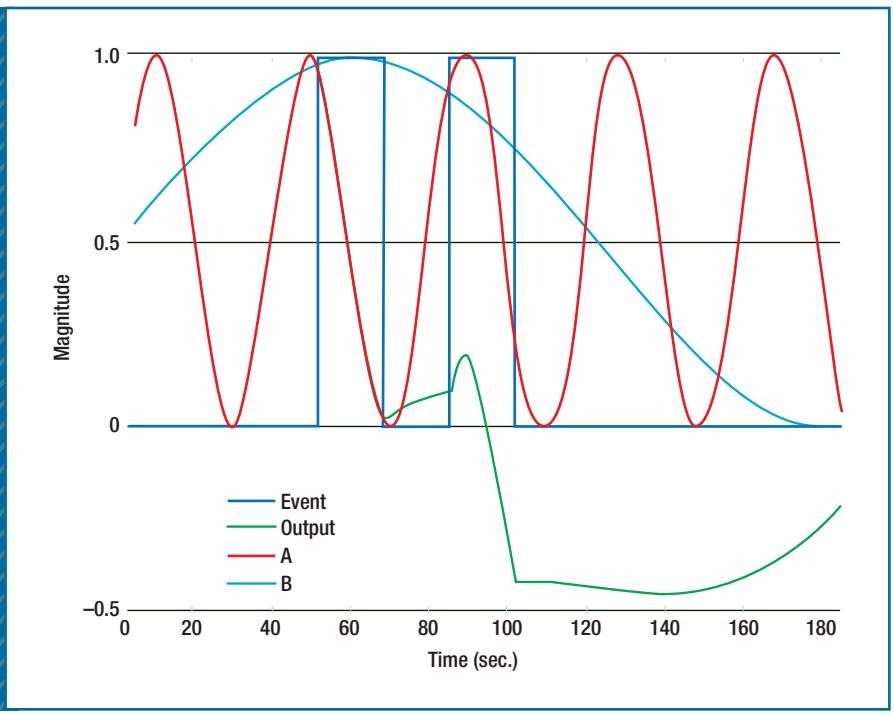



**FIGURE 3.** The transient-free switch behavior that caused slat failure. The two input signals, A and B, are of unit amplitude varying between 0 and 1, but the resulting output is negative.

finite time indicated by the number of frames. For example, signal A should fade to signal B in 100 frames if the switching event is `true`; vice versa for `false`. To do this, we compute the difference between the current output and the signal to be faded. We then divide this difference into equal parts over the fade time. Every frame, the output is added to or subtracted from this small delta until it reaches signal B after 100 frames. This logic works well when A and B are constant.

The story is entirely different when A and B change with time. During LCA control-law verification and validation, we injected random sinusoidal waveforms to one such fader block and randomly toggled the event `true` or `false`. This resulted in unexpected behavior: input signals A and B were of unit amplitude, but the fader output's value was higher (see Figure 3). The design team saw this behavior but said it wasn't a safety issue. ("Do you think it will happen in flight?" the team asked.) Their conclusion was that testing was too rigorous for the whole exercise.

We eventually found that the fader switch caused problems in the gain scheduling. The LCA program uses scheduled filters, with the filter coefficient changing on the basis of altitude and speed. The fader logic changed the filter coefficient's sign, making the filter unstable during the preliminary designer tests. A quick workaround was to limit the filter coefficient value so that the sign didn't change. This worked well for the time being for this specific problem but didn't resolve the issue, as we learned later.

From this experience, I learned three things. First, safety-critical flight control systems require rigorous testing. A single-board computer provides a good platform for random tests. Random testing brings out many hidden issues that system designers might not otherwise consider.

Second, you should be more persistent when demanding a change. In this case, owing to project pressures, changes so late in the program were ruled out. However, I still often see these situations in aerospace projects.

Finally, quick fixes aren't a solution.

## Revisiting the Filter Issue (2001)

The LCA flight control law needed a limiter. LCA flight control-law code is automatically generated in Ada using a qualified Beacon tool. But in this case, management decided to manually code this limiter because updating the Beacon diagrams would mean regenerating the code. Management thought this would have resulted in too much effort for testing and verification. The quick fix here was to just change that Ada code function by simply including `IF ... THEN ... ELSE` to limit the filter output. The code fix took place, but the code reused the same variable to optimize variable use. NRT testing found the error immediately.

Here, I learned two things. First, people keep repeating mistakes. In this case, the coding team was unaware of the similar mistake made earlier in a different laboratory. Both situations involved the same thought process: "I want to save variable use"—a habit drilled into good programmers. But in this case, the control system wasn't just a piece of code; it was a dynamic behavior.

Second, don't look at control system element blocks as just code variables, states, data types, and flowcharts. These blocks are dynamic and change system behavior over time. We must see them as dynamic entities, simulate them, and understand them.

## Fader Logic, Part 2 (2003)

I was monitoring the telemetry station when the pilot announced slat failure.

While attempting to land, he had simultaneously deployed the landing gear, selected the standby gains, and operated the slats. The combination resulted in a negative slat command. The slat command should fall between 0.0 and 1.0; the negative value tripped the monitor, declaring an error. Postflight data analysis showed that the fader logic in the slat command had caused the negative value. The pilot's simultaneous switching actions caused the input to the fader to be dynamic instead of constant. This resulted in another quick fix: don't operate all these switches at the same time. (This reminds me of the quick fix suggested for Therac-25 race condition: do things slowly.)

After a few months, the aircraft, during another test sortie, bounced at touchdown. The nose pitched up, and the pilot had to force the stick down to make the nose wheel touch the ground. He said this had never happened to him before. Postflight data analysis showed that the erratic behavior was due to the fader logic in the command path. At this point, the phenomenon had my name attached to it; the erratic behavior was called the "Yoga syndrome." "You found it, you fix it," said the manager.

I was wearing my designer hat this time instead of my usual verification-and-validation hat. The solution was to use only the constants 1.0 and 0.0 for the fader. The idea was to multiply the faded output by the parameter to be faded. This solution behaved the same way for the constants' inputs but proved safer, without any overshoots, for dynamic inputs. This decision was deliberate—we couldn't change the Beacon tool, and we were wary of any manual changes. It meant a lot of work. We had to use a new set of blocks for all the locations where this fader was used, rerun the simulations, and release a new set of control-law diagrams.

Even after we did all this, the tests failed in NRT mode. The coding team hadn't changed anything! "It works the same way, doesn't it?" they said. "Why did we go through all this trouble, then?," we wondered.

Here, I learned three things. First, what happens on the ground happens in the air. An error can remain dormant for a long time, like a volcano. The resulting eruptions can be catastrophic. We could have easily lost an aircraft.

Second, when looking at requirements, the coding team shouldn't assume anything. When in doubt, ask. The released documents normally won't explain design decisions, but there will be a reason for a change.

Finally, be bold when you encounter an error on the ground. Errors found in ground tests have frequently manifested themselves in system failures. Cite this article, if it helps.

## Fader Revisited (2009)

I was one week into a new job with a new team, and I found the familiar fader logic problem to help me settle into my new job. Finding a familiar logic after 10 years, this time in a US commercial-aircraft program, was, in an odd way, heartening. This logic caused similar problems here but was rectified only when it caused problems in flight tests. We designed a new fader logic, but the coding team failed to implement it. (Does this sound familiar?)

Here, I learned that history repeats itself. The new function was coded but not called in the main code. The coding team overlooked this, and the code passed the code review. NRT tests found this error.

## Delay On/Off (2010)

I've come a long way in my career in testing flight controls, and I'm still surprised at implementation errors. One commercial-aircraft program has **delay on**, **delay off**, and **delay on/off** blocks. A **delay on** block looks for persistence in a failed signal. It generates a **true** output if the input holds **true** for a specified duration—say, two seconds. Its output immediately becomes **false** if its input becomes **false**. A **delay off** block looks at the **false** condition similarly. A **delay on/off** block looks at both **true** and **false** conditions. If the input is **true** for the **delay on** duration, the output becomes **true**. If the input becomes and remains **false** for the **delay off** duration, the output becomes **false**.

In this case, the coding team had implemented the **delay on/off** block as a combination of a **delay on** block feeding its output to a **delay off** block. They tested this and found it worked well. Such a function reuse was considered a good example of optimization and productivity improvement. However, stress testing at the NRT level revealed the difference in the model and code implementations. A **delay on/off** block isn't a combination of **delay on** and **delay off** blocks (www.mathworks.com/matlabcentral/fileexchange/33129-testing-of-safety-critical-control-systems). This was surprising to the systems team.

> What happens on the ground happens in the air. An error can remain dormant for a long time, like a volcano.

Here, I learned three things. First, test the small library functions or elements, as the new Model-Based Development and Verification Supplement

## ABOUT THE AUTHOR

**YOGANANDA JEPPU** is a senior systems specialist at the Moog India Technology Center and a postgraduate student in missile guidance and control at Pune University. His research interests include safety-critical software, control systems, model-based development, and verification and validation. Jeppu received an ME in mechanical engineering from the University of Poona, India. Contact him at yvjeppu@gmail.com.

(DO-331) to DO-178C says.[3] You must test element block functionality with random waveforms in various scenarios to really understand the functionality and behavior.

Second, in safety-critical applications, take terms such as "code optimization" and "reuse" with a grain of salt. Weigh the reuse with the required functionality. Assumptions are dangerous. Testing rigor can't bring out all the defects; take care of this during development. "Does it do what I want?" is a question system engineers must answer very early in the program.

Finally, when coding, also ask, "Am I coding for testability? Are the block outputs available to me at a higher-level test to compare and debug?"

## Where We Are Now

In 2011, we certified five major safety-critical programs per DO-178B Level A, flawlessly passing through audits during the fourth stage of involvement (final review). Did we learn anything new? I believe that testing safety-critical control systems has something new to offer on a regular basis. Sometimes you think you've seen it all, when along come failures that make you reassess what you've been doing.

We're generating an autoreview tool that converts the knowledge gained from all these mistakes into functional metrics for the control system elements. Each control system element block has a set of functional metrics that the input waveform must satisfy during testing. An example is the well-designed test case that brought out the error in the **delay on/off** block. The test case must satisfy four criteria:

- The input waveform holds **true** for longer than the **delay on** time, a good rule of thumb being at least 20 percent longer.
- The input waveform holds **false** for longer than the **delay off** time.
- The input waveform holds **true** for less than the **delay on** time, an approximate rule being 50 percent less.
- The input waveform holds **false** for less than the **delay off** time.

We verified the metrics using mutation-based tests.[4] Test cases must reveal random errors injected into the model and code. This has worked well so far, and we're qualifying the metrics according to DO-178B tool qualification criteria.

We have Simulink with its chain of tools that help us in model-based testing and certification. We have Scade (Safety Critical Application Development Environment) block sets that use formal methods to prove an implementation's correctness. We have numerous test tools that use formal methods, random tests, assertions, and coverage metrics to generate test cases. But all these tools come with a price tag. They can't entirely handle real-world variability and the numerous interactions aircraft have in the real world. We're still far from one-click "certification done."

### References

1. N.G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*, MIT Press, 2012.
2. Y.V. Jeppu, K. Karunakar, and P.S. Subramanyam, *Flight Clearance of Safety Critical Software Using Non Real Time Testing*, tech. report AIAA-2002-5821, Am. Inst. Aeronautics and Astronautics, 2002.
3. *Std. DO-178B/C, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, 2011.
4. C. CU et al., "A New Input-Output Based Model Coverage Paradigm for Control Blocks," *Proc. 2011 IEEE Aerospace Conf.*, IEEE, 2011; doi:10.1109/AERO.2011.5747530.



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

# SCEPYLT: An Information System for Fighting Terrorism

**Jesús Cano and Roberto Hernández**, Universidad Nacional de Educación a Distancia

*// A safety-critical software system called SCEPYLT provides an information solution for a field traditionally not computerized: explosives and their associated risks in handling, storage, transport, and use. SCEPYLT is a model for cooperative distributed systems engineering projects, synchronized over multiple databases. //*

**THE TERRORIST ATTACKS** on 11 September 2001 in the US, 11 March 2004 in Madrid, and 7 July 2005 in London shocked people around the world, waking everyone up to the realization of our vulnerability in the face of such events. Judicial proceedings to investigate and prosecute the perpetrators of the attacks in Europe revealed the alarming fact that a large amount of the explosives used in the bombings had been diverted from commercial use or stolen.[1,2]

The civilian sector uses explosives for mining, extraction, demolition, drilling, and farming. Explosives travel daily via road, rail, sea, and air networks. Their forms include dynamite, detonator capsules, ammunition, sporting powder, and fireworks. European rules on using these materials include security measures required for their design, manufacture, and handling as well as basic guidelines on the harmonization and control of explosives intended for civilian usage. However, terrorist attacks demonstrate that these rules are clearly insufficient in practice.

To attempt to control explosives and introduce precautionary measures, the Interior Ministers of the G6 countries (Spain, France, UK, Italy, Germany, and later, Poland) convened in Sheffield, England, on 5 and 6 July 2004, to discuss, among other issues, the security of European explosives.[3]

In a related effort, the countries provided experts to form a multidisciplinary working group under Spanish coordination that included representatives from the European Commission and industry advisors. We participated as computer experts for the group starting in 2005.

The first challenge was discerning the role new technologies could play in tracking explosives. Our engineering studies in 2006 resulted in the proposal of an economically viable computer-based solution to improve explosive control: SCEPYLT (explosives control to prevent and fight against terrorism) is a distributed computer system that enhances the control of explosives through flexible, standard, and secure information exchange among the G6 countries.

SCEPYLT directly connects the concepts of dependability and security over the Web and distributed services environments under the prism of the CIA triangle: confidentiality, integrity, and availability. Here, we describe SCEPYLT as an engineering framework of cooperative distributed systems of multiple databases synchronized via a service-oriented architecture.[4–6]

## Objective and Process

The goal of designing the project framework focused on analyzing the security of explosives intended for civilian usage and providing the technology needed to exercise effective control over their transport and commercialization. The requirements included the electronic

0740-7459/13/$31.00 © 2013 IEEE

exchange of all information about a shipment of explosives by road, sea, or air:

- authorizations required,
- data on the vehicles involved,
- itineraries,
- entry and exit points into and out of a specific country,
- haulage contractors and drivers, and
- inventories of the batches of explosives.

As added value to the security objective, the working group specified a warning messaging module so the system could produce a rapid alert should an explosives-related incident occur. Examples of such potential incidents include the following:

- a vehicle breakdown that could delay delivery,
- an act of sabotage,
- a robbery,
- the loss of documentation,
- an accidental explosion, and
- known threats.

This requires the explosives industry to code each component or unit of explosive material to allow for traceability, with the system itself providing a help portal as an optional module so that sector companies can directly report information to the appropriate authorities in each country.

State public organizations typically have large computer systems linked to large networks. The civil servants charged with interconnecting these systems and the internal networks themselves can sometimes hinder the deployment of applications, system error detection, and changes in configuration.[7]

## Lessons Learned

Some lessons we learned from the SCE-PYLT project include basic principles for developing international distributed system applications.

**Lesson 1. Sovereign technology.** The challenges of networking a system among various countries also apply to large corporations and companies. Technology is universal, but each entity makes its own choices. Consequently, a transnational development project must bear in mind that each member will either welcome the developed solutions or show reticence based on various internal economic, socio-cultural, organizational, and political factors that correspond to a diversity of interests. Accepting a transnational approach means respecting each member's technological choices and thus accepting the flexibility and extra effort that this requires.

**Lesson 2. The need to share.** Countries, organizations, and institutions jealously guard the data that affects them. Thus, sharing sensitive data in an international context must be done in such a way that prevents any one member from having total control over all the information. Members must have access to data that applies to them, but complete system data should only be obtained via the entire group's collaboration. Therefore, by sharing issues that affect others, no single party can access all the information unless all parties agree.

The challenges of networking a system among various countries also apply to large corporations and companies.

**Lesson 3. Benevolent cooperation.** Partnerships work best when participants cooperate to reach a common solution in a loyal and constructive manner throughout each phase and activity. This means strengthening ties and making the effort to empathize, even outside strictly professional relationships. This benevolence seems to be carried in the backpack of any successful project, but requires a high dose of motivation and subjectivity that's essential when the technical group is large, heterogeneous, and has few face to face meetings.

**Lesson 4. Security.** Regarding security and dependability, we learned that security is the process and not the goal. The explosives working group commissioned a computer expert, the head of the development area of the Civil Guard, to serve as technical director and define a solution that provided a compromise among the political, social, and technological realities. Three fundamental decisions composed the choice of the architecture: the decision to use a completely decentralized mesh network design, using member nodes of the global database, and choosing an information protocol for Web transactions and queries.

## Completely Decentralized Mesh Network Design

Distributed databases consist of a combination of various computer network nodes, distributed physically but forming a unified logical data system: in other words, a global database. A distributed database design includes the very crucial decision of choosing the type of control used to process transactions to the other nodes in the system. If control is shared among all nodes, the

architecture is decentralized or federated. This decision isn't just technical: each shareholder's perception must be considered, such as a sense of proprietary data loss or security concerns.[8]

A completely decentralized distributed system means no coordinator node exists and control of the distribution is shared fairly among all nodes. A centralized control design is easier to develop and administer, but it also means that all information passes through a single node, which thus becomes a critical infrastructure resource and requires additional performance and security measures.[9]

A completely decentralized distributed design assumes that each node manages communication with other nodes and that it possesses sufficient knowledge of the information's location when making a query. Nodes, especially sovereign and benevolent nodes, must participate in accordance with the lessons learned that we outlined for an international distributed system or exercise a degree of compromise among them. In practice, a completely distributed system has the following benefits for each member:

- the system only shares information of interest with the entire group (the "need to share" concept);
- the complete database doesn't exist at one single site because it's a set of distributed databases (nobody can get all the information just for themselves); and
- all nodes are equally important.

Implementing this decentralized vision in a physical network is accomplished through a mesh network topology to form a distributed base. However, a mesh network is more difficult to configure because each node must have an inventory of the other local nodes in the form of a configuration file.

Literature about full mesh networks shows that network administration gets progressively more convoluted when the number of nodes rapidly increases. On the other hand, a node failure doesn't affect the entire network, allowing us to design a system in which all nodes are equally important and none prevails over the others.[9]

In a geographically distant environment, where government networks intervene alongside the Internet, it's logical to use a mesh network representation. This is achieved through control software, whose monitoring is based on the node connection's configuration file and thus involves no additional network infrastructure or wiring expenses.

In addition to the additional security a mesh network offers in the event one of its nodes fails, mesh networks typically avoid bottlenecks because all the nodes have the same comparative roles.

One important consequence of the previous lessons learned is to achieve a suitable decoupling of the nodes. The system can't depend on any one node or group of nodes, so asynchronous communications and service orientation are important technology options.[10]

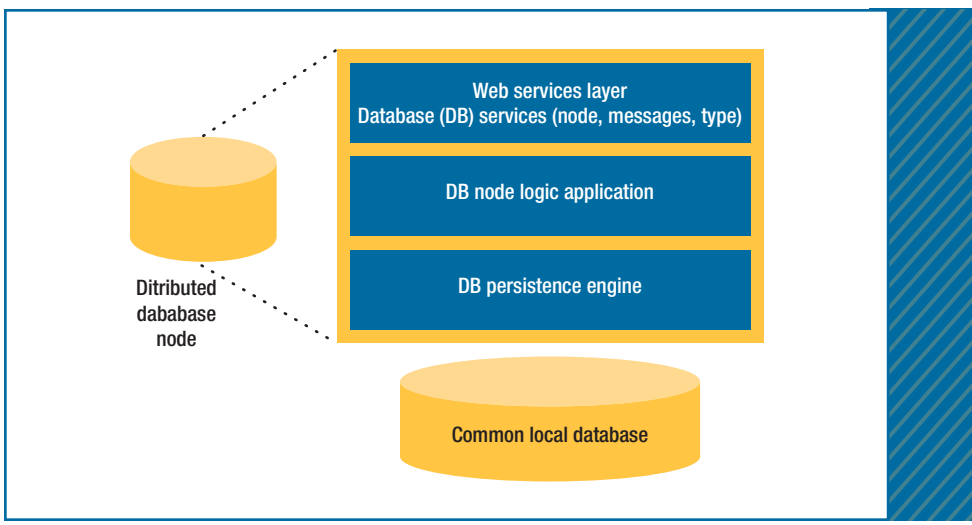



**FIGURE 1.** Software architecture of a distributed node. Each country has a node consisting of a technological infrastructure composed of multiple layers, similar to an onion. The top layer is the interface with other nodes and has a higher agreed engagement with them. The inner layers are much more flexible in case a country needs to switch to other implementations so the repository can adapt to a technologically sovereign choice.

## Member Node of the Global Database

To design a database from an entirely decentralized approach, a level of abstraction for control must be set over the local nodes forming the global distributed database. This control is organized in a multilayer structure:

- a messaging Web services layer,
- a common interface for everyone in the system,
- a business logic layer that processes the global transactions,
- a data persistence engine layer, and
- a local database repository.

This architectural design is sufficiently independent of the technology, which means numerous options exist to implement it ("sovereign technology").

Figure 1 shows how one part of the control, the Web services layer, consists of an initial interface layer connected to

the other nodes by SOAP-based XML messaging, displaying Web services that implement the specific functionally necessary for global database transactions. The essential parameters are the transaction type (type), the node with which communication is established (node), and the content of the transaction (message). This level of control abstraction is completed by the node logic layer, where functional operations are carried out over the base, and the persistence engine layer, which implements the database management system forming the local repository independent of the local nodes' technology.

The Web services-oriented approach provides a series of benefits, such as a better organization of the functionalities a collaborator node offers, lower development costs, and greater flexibility. This is because the service layer is the same for all nodes, which means that users can utilize their own systems by making the adaptations required to meet the services offered. It also makes it easier to put the services to further use and reuse the code. Moreover, top-level security layers can be added, such as XML application-based firewalls.

Both the more external Web services layer and the internal persistence layer allow for an architecture that's sufficiently decoupled from other nodes while respecting different technology options.[11]

### Information about Implemented Web Services

When a node sends information to other nodes, it invokes a Web service. This type of communication is fully synchronous. If any of the receptor nodes are unavailable, it throws a retry cycle and a subsequent recovery mechanism.

Other widely used sublayers of database node logic are also part of the design: DO (domain object), BO (business object), DTO (data transfer object), and SVC (service object). Each type of message requires the implementation of at least four service objects: insert (new or generate), update, delete, and query.

The types correspond functionally to design objects such as explosive transfer, shipment, warning, alert, or document message. Attributes comprise data value, a valid flag, and a descriptive attribute error, if necessary. This principal message is a transfer and thus collaborative by definition: a transfer consists of an itinerary that must be approved or rejected by all participants in the carriage. Thus, two additional status services include accept and reject.

Finally, the message is encapsulated in XML and safely enveloped for sending.

> The purpose of the global database's horizontal partitioning is to enhance transaction and query performance.

### Transactions and Queries

Key design points in processing the physical distribution of data in a distributed network include efficient partitioning and an optimally designed network structure (combining the "need to share" lesson and a suitable consequence of decoupling). For example, suppose that a factory in Germany wishes to transport a batch of explosives to Italy on a truck (road transport), via France. Because the transport involves three countries, the complete information must be stored in the global distributed database. A first correct approach is that each country should store its own data. This is the case with the legal data authorizing the right to drive through a country, the security measures to be implemented by the haulage contractor, the data on the drivers of the vehicles carrying the explosives, and the public servants dealing with the application. All this information makes up a functional criterion for setting the database distribution.

Partitioning has a considerable influence on the performance and administration of the database. Vertical partitioning divides a relational entity into various subsets of columns or attributes, while horizontal partitioning consists of obtaining a set of relational fragments, each of which has a subset of rows or tuples.[12] In our example, if the security measures are taken into account, Germany would have a tuple to represent its own security measures, France another tuple for its measures, and Italy another.[13]

The purpose of the global database's horizontal partitioning is to enhance transaction and query performance. Partitioning optimization results from studying the queries in an adequate test set. In the proposed example, any queries regarding the transfer of explosives must have access to other nodes via the Web. That is, to know the details of an authorization for an explosives truck, three different local database accesses must be made and the data exchanged via the international network. The proposed optimization consists of making redundant the information related to transports that are susceptible to sharing (the "need to share"). So, for the specific transport in the example of the German node, France and Italy might keep records of that transfer concerning authorizations, limitations, and truck drivers. But there

will be no unnecessary duplication of information. For example, the public servants in charge of the processing don't leave the local node. Thus, query performance is optimized. To access the transport data, each node involved will have to make a transaction to its own local database. The data distribution and the redundancy are therefore irregular.

Geographic location comprised the principal criterion of our system so that information semantically related to a node is hosted in its own repository. Thus, users can find tuples that directly affect them. When semantics include information from two different member nodes, queries regarding this relationship require a request via a data network. The same is true for relationships among databases whose tuples include identification semantics from more than one member node. This shuffling around of information on the network can seriously hinder performance.

We therefore adopted the criterion of inserting the same row in all member nodes identified in a relationship to add a horizontal redundancy to improve queries in exchange for a slight increase in the transactions of insertion, updating, and deleting. One example is the "itinerary" relationship, in which the semantics indicate geographical sections between two departure and arrival points. However, to optimize the itinerary queries, each local relationship will contain not just the tuples of sections in which it's included but also any others that are semantically related.

## Consolidation Phase

In 2007, the technical director presented the SCEPYLT solution to the explosives working group, and the group widely accepted and supported it. Previously, a subset of this group formed for each country's technologists to provide feedback. The proposal essentially consisted of creating a global database among the six countries, where each would have all the data available on explosives transferred from one country to another. This would be achieved by distant geographic nodes comprising properly synchronized multiple local databases. In addition, each country would either host the information directly affecting it or participate as part of the itinerary of some form of transport—that is, the country would become a fragment of that global database. There would thus be a redundancy of information with deliberately set functional criteria to allow a country a degree of autonomy in managing basic consultations without the need for the other nodes to be active.

The presented proposal addressed all the concerns of European directive 93/15/EEC, which urged countries to set up data networks to exchange information on explosives. However, it wasn't just a question of exchanging information on transport authorizations. The system also allows traceability of shipments, which led to a detailed and exhaustive control of explosives in real time. This traceability required harmonizing the rules of the European countries with a Directive in 2008 and, recently, in 2012 (that is, 2008/43/CE and 2012/4/UE, respectively).

SCEPYLT defined software under a Web services-oriented Java Enterprise architecture with an XML information exchange via semantic protocols, an intensive use of design patterns, and good development practices. It also included a definition of the demanding levels of computer security, scalability for adding nodes/countries, an XML semantic communication protocol, and an e-government approach. The system had to enable sector companies to participate to help speed up bureaucratic procedures and cooperate with traceability security, public-key cryptographic technology support, and the use of technology specifically designed for this project.

The costs of SCEPYLT were defrayed by a European subsidy from the Programme for Police and Judicial Cooperation in Criminal Matters (AGIS).

## Best Practices

Bearing in mind the conclusions of the group's meetings, and the experiences during the development period, which took about six months, we defined four best practices.

**Invite individuals and companies to participate.** With a focus on e-government, citizens and businesses can provide a significant source of collaboration due to Internet globalization. They can share experiences working with public administrations to provide more efficient services. All of this contributes to a synergy of good governance.

> This shuffling around of information on the network can seriously hinder performance.

**Give communications top security.** An extension of the CIA triangle handles security protection in three stages:

- attaching a physical tier based on privacy tunnelling between networks,
- obtaining channels based on the

use of end-to-end host encryption, and
- ensuring message delivery based on the use of an XML digital signature.

The most notable is that confidentiality is a double encryption between networks, and there is always a point to point encryption overlying.

**Ensure transparency in IT.** Technical information must be accessible to stakeholders—including each country's technical authorities, operators, and developers—along with an easy-to-understand methodology with well-defined deliverables and well-organized source code. We learned that you have to bring the pillars of open government to IT management, which include transparency, collaboration, and participation.

**Internationalize ontology, encoding, and user interface.** The language-dependent issues or implementation decisions about encoding require applying a set of best practices that have a bearing on project management and product quality. Because we're dealing with message exchange among nodes in SCEPYLT, a message exchange protocol needed to exist that uses a common ontology and a common vocabulary to ensure system development and subsequent maintenance.

A message exchange protocol must exist that uses a common ontology and a common vocabulary.

## Moving Forward

Taking into account the aforementioned objectives during the development period, the project used an iterative, incremental, process-oriented methodology that combined flexible and agile technical methods with features of other more formalized methodologies. We planned and designed three software iterations or pilots and closely watched them to ensure that rigid time constraints were met, which was key because few European working group meetings were held (usually only two a year).

To guarantee secure communications, the system must use encoding and authentication to ensure countries' trust—examples include widely recognized protocols such as SSL/TLS and digital certificates. Isolated Internet networks were preferable, such as sTESTA, which is the common European intranet for all EU member states, or lacking that, virtual private networks to tunnel national networks.

The EU chose to use SCEPYLT and subsidized the expansion project through a specific program of the European Commission's Directorate General of Justice, Freedom and Security, known as "Prevention, Preparedness and Consequence Management of Terrorism and other Security Related Risks."[14]

SCEPYLT was the first information system specifically oriented to the control of explosives against terrorism within the framework of measures to improve public safety in Europe. Other initiatives have since followed SCEPYLT, such as the "European Bomb Data System" and the early warning system for policing purposes under the auspices of Europol.[15]

The project continues to develop and serve as the driving force for a wider study:

- Through an initiative of the European Commission's observers in the working group (from the Directorate-General of Enterprise and Industry and the Directorate-General of Justice, Freedom and Security), the project now extends to the 27 countries of the EU.
- The SCEPYLT platform is the reference system for the control and exchange of information on explosives in Europe.
- The project has aroused the interest of companies in enhancing public security and encouraged them to participate by providing more details of commercial batches of explosives.
- The project has also given rise to regulatory changes, including Commission Directives 2008/43/EC and 2012/4/EC, regarding setting up a system for the identification and traceability of explosives for civil uses. Use of the system became mandatory on 5 April 2013.

Some lines of future research and development include

- availability control and the optimization of decentralized control management,
- fault-tolerance system backups, fragmented backups from nodes, and full nodal recovery based on the information in the distributed system,
- application of social networks as actors for the system, and
- devices for geolocation of explosive materials and traceability.

At the international policy level, the UN should promote the harmonization of laws in areas that affect

## ABOUT THE AUTHORS

**JESÚS CANO** is a part-time professor in the School of Computer Science at the Universidad Nacional de Educación a Distancia (UNED) and an IT practitioner of the Spanish Civil Guard. His research interests include technology for citizens, e-government, e-democracy, smart cities, IoT, safety-critical infrastructures, distributed systems, and software architectures, especially concerning security and cryptography. Cano has an MS in communications, networks, and content management from UNED. He is an IEEE member and also belongs to the Education, Communication, and Computer societies. Contact him at jcano@scc.uned.es.

**ROBERTO HERNÁNDEZ** is a professor in the Control and Communication Systems Department and the dean-director of the School of Computer Science at the Universidad Nacional de Educación a Distancia (UNED). He has coauthored more than 60 publications in international journals and conferences on his research interests, including quality-of-service support in distributed systems and development of infrastructures for e-learning. Hernandez has a PhD in science from UNED. Contact him at roberto@scc.uned.es.



global security and traceability such as information-sharing strategies that are technology friendly, benevolent, safe, and economically viable. Cooperation among the private and public sectors should be reviewed to improve information flows, and agencies at the national, regional, and local levels should promote best practices and remove barriers.

### References

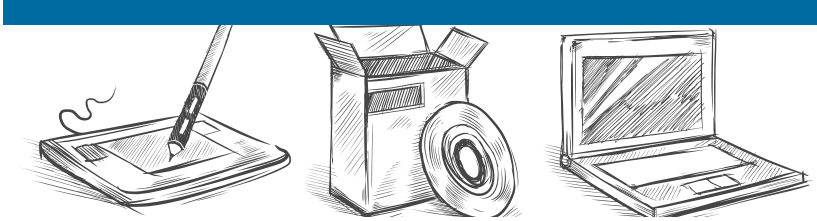
1. G. Tremlett and I. Black, "Explosives Theft Linked to Madrid Bomb," *The Guardian*, 22 March 2004; www.guardian.co.uk/world/2004/mar/23/spain.gilestremlett.
2. Agence France-Presse, "Terrorism Police Are Investigating Theft of Explosives in France," *New York Times*, 19 July 2008; www.nytimes.com/2008/07/19/world/europe/19lyon.html.
3. House of Lords, European Union Committee, "After Madrid: The EU's Response to Terrorism," Mar. 2005; www.publications.parliament.uk/pa/ld200405/ldselect/ldeucom/53/53.pdf.
4. J.J. Wylie et al., "Survivable Information Storage Systems," *Computer*, vol. 33, no. 8, 2000, pp. 61–68.
5. J.C. Knight, "Safety Critical Systems: Challenges and Directions," *Proc. 24th Int'l Conf. Software Engineering* (ICSE02), IEEE CS, 2002, pp. 547–550.
6. A. Avizienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, 2004, pp. 11–33.
7. W.R. Dunn, "Designing Safety-critical Computer Systems," *Computer*, vol. 36, no. 11, 2003, pp. 40–46.
8. B. Friedman, P.H. Kahn, and D.C. Howe, "Trust Online," *Comm. ACM*, vol. 43, no. 12, 2000, pp. 34–40.
9. R.J. Dunn et al., "Presence-based Availability and P2P Systems," *IEEE 5th Ann. Int'l Conf. Peer-to-Peer Computing*, IEEE CS, 2005, pp. 209–216.
10. M.P. Papazoglou et al., "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, 2007, pp. 38–45.
11. N. Milanovic and M. Malek, "Current Solutions for Web Service Composition," *IEEE Internet Computing*, vol. 8, no. 6, 2004, pp. 51–59.
12. S. Agrawal, V. Narasayya, and B. Yang, "Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design," *Proc. Int'l Conf. on Management of Data* (Sigmod 04), ACM, 2004, pp. 359–370; doi:10.1145/1007568.1007609.
13. M. Kantarcioglu and C. Clifton, "Privacy-preserving Distributed Mining of Association Rules on Horizontally Partitioned Data," *IEEE Trans. Knowledge and Data Eng.*, vol. 16, no.9, 2004, pp. 1026–1037.
14. European Federation of Explosives Engineers (EFEE), "The EU Directives Committee: Enhancing the Security of Explosives," *EFEE Newsletter*, Mar. 2009; http://efee.eu/wp-content/uploads/2012/03/2009-03-EFEE-Newsletter.pdf.
15. Council of the European Union, "Council Conclusions on Systems and Mechanisms for the Enhancement of Security of Explosives," 2010; www.consilium.europa.eu/uedocs/cms_data/docs/pressdata/en/jha/114017.pdf.

FEATURE: SOFTWARE TOOLS

# Software Sketchifying: Bringing Innovation into Software Development

**Željko Obrenović**, Software Improvement Group

// Software sketchifying is a software development activity that stimulates spending more time generating and considering alternative ideas before making a decision to proceed with engineering. It's supported by Sketchlet, a flexible tool that empowers both engineers and nonengineers to work with emerging technologies and explore their possibilities. //



**HENRY FORD'S ASSEMBLY-LINE** production of the Model T inspired changes in the automotive industry, and the software industry has made numerous attempts to apply similar ideas (for example, see the chapter, "Will the Real Henry Ford of Software Please Stand Up" in Robert L. Glass's book).[1] While the assembly-line philosophy is well known, Ford's approach to innovation and the process that preceded the Model T's production is less so. Between 1892 and the formation of the Ford Motor Company in 1903, while working mostly for the Edison Illuminating Company, Ford built about 25 cars. In the five years after the company's formation, he built and sold eight models—Models A, B, C, F, K, N, R, and S— before settling on the Model T. He tested prototypes labeled with the 11 missing letters. Ford summed up this experience this way:[2]

> I do not believe in starting to make until I have discovered the best possible thing. This, of course, does not mean that a product should never be changed, but I think that it will be found more economical in the end not even to try to produce an article until you have fully satisfied yourself that utility, design, and material are the best. If your researches do not give you that confidence, then keep right on searching until you find confidence.... I spent twelve years before I had a Model T that suited me. We did not attempt to go into real production until we had a real product.

Today's automotive industry has changed significantly since Ford's initial success, but some of his philosophy behind innovation still remains. For example, Toyota's "nemawashi" principle states that decisions should be implemented rapidly but made slowly, by consensus, and after considering all options.[3] Bill Buxton, who studied innovation in the automotive industry, noted that a new car's design phase starts with a broad exploration that culminates in the construction of a full-size clay model and costs over a quarter of a million dollars.[4] Only after bringing the new concept to a high level of fidelity in terms of its form, business plan, and engineering plan does a proj-



0740-7459/13/$31.00 © 2013 IEEE

ect get a "green light." After that, it typically takes a year of engineering before the project can go into production.

Inspired by general ideas about how the automotive industry brings innovation into manufacturing, I developed software sketchifying as an activity to stimulate and support software stakeholders to spend more time generating and considering alternative ideas before making a decision to proceed with engineering. My view on software sketchifying combines general ideas of sketching[4] and creativity support tools[5] with several existing software engineering approaches. To support and explore this view, I developed Sketchlet (http://sketchlet.sourceforge.net), a flexible, Java-based tool that empowers engineers and nonengineers to work with emerging software and hardware technologies, explore their possibilities, and create working examples—called *sketchlets*—that incorporate these emerging technologies.

## Product Innovation and Software Engineering

Contrary to the automotive industry, the software industry has a rich history of engineering wrong products. Ill-defined system requirements and poor communication with users remain top factors that influence software project failures.[6] Frederick Brooks also noted that the hardest single part of building a software system is deciding precisely what to build.[7] He proposed rapid system prototyping and iterative requirements specification as a way to solve this problem. Many existing software engineering methodologies, including the Rational Unified Process, Extreme Programming, and other agile software development frameworks follow iterative and incremental approaches.

However, these approaches have limitations when it comes to true innovation. Although prototyping can let us cheaply represent and test our ideas, and iterative and incremental development can help further refine our ideas based on frequent user feedback, neither approach directly supports the generation of new product ideas, nor do they encourage the consideration of alternatives.

Buxton went further in his critique of the innovation capacity of iterative, incremental software development, seeing no comparison between software product design and the development of new automobiles.[4] He argued that innovative software projects need at least a distinct design phase followed by a clear green-light process before proceeding to product engineering. He saw design and engineering as different activities that employ different processes and for which people suited to one are typically not suited for the other.

## Software Sketchifying

I built on Buxton's suggestion by introducing software sketchifying into software product development as a complement to prototyping and engineering. The sidebar presents a sketchifying example scenario of how it might work in developing software systems for an automobile.

### Software Sketchifying Approach

One key characteristic of this approach is postponing the main development activity for the benefit of free exploration, following a main principle of creativity: to generate a good idea, you must generate multiple ideas and then dispose of the bad ones.[1,4] Another key characteristic is stimulating early involvement of nonengineers. Such users often have expertise that's important for understanding customers and their needs. More specifically, the example scenario in the sidebar illustrates several points about software sketchifying:

- The designer's main activity is exploration, learning about a problem and potential solutions and answering a question about what to build.
- Such explorative activity is heuristic, creative, and based on trial and error, rather than incremental and iterative. The designer generates several ideas, most of which will be rejected. However, this process yields important lessons and stimulates generation of novel ideas. These lessons and ideas are the activity's main outcome.
- The exploration activity is not accidental, but disciplined and systematic.
- The exploration is holistic, enabling designers to reflect on relations among user issues, software and hardware possibilities, and the overall dynamics of human-computer interaction. The ideas in the example scenario are influenced not only by software but also by human factors and problems related to car mechanics and equipment.
- The exploration enables early user

> Neither prototyping nor incremental development directly support the generation of new product ideas.

## A SKETCHIFYING SCENARIO

Consider an example scenario with Mirko, an interaction designer at a company that builds software for new generations of cars with advanced sensing and display technologies. Mirko has recently joined the company to explore ideas for software applications that exploit novel opportunities, such as using data from a car radar, GPS sensors, and links to Web services.

Mirko's first task is to explore two applications: a system for warning about the proximity of other cars and a system for presenting news in idle situations, such as waiting for a traffic light. Mirko isn't a programmer, nor is he familiar with all the technical possibilities of modern cars, but he uses a design environment through which he can access and explore software services and components related to his task without serious programming.

To understand what's possible, Mirko first talks with several of his company's engineers. They advise him to start by using a car simulator, which provides a realistic but safe environment to learn about new automotive technologies. One engineer also writes a small adapter that connects the car simulator logger to Mirko's design tool. This adaptation gives Mirko immediate access through a simple spreadsheet-like interface to the simulator data—such as the car's speed and its distance from the car in front of it.

Mirko starts a design environment on his laptop and connects it to the simulator. After becoming familiar with the simulator's possibilities, he turns to his laptop to create a few *sketchlets*, which are simple interactive pieces of software.

### PROXIMITY WARNING SYSTEM

To explore the options for implementing a proximity warning system, Mirko first considers three presentation modes: graphical, audio, and haptic (vibration). For graphical presentation, he uses an editor in his design environment and creates several simple drawings. Then he opens the properties panel and connects the variables from the car simulator to the graphical properties of drawn regions. For example, he creates a sketchlet in which an image's transparency dynamically changes as a function of the distance from the car in front of the driver. He then experiments with other graphical properties, such as image size, position, or orientation. He returns to the simulator and tries each alternative. He also invites a few colleagues to try out and comment on his ideas.

After exploring graphical options, he proceeds to create audio sketchlets. He first tries a MIDI-generator service and connects data coming from the sensor to MIDI note parameters, such as pitch or tone duration. He also experiments with a text-to-speech service, generating speech based on the conditions derived from car data. Finally, he explores using an MP3 player with prerecorded sounds. He then goes back to the simulator and tries these alternatives.

Mirko also wants to try a vibration modality to present navigation information, which the simulator doesn't support. He decides to use a simple trick, starting an application on his mobile phone that lets his design environment control the phone's resources, including its vibrator. Using gaffer tape, he fixes the mobile phone to the steering wheel and creates several sketchlets that map the distance from the car in front of him to vibration patterns. Marko knows it's not a very elegant solution, but it lets him explore basic opportunities of this modality with available resources and little work.

### NEWS PRESENTATION

Mirko also plays with some other options related to the application for presenting news. He starts a Google news service in his design environment and creates a simple page that presents an HTML output of the news service. He then creates a condition for the page's visibility so that the news appears as an overlay on part of the windshield, but only when the car's speed is zero and the automobile is not in gear. He also experiments with speech services that let a user set a news search query by speech.

After finishing his work in the lab, Mirko decides to collect some real-world experiences and try some of his more promising sketchlets in a real car. With help from engineers who are working on testing cars, Mirko gets an extension of his design environment that uses a Bluetooth connection to a test car's onboard diagnostic (OBD) system. With this addition, Mirko creates a simple setting using his smartphone as a presentation device, positioned under a windshield. He connects the smartphone to his laptop, which uses a simple remote desktop client to capture a part of a screen from his laptop. On the laptop, Mirko is running the sketchlets that he created in the lab and that are now connected to the car's OBD system. He asks a colleague to drive the car while he observes a situation and videorecords a whole session for later analysis.

During the process, Mirko constantly interacts with other stakeholders, regularly presents his findings, and lets clients and colleagues try out some of his sketchlets. In this way, Mirko is helping develop new products by providing realistic and tested ideas before and outside the main development activity.

involvement through simple but functional pieces of software in the form of sketchlets.
- Working with real systems, such as the car simulator, car diagnostic system, and Web services, lets a designer learn about the possibilities and limitations of software technologies and create conceptual proposals that are more realistic.

Designers generally aren't engineers who can program and extend their design environments. However, they're part of a broader community of people who can help them learn and extend the exploration space on an ad hoc basis. Sketchifying supports this interaction without taking too much time, thereby empowering nonengineers to explore emerging technologies and to test their ideas without additional help from developers.

### Software Sketchifying Tools

To support and explore this approach, Sketchlet combines elements from traditional sketching, software hacking, opportunistic software development, and end-user development. Sketchlet builds on the results of the Sketchify project (http://sketchify.sourceforge.net), which explored possibilities to improve early design stages and education of interaction designers.[8]

Sketchlet has two main roles:

- to enable designers to create a number of simple pieces of software—sketchlets—as a way to quickly and cheaply explore software and hardware technologies and their potential applications, and
- to support involvement of software engineers in short, ad hoc sessions that give designers realistic pieces of technologies that might be useful for design exploration.

Sketchlet lets designers interact directly with software and hardware technologies through a simple, intuitive user interface. To simplify the integration with these technologies, Sketchlet combines ideas from opportunistic software development with techniques used by hacking and mashup communities.[9,10] A full description of Sketchlet is out of scope for this article. Two appendices containing more detail about how Sketchlet implements the sidebar's example scenario are available online at http://doi.ieeecomputersociety.org/10.1109/MS.2012.71. I also encourage readers to download and try the tool for themselves.

> Sketchlet combines elements from traditional sketching, software hacking, and opportunistic software development.

## Initial Sketchlet Applications and Results

I've developed and applied the ideas about software sketchifying in three projects that featured collaboration among software engineers, interaction designers, and researchers. In these projects, interaction designers and researchers were primarily responsible for creating and evaluating novel conceptual proposals and ideas:

- *Connect & Drive project* (www.tue.nl/en/university/departments/industrial-design/research/research-programs/user-centered-engineering/projects/explorations-in-interactions/connect-drive). Several researchers used Sketchlet to explore options for developing software systems for cooperative adaptive cruise control systems in cars, based on Wi-Fi communication between vehicles and road infrastructure.
- *Persuasive Technology, Allocation of Control, and Social Values project* (http://hti.ieis.tue.nl/node/3344). Sketchlet played a similar role as it did in the Connect & Drive project, helping researchers investigate software products for developing persuasive technologies that encourage people to hand over control to intelligent automation of cars.
- *Repar project* (Resolving the Paradox; www.repar-project.com). Sketchlet was one of the flexible prototyping tools in user-centered design processes, allowing designers to create and evaluate (ill-defined) product concepts early in the development.

Although Sketchlet is still in early development, the approach and tool showed several positive effects in these projects. First, it broadened the opportunities to constructively involve nonengineers, including interaction designers, psychologists, and students. Our tools empowered nonengineers to easily explore relevant technologies and to independently create and test their ideas. The companies involved benefited from their nonengineering expertise and knowledge early in the design process.

Sketchlet also promoted different collaboration between engineers and nonengineer designers. Prior to using Sketchlet, most of the companies followed the approach of making designers responsible for creating a conceptual proposal, which they gave to developers for implementation with

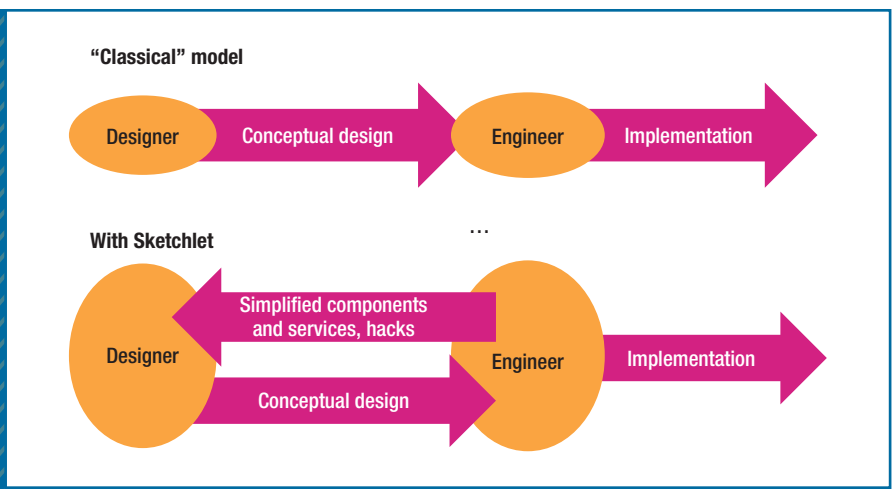



**FIGURE 1.** Comparing the classical design-engineering interaction with sketchifying. With sketchifying, supported by tools like Sketchlet, the interaction between designers and engineers can work in two ways, allowing engineers to give designers early access to simplified versions of software components and services that the engineers might use later in the implementation.

little interaction, except to clarify their designs. With Sketchlet, the interaction between designers and engineers could work in two ways, with engineers giving designers simplified versions of software components and services—early in the design process—that the engineers might use later in the implementation (see Figure 1).

The connected services, although simplified, resemble real components, and sketchlets expressed in terms of these services come closer to the implementation platform that the engineers will use. This change addressed one problem that many companies experience when designers and engineers need to work together—namely, the engineers perceive designers' ideas as unrealistic, too distant from available technology, and not precise enough to be useful. Through the exploration of these services, designers can develop more realistic expectations about the possibilities and limitations of technologies, and incorporate this understanding into design proposals.

Lastly, Sketchlet influenced the mindset of companies toward more and broader explorations early in the software design. Sketchlet helped illustrate the potential of such exploration and inspire the companies to think how other tools could be used in a similar explorative way.

## Sketchifying Benefits and Relation to Other Approaches

Software sketchifying can help better define product requirements so that the subsequent engineering process has a clear focus and goal. It promotes direct exploration of emerging technologies and creation of working examples of simple pieces of software with these technologies as a way to identify potential problems and provoke reactions from users as early as possible. The tool shows the effects of design decisions on user experience and supports user testing before actual development starts.

Exploring the possibilities and limitations of technologies early in the design helps identify a number of problems or user issues before investing in a significant development effort. Discovering such problems later in the process could require changes and additional effort. Early discovery is particularly important in projects using emerging technologies, which have many unknowns—including how well users will accept them.

Promoting the constructive involvement of nonengineers in the design process opens the door to help from experts in fields such as human psychology, which in turn reduces the burden on developers. Moreover, as Glass noted,[1] users who understand the application problem to be solved are often more likely to produce innovation than computer technologists, who understand only the computing problem to be solved. The sketchifying approach requires occasional involvement of developers, but it aims to incorporate them in short ad hoc sessions, and the intent is to empower nonengineers to explore further without developers' help. Once the developer adapts some technology for Sketchlet, nonengineers can work with this technology through a simple end-user interface that does not require technical expertise or programming knowledge.

### Relation to Prototyping and Engineering

Software sketchifying complements existing prototyping and engineering approaches by its focus on free exploration and a trial-and-error approach versus a more iterative, incremental approach of prototyping and engineering (see Figure 2).

Sketchifying supports users in constructing a novel idea and enables nonengineers to actively contribute. This brings software design closer to the practice of other engineering disciplines, in which the design phase precedes the main engineering activity, and designers (usually nonengineers) are encouraged to freely explore ideas before consolidating a few of them for

further development. For instance, it's not unusual for an industrial designer to generate 30 or more sketches a day in the early stages of design, each possibly exploring a different concept.[4]

Software sketchifying precedes prototyping, which tests, compares, and further develops aspects of selected ideas. With a prototype in place, the development can take an evolutionary approach. Prototyping should assess whether selected ideas are feasible and should help decide whether to proceed with engineering. Prototyping aims at making an idea more detailed and concrete, rather than coming up with radically new ideas. Engineering turns the winning idea into a robust and usable product.

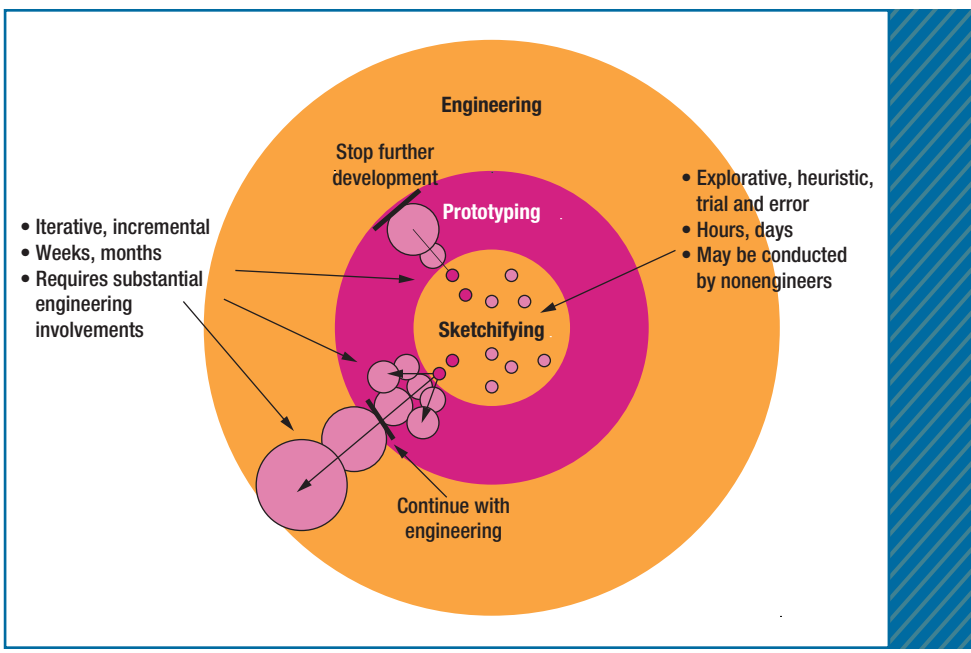



**FIGURE 2.** An idealized representation of relationships among sketchifying, prototyping, and engineering. Sketchifying supports users in constructing a novel idea. It precedes prototyping, which tests, compares, and further develops aspects of selected ideas. Engineering turns the winning idea into a robust and usable product.

### Relation to Other Software Tools

In principle, tools other than Sketchlet could implement the sketchifying idea. However, many current tools can't fully support it because they're not optimized for free exploration and involvement of nonengineers. For example, we could use standard programming languages, such as Java, C#, C++, or programming tools oriented toward interaction design such as Flash and Processing to implement our example scenario. However, programming requires significant expertise, time, and effort—an investment that's simply too high for the intended purpose of generating new ideas and exploring possibilities.

Existing low-fidelity prototyping environments provide ways to quickly create prototypes with inputs taken from external services or sensors.[11,12] These environments might be excellent choices for exploring interactions in various domains. The problems I'm addressing cross these domains and require a variety of sensory inputs and links to diverse software services as well as additional components specific to the companies I'm working with. In addition, such tools often require too much precise specification, partly because they're primarily developed for advanced prototyping rather than for free and broad exploration.

Electronic sketching systems are another promising direction for design tools, enabling designers to create interactive systems with ease using intuitive and natural pen gestures.[13] From the viewpoint of my example scenario, these systems have the drawback of being specialized for specific domains and used successfully only in inherently graphical domains that have a stable and well-known set of primitives, such as 2D and 3D graphics or websites.

Another alternative is to use simple freehand drawings and techniques such as screen prototyping. Such techniques can help in exploring a solution's graphical elements. However, they can describe overall system interactions, such as sensing device inputs and user response dynamics, only in very abstract terms. Moreover, paper sketching doesn't let users explore the possibilities and limitations of emerging technologies. Direct exploration of such technologies yields more concrete ideas about how to best employ them.

Sketchlet borrows ideas from existing solutions, while trying to overcome some of their limitations. I also see it as a complement to existing tools, rather than a replacement. On several occasions, designers have used Sketchlet in conjunction with other tools. For example, some of our users employed Max MSP for signal processing and audio effects and Sketchlet for connections to sensor devices and visualization.

My initial experiences with applying software sketchifying are encouraging. However, an important limitation of this approach is that it requires significant changes of current development culture

## ABOUT THE AUTHOR

**ŽELJKO OBRENOVIĆ** is a technical consultant at Software Improvement Group, Amsterdam. He did the work reported here while working as an assistant professor in Eindhoven University of Technology's Department of Industrial Design. His professional interests include, software engineering, design of interactive systems, end-user development, rapid prototyping, creativity support tools, and universal accessibility. Obrenović received a PhD in computer sciences from the University of Belgrade. Contact him at obren@acm.org.

in its emphasis on postponing the start of development to benefit free exploration, more active involvement of nonengineers and end users, and new forms of interaction between engineers and nonengineers prior to the main development activity. Such changes, in my experience, aren't easy to achieve, but without them, the sketchifying tools are less effective and tend to be used in a limited way.

In future work, I plan to develop a more general approach toward building software services and components so that each service could have two sets of APIs: one engineering API with full functionality, and one sketchifying API that would represent a simplified, limited sample of the full functionality. I also plan to address collaboration because the current implementation primarily supports individual use and is of limited value in collaborative design sessions.

### References

1. R.L. Glass, *Software Creativity 2.0*, developer.* Books, 2006.
2. J. Grudin, "Travel Back in Time: Design Methods of Two Billionaire Industrialists," *ACM Interactions*, vol. 15, no. 3, 2008, pp. 30–33.
3. J. Liker, *The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer*, McGraw-Hill, 2004.
4. B. Buxton, *Sketching User Experiences: Getting the Design Right and the Right Design*, Morgan Kaufmann, 2007.
5. B. Shneiderman, "Creativity Support Tools: Accelerating Discovery and Innovation," *Comm. ACM*, vol. 50, no. 12, 2007, pp. 20–32.
6. R.N. Charette, "Why Software Fails," *IEEE Spectrum*, vol. 42, no. 9, 2005, pp. 42–49.
7. F. Brooks, "No Silver Bullet—Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, 1987, pp. 10–19.
8. Ž. Obrenović and J.B. Martens, "Sketching Interactive Systems with Sketchify," *ACM Trans. Computer-Human Interaction*, vol. 18, no. 1, 2011, article 4.
9. B. Hartmann, S. Doorley, and S.R. Klemmer, "Hacking, Mashing, Gluing: Understanding Opportunistic Design," *IEEE Pervasive Computing*, vol. 7, no. 3, 2009, pp. 46–54.
10. Ž. Obrenović, D. Gaševic, and A. Eliëns, "Stimulating Creativity through Opportunistic Software Development," *IEEE Software*, vol. 25, no. 6, 2008, pp. 64–70.
11. M. Rettig, "Prototyping for Tiny Fingers," *Comm. ACM*, vol. 37, no. 4, 1994, pp. 21–27.
12. Y.K. Lim, E. Stolterman, and J. Tenenberg, "The Anatomy of Prototypes: Prototypes as Filters, Prototypes as Manifestations of Design Ideas," *ACM Trans. Computer-Human Interaction*, vol. 15, no. 2, 2008, article 7.
13. J.A. Landay and B.A. Myers, "Sketching Interfaces: Toward More Human Interface Design," *Computer*, vol. 34, no. 3, 2001, pp. 56–64.

IMPACT

Editor: **Michiel van Genuchten**
mtonyx
genuchten@ieee.org

Editor: **Les Hatton**
Kingston University
les.hatton@kingston.ac.uk

# The Generational Impact of Software

Anne-Francoise Rutkowski, Carol Saunders, and Les Hatton

Eighteen Impact columns to date have talked only about growing software. Anne-Francoise Rutkowski, Carol Saunders, and Les Hatton indicate that there are also generational factors that can have significant business impact and can limit growth in our field. —*Michiel van Genuchten*

**INSTALLMENTS OF THIS** column have discussed software's extraordinary impact on society. The original intent was to produce an approximate idea of the answers to several important business questions: How much software do we depend on, where is it, how is it produced, and what's the business impact of deploying and maintaining it?

However, software can have a significant negative generational impact on society—in particular, on its oldest and youngest users. This rapidly growing and often relatively affluent sector of society is becoming increasingly disenfranchised by consumer systems with interfaces that appear to have been designed by Klingons. This isn't meant in a derogatory sense for those *Star Trek* fans out there; it simply represents a generationally disjointed viewpoint. In short, the relentless pursuit of technology to gain a marketing edge has led to consumer system interfaces being loaded with software features that might be transparently obvious to their designers but are anything but obvious to the end user. These features cost money, so they have an obvious business cost, but their business value is dubious, to say the least.

## Cars, Washing Machines, and TVs

The interfaces in modern automobiles are becoming so complex that they now challenge a long-standing, fundamental assumption: wherever you are in the world, you can rent a car, flip your mind into left- or right-side driving, and just drive away knowing that all the basics are in the same place regardless. Let's face it, in an emergency, you need them to be. But this is more and more difficult as digital interfaces and their erratic and frequently fashion-driven design replace all the familiar components—the handbrake, the radio (which can be surprisingly difficult to turn off in some cars), and even the lights.

Just recently, one of us (Les) tried to buy a washing machine for his elderly mother. He failed. Nothing on the market appears to have an interface that, well, washes and dries. Instead, a long list of features is presented involving exotic kinds of wash—maybe somebody wants to wash their sub-aqua gear along with the net curtains, but generally speaking, most don't.

A walk to a nearby elderly care home revealed the majority of the occupants struggling with two or even three handsets with tiny handwriting and incomprehensible menus designed by other Klingons. As digital television transmission sweeps Europe, televisions need to be reset or retuned, which requires wading through a Kafkaesque menu system full of idioms that don't have the slightest meaning to most of

0740-7459/13/$31.00 © 2013 IEEE

the population—"Download software: yes/no?" Why? What does it do? Will it crash? Do I need it? A chat with the technician who has to go around sorting the digital conversions out confirmed that it is indeed a nightmare.

## More Software, More Problems

It doesn't stop here; society plunges ever deeper into a digital mire of its own making. A recent trip through Heathrow Airport with a boarding pass printed from Les's home printer couldn't be read by the departures system. On being sent back to the ticket counter, he found that a second pass wouldn't be issued because one already had. After a fair amount of arguing, a second pass eventually was issued, but the departures system still wouldn't accept it because now there were two (even though it couldn't read one).

Dropping into the local hospital reveals multiple systems that can't talk to each other and can't even locate patients based on their surname—or maybe it's just that the operator can't figure out how to do it. Again, Klingon speak dominates the menus and help systems, making it virtually impossible to figure out what on Earth the programmer is trying to get the user to do.

In some ways, we seem to be at a crossroads: the amount of software and the opportunities it brings are growing exponentially, but an increasingly large part of the population, and a reasonably affluent one at that, is becoming disenfranchised and fed up with the whole thing.

Where do we go from here? You could argue that the problem will eventually sort itself out as the older generation dies off, but remember this: today's younger generation is tomorrow's older generation, and the technological overturn under market stresses won't just go away.

Let's consider some insights from organizational psychology.

Software is constantly generated for a growing number of applications. Previous Impact columns persuade us that this code continues to grow at a steady pace across a broad range of industries. The compound annual growth rates (CAGRs) of approximately 1.16 for lines of code (LOC) demonstrate the human mind's underlying power—but also its limitations. Teams of smart software engineers add to the software inventory at this steady pace.[1] The business viewpoint is that the technology industry provides a great service by increasing the number of options and multiplying functionality. Countless software jewels are being generated—but can we figure out what we want, let alone what we need? Businesses of course want to grow their revenue, and the only way they see to do this is to grow their software. Much like soda companies wanting to sell more sugar, many businesses add features to their applications to sweeten their appeal. Glenn Ellison and Drew Fudenberg explained monopolists' incentives for providing upgraded versions of software even when society would be much better off without them.[2] Let's now address the negative impacts that increasing numbers of software features and upgraded versions are having on users' minds, in particular, IT-related overload and the potential loss of skills in younger generations.

> Overload—more precisely, IT-related overload—has emotional and cognitive symptoms.

### IT-Related Overload

Are there any limits to the growth of software from the user's perspective? Research suggests that users are generally no longer interested in struggling through yet another upgrade or learning one more software tool or technological gadget designed solely to increase its developer's revenues.[3] Indeed, technology users have reported experiencing technology overload as well as the much-heralded information overload. Both negatively impact workers' productivity and performance.

Overload—more precisely, IT-related overload—has emotional and cognitive symptoms.[4] It's typically associated with emotional symptoms such as frustration, distractibility, and inner frenzy. It's also associated with cognitive symptoms such as making mistakes or simply dropping tasks. Regardless of the type of overload symptom, organizational performance suffers.[5]

We can use a blender to explain users' limits in dealing with IT-related overload. The input to be processed represents requirements created by the new technology, and the blender represents the mind. A certain level of mental effort is necessary to process new technologies and their features. Indeed, just like the energy that runs the blender, mental effort is required to process technological features.

However, while blenders might be relatively similar in their processing power, this certainly isn't the case for individuals. Some individuals can easily adapt to multiple technologies while others struggle with only a few. Let's take the example of finding a washing

machine for your elderly mother, as described earlier. You have to adapt your perception of the washing machine's many features to anticipate those of your mother. You must identify what will be pertinent to her. Obviously, your mother won't need some (if not most) of the software-supported features of the new washing machines. For her, these buttons likely will only serve as a source of emotional and cognitive overload when she tries to figure out which options she should select to wash her tea towels.

### Loss of Skills in Younger Generations

Software marketers want us to believe that all the technology mastery problems will fade away with new generations of technology users as the older ones leave the market. We know that the idea that younger generations will be able to handle the technology so well that they'll never experience overload is an illusion, owing to the limitation of our brains. Our brains can only store a limited amount of information at one time,[6] so IT applications could actually limit younger generations' brains even more. Recently, researchers demonstrated that, in the developing brain, associations are built on real-world interactions between the body and environment.[7] For instance, they found that neural activity was far more enhanced in children who had practiced printing by hand than in those who had simply looked at letters on a screen.[8] The human brain must cognitively adapt to any new technology's features.[9]

As economic growth in software is led by hungry developers, marketers will want to portray older generations as outdated when they can't cope with new technologies. In ancient and other civilizations, the old are revered as wise, and getting old is an achievement in and of itself. In today's society, getting old is typically equated to being outdated. Marketers want older folks to believe that technologies are outsmarting them and that they need a serious update to stay in tune with the world of modernity. Is that true? We think not for several reasons. First, there will always be younger generations of IT developers to overload the aging generation of users. Second, many people are concerned that our kids will lose important skills. They won't be able to orient themselves without a GPS when lost in town or add numbers without a calculator. They could lose such skills if we stop educating them. Who is to decide which skills are to be lost?

Some designers see simplification of software by designing one magic button as the solution. Is this progress? Some miss the complexity of fiddling with the equalizer on their sound system while listening to music: they enjoy the precision. Others are roused by the complexity of their GPS and multiple display options and voices: they enjoy mastering them.

Since the beginning of time, humans have wanted to be faster and better and to overcome their limitations. Today, software allows amazing discoveries on Mars. But in a world where the universe and answers to its many mysteries seem to be only one click away, we still face those human limitations, both as individuals and as teams of software developers.

So if the magical button isn't a solution to all our problems, what could we recommend instead? First, software developers need to carefully reflect on upgrades. They need to ask themselves and their teammates why a new version is being brought to the market in the first place and who it benefits. Functionality extensions require an effort on the part of users, so software developers should only prepare upgrades that are truly helpful to their users. Second, they need to design new technologies that are customizable and adaptable to users' needs. You can still please users who want complexity and greater functionality, but not at the expense of those who prefer the magic button. Most importantly, software developers should be responsible and think faster than they develop, not the other way around.

## References


1. M.V. Genuchten and L. Hatton, "Compound Annual Growth Rate for Software," *IEEE Software*, vol. 29, no. 4, 2012, pp. 19–21.
2. G. Ellison and D. Fudenberg, "The Neo-Luddite's Lament: Excessive Upgrades in the Software Industry," *RAND J. Economics*, vol. 31, no. 2, 2000, pp. 253–272.
3. P. Karr-Wisniewski and Y. Lu, "When More Is Too Much: Operationalizing Technology Overload and Exploring Its Impact on Knowledge Worker Productivity," *Computers in Human Behavior*, vol. 26, no. 5, 2010, pp. 1061–1072.
4. A.F. Rutkowski and C. Saunders, "Growing Pains with Information Overload," *Computer*, June 2010, pp. 94–96.
5. E.M. Hallowell, "Overloaded Circuits: Why Smart People Underperform," *Harvard Business Rev.*, Jan. 2005, pp. 1–9.
6. G.A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Rev.*, vol. 63, no. 2, 1956, pp. 81–97.
7. K.H. James and T.P. Atwood, "The Role of Sensorimotor Learning in the Perception of Letter-Like Forms: Tracking the Causes of Neural Specialization for Letters," *Cognitive Neuropsychology*, Feb. 2009, pp. 91–110.
8. "How Hand Writing Boosts the Brain," *Wall Street J.*, 5 Oct. 2010; http://online.wsj.com/article/SB10001424052748704631504575531932754922518.html.
9. C. Saunders et al., "Virtual Space and Place: Theory and Test," *MIS Q.*, vol. 35, no. 4, 2011, pp. 1079–1098.



**ANNE-FRANCOISE RUTKOWSKI** is an associate professor in the Information and Management Department at Tilburg University. Contact her at a.rutkowski@uvt.nl.

**CAROL SAUNDERS** is a professor at the College of Business Administration at the University of Central Florida and Schoeller Senior Scholar at the Dr. Theo and Friedl Schoeller Research Center for Business and Society. Contact her at csaunders@bus.ucf.edu.

**LES HATTON** is a director of Oakwood Computing Associates and professor of forensic software engineering at Kingston University. Contact him at les.hatton@kingston.ac.uk or via www.leshatton.org.

*continued from p. 92*

Consider the *W*1 case-based reasoning (CBR) system, also known as "Dub-ya" or the "the decider."[3] CBR makes conclusions by inspecting the nearest similar historical cases. To make *W*1 a landscape miner (which we'll call *W*2), we can cluster the training data into a tree of clusters, where child nodes contain subclusters of the parents. Then, a feature selector runs over the data to reject features whose values can't distinguish the clusters. Specifically, we're checking the entropy of each attribute value over all clusters and deleting those with the highest entropy. Finally, we can replace all leaf clusters with the median of each cluster. The resulting space of features and examples is very small: dozens of features reduce down to just a handful, and hundreds of examples reduce down to just one example per cluster.

By restricting inference to just some subtree of clusters (where the leaves now contain just one representative example), we can quickly build many local models specialized to particular contexts.

*W*2 has two important features. First, it's a landscape miner in that it maps out different regions of data inside of which we might build different models. Second, while the assembly of ideas is somewhat unique, each part of *W*2 is a known tool to the predictive modeling community. That is, it's possible for the predictive community modeling community to refocus and redirect its tools toward an interesting new goal.

## Decision Mining

At a recent panel on software analytics at ICSE 2012, industrial practitioners reviewed the state of the art in data mining.[4] Panelists commented, "Prediction is all well and good, but what about decision making?" Data mining is useful because it focuses an inquiry onto particular issues, but data miners are subroutines in a higher-level decision process.

To convert *W*2 into a decision miner (which we'll call *W*3), we add contrast set learning. While classifiers report what's true about different regions of data, contrast set learners report how those regions differ. Contrast sets can be much smaller than classification rules, particularly if they're generated as a postprocessor to some decision tree process. Contrast sets learned high in a decision tree tend to wipe out most possibilities and select for few classes—they do this by using fewer extra constraints.

*W*3 uses the same clusters as found by *W*2, but applies the principle of envy. Each cluster finds the closest neighboring cluster that it most desires—for example, for effort estimation, the neighboring cluster with the projects that are cheaper to build. *W*3 then applies a contrast set learner to the neighboring cluster to find best practices for achieving those better results in that cluster. In a recent *IEEE Transactions on Software Engineering* paper, I showed that such envy-based "local learning" can result in much better models than if we overgeneralize by learning from all the data.[5]

The lesson of *W*3 is the same as *W*2: new and innovative approaches to predictive modeling can be achieved by refactoring our current tools.

> Prediction is all well and good, but what about decision making?

## Discussion Mining

Pablo Picasso once said "computers are stupid; they only give you answers." Discussion miners aren't stupid; they know that while predictions and decisions are important, so too are the questions and insights generated on the way to those conclusions. In my view, discussion mining is the next great challenge for the predictive modeling community. In the coming century's heavily digital world, such discussion tools are going to be essential. Without them, humans will be unable to navigate and exploit the ever-increasing quantity of readily-accessible digital information.

In some sense, discussion miners are the very opposite of the Web:

- The Web was designed for information transport and access, with a primary goal of rapid sharing of new information.
- If the Web were a discussion miner, it would be possible to instantly query each webpage to find other pages with similar (or disputing) beliefs, find the contrast set between then agreeing and disputing pages, and then run queries that helped the reader assess the plausibility of each item in that contrast set.

Note that much of the current predictive modeling research wouldn't qualify as a discussion miner because, in the usual case, most of that literature is still struggling with methods to

TABLE 1

Internals of a discussion miner.

| Level | What | Task | Uses |
|---|---|---|---|
| 0 | Do | Predict, decide | Regression, classification, nearest neighbor reasoning |
| 1 | Say | Summarize, plan, describe | Instance section, feature selection, contrast sets |
| 2 | Reflect | Trade-offs, envelopes, diagnosis, monitoring | Clustering, multiobjective optimization, anomaly detectors |
| 3 | Share | Privacy, data compression, integrate old and new rules, recognize and debate deltas between competing models | Contrast set learning, transfer learning |
| 4 | Scale | Do all of the above, quickly | ? |

create one model, let alone updating a model as time progresses.

One fascinating open issue with discussion miners is how they should be assessed. In discussion mining, the model's goal is to find its own flaws and replace itself with something better, which brings to mind a quote from Susan Sontag: "The only good answers are the ones that destroy the questions." In other words, we shouldn't assess such models by accuracy, recall, or precision—rather, we should assess the audience engagement they engender. No, I don't know how to do that either, but I find it exciting that there are such clear and important problems waiting for us to solve tomorrow.

In terms of engineering principles, Table 1 shows the internals of a discussion miner. Note that the predictive modeling community already has the parts needed to assemble this and other new kinds of miners.

We must move on, and we can. Enough already with algorithm mining: it's time to do other things. Industrial practitioners aren't really concerned with the internal details of our algorithms or how our data divides into regions. They're more concerned with the tools needed to help push the community to debate different possible decisions.

## References


1. K. Dejaeger et al., "Data Mining Techniques for Software Effort Estimation: A Comparative Study," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 375–397.
2. T. Hall et al., "A Systematic Review of Fault Prediction Performance in Software Engineering," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1276–1304.
3. A. Brady and T. Menzies, "Case-Based Reasoning vs. Parametric Models for Software Quality Optimization," *Proc. 6th Int'l Conf. Predictive Models in Software Eng.* (PROMISE 10), ACM, 2010; http://doi.acm.org/10.1145/1868328.1868333.
4. T. Menzies and T. Zimmermann, "Goldfish Bowl Panel: Software Development Analytics," *Proc. 2012 Int'l Conf. Software Eng.* (ICSE 2012), IEEE, 2012, pp. 1032–1033.
5. T. Menzies et al., "Local vs. Global Lessons for Defect Prediction and Effort Estimation," *IEEE Trans. Software Eng.*, preprint, published online Dec. 2012; http://goo.gl/k6qno.


**TIM MENZIES** is a full professor of computer science at the Lane Department of Computer Science and Electrical Engineering, West Virginia University. Contact him at tim@menzies.us.



*IEEE Software* (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscription rates: IEEE Computer Society members get the lowest rate of US$56 per year, which includes printed issues plus online access to all issues published since 1984. Go to www.computer.org/subscribe to order and for more information on other subscription prices. Back issues: $20 for members, $209.17 for nonmembers (plus shipping and handling).

**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.

SOUNDING BOARD

Editor: **Philippe Kruchten**
University of British Columbia
pbk@ece.ubc.ca

# Beyond Data Mining

Tim Menzies

**THE PREDICTIVE MODELING** community applies data miners to artifacts from software projects. This work has been very successful—we now know how to build predictive models for software effects and defects and many other tasks such as learning developers' programming patterns (see the extended version of this article at http://menzies.us/pdf/13idea.pdf for more detail).

That said, to truly impact the work of industrial practitioners, we need to change the predictive modeling community's focus. To date, it has spent too much time on *algorithm mining* when the field is moving into what I call *landscape mining*. To support industrial practitioners, we're going to have to move on to something I call *decision mining* and then *discussion mining*.

This article compares and contrasts the four kinds of miners shown in Figure 1:

- Algorithm miners explore tuning parameters in data mining algorithms.
- Landscape miners reveal the shape of the decision space.
- Decision miners comment on how best to change a project.
- Discussion miners help the community debate trade-offs regarding the different decisions.

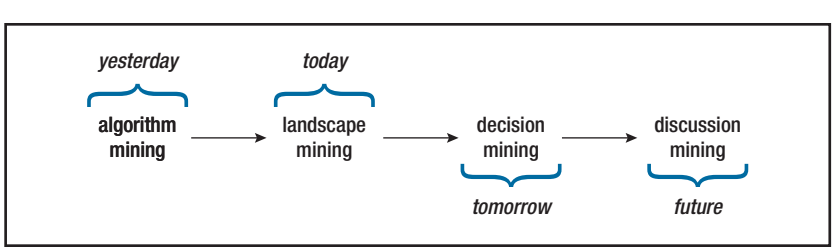



**FIGURE 1.** Four kinds of miners shown left to right, past to future.

Note that algorithm and landscape mining are more research-focused activities that explore the miners' internal details. However, decision and discussion miners are more practitioner-oriented because they're focused on how a community can use conclusions.

## Algorithm Mining

While it's rarely stated, the original premise of predictive modeling was that predictions should guide software management—in other words, once upon a time, the aim of a prediction was a decision.

Sadly, that original aim seems to be forgotten. Too many researchers in the field are stuck in a rut, publishing papers that spend very little time exploring the data and much more time on the data algorithms. Most of these papers focus on exploring configuration options with the algorithms, rather than reflecting on the underlying data. Recent papers report that there's little to be gained from such algorithm mining because the "improvements" found from this approach are marginal, at best—for example, for effort estimation and defect prediction, simpler data miners do just as well or better than more elaborate ones.[1,2]

## Landscape Mining

Algorithm mining is a "leap before your look" approach in which researchers throw algorithms at data and then see what comes out. A second approach is the "look before you leap" option—mining the data to find the space of possible inferences before leaping in with the learners. This is the data's "landscape."

*continued on p. 90*



0740-7459/13/$31.00 © 2013 IEEE

# Software Experts Summit

Redmond, WA • July 17, 2013

## Smart Data Science: Harnessing Data for Intelligent Decision Making

VIDEO

**REGISTER NOW!** This dynamic event, brought to you by *IEEE Software* magazine, will look beyond the questions of how to manage and store "big data" to how organizations can harness "smart data" to make intelligent decisions.

**Take advantage of early-bird pricing!**
**Register by June 1 and SAVE 25%!**

Learn from thought leaders from both industry and research. Speakers include:

James Whittaker
Microsoft

Paul Zikopoulos
IBM

Wolfram Schulte
Microsoft Research

Ayse Bener
Ryerson University

Forrest Shull
Fraunhofer Center for Experimental Software Engineering

Presented by
Software
IEEE computer society

**www.computer.org/ses13**