Complexity is the Only Constant: Trends in Computing and Their Relevance to Model Driven Engineering

Juergen Dingel

School of Computing
Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

Abstract. Despite ever increasing computational power, the history of computing is characterized also by a constant battle with complexity. We will briefly review these trends and argue that, due to its focus on abstraction, automation, and analysis, the modeling community is ideally positioned to facilitate the development of future computing systems. More concretely, a few, select, technological and societal trends and developments will be discussed together with the research opportunities they present to researchers interested in modeling.

1 Introduction

The development of computing is remarkable in many ways, and perhaps most of all in its progress and impact. However, due to the economic significance of computing and the pace of societal and technological change, we are constantly presented with new questions, challenges, and problems, giving us little time to reflect on how far we have come. Also, computing has become such a large and fragmented field that it is impossible to keep abreast all research developments.

This paper wants to briefly review some select past and present developments. Its main goal is to inform, stimulate, and inspire, not to convince. It will attempt to do so in a somewhat eclectic, anecdotal manner without claims of comprehensiveness, mostly driven by the author's interest, but with ample references to allow interested readers to dig deeper.

2 Complexity

> *"Complexity, I would assert, is the biggest factor involved in anything having to do with the software field."*
>
> Robert L. Glass [23]

In general, complex systems are characterized by a large number of entities, components or parts, many of which are highly interdependent and tightly coupled such that their combination creates synergistic, emergent, and non-linear

	Lines of code (approx.) (in million)
Operating Systems	
Windows NT 3.1 (1993)	0.5
Windows 95	11
Windows 2000	29
Windows XP (2001)	35
Windows Vista (2007)	50
Windows 7	40
Mac OS X	85
Android OS	12
Automobiles	
1981	0.05
2005	10
2015 (high end)	100
Miscellaneous	
Pacemaker	0.1
Mars Curiosity Rover	5
Firefox	10
Intuit Quickbook	10
Boeing 787	14
F-35 fighter jet	24
Large Hadron Collider	50
Facebook	60
Google (gmail, maps, etc)	2,000

Fig. 1. Approximate size of software in various products [7,48]

behaviour [29]. One of the prime examples of a complex system is the human brain consisting, approximately, of 10^{11} neurons connected by 10^{15} synapses [11].

Figure 1 shows the size of software in different kinds of products. Noteworthy here are not only the absolute numbers, but also the rate of increase. Automotive software is a good example here. Just over 40 years ago, cars were devoid of software. In 1977, the General Motors Oldsmobile Tornado pioneered the first production automotive microcomputer ECU: a single-function controller used for electronic spark timing. By 1981, General Motors was using microprocessor-based engine controls executing about 50,000 lines of code across its entire domestic passenger car production. Since then, the size, significance, and development costs of automotive software has grown to staggering levels: Modern cars can be shipped with as much as 1GB of software encompassing more than 100 million lines of code; experts estimate that more than 80% of automotive innovations will be driven by electronics and 90% thereof by software, and that the cost of software and electronics can reach 40% of the cost of a car [25].

The history of avionics software tells a similar story: Between 1965 and 1995, the amount of software in civil aircraft has doubled every two years [14]. If growth

continues at this pace, experts believe that limits of affordability will soon be reached [79].

Lines of code is a doubtful measure of complexity[1]. Nonetheless, it appears fair to say the modern software is one of the most complex man-made artifacts.

2.1 Why has complexity increased so much?

An enabler necessary for building and running modern software certainly is modern hardware. Today's software could not run on yesterday's hardware. The hardware industry has produced staggering advances in chip design and manufacturing which have managed to deliver exponentially increasing computing power at exponentially decreasing costs. Compared to the Apollo 11 Guidance Computer used 1969[2] a standard smart phone from 2015 (e.g., iPhone 6) has several tens of million of times the computational power (in terms of instructions per second)[3]. In 1985, an 2011 iPad2 would have rivaled a four-processor version of the Cray 2 supercomputer in performance, and in 1994, it still would have made the list of world's fastest supercomputers [45]. According to [47], the price of a megabyte of memory dropped from US$411,041,792 in 1957 to US$0.0037 in December 2015 — a factor of over 100 billion! The width of each conducting line in a circuit (approx. 15 nanometers) is approaching the width of an atom (approx. 0.1 to 0.5 nanometers).

But, it is not just technology that is getting more complex, life in general does, too. According to anthropologist and historian Josef Tainter, "the history of cultural complexity is the history of human problem solving" [73]. Societies get more complex because "complexity is a problem solving strategy that emerges under conditions of compelling need or perceived benefit". Complexity allows us to solve problems (e.g., food or energy distribution) or enjoy some benefit. Ideally, this benefit is greater than the costs of creating and sustaining the complexity introduced by the solution.

2.2 Consequences of complexity

On the positive side, complex systems are capable of impressive feats. AlphaGo, the Go playing system that in March 2016 became the first program to beat a professional human Go player without handicaps on a full-sized board in a five-game match, was said by experts to be capable of developing its own moves: "All but the very best Go players craft their style by imitating top players. AlphaGo seems to have totally original moves it creates itself" [5], providing a great example of — seemingly or real — emergent, synergistic behaviour.

[1] So many alternative ones have been proposed [61] that even the study of complexity appears complex

[2] A web-based simulator can be found at http://svtsim.com/moonjs/agc.html

[3] https://www.quora.com/How-much-more-computing-power-does-an-iPhone-6-have-than-Apollo-11-What-is-another-modern-object-I-can-relate-the-same-computing-power-to

On the negative side, complexity increases risk of failure. Data on the failures of software or software development are hard to come by; according to the US National Institute of Standards and Technology, the cost of software errors in the US in 2001 was US$ 60 billion [63] and in 2012 the worldwide cost of IT failure has been estimated to be $3 trillion[4].

A recent example illustrates how subtle bugs can be and how difficult it is to build software systems correctly: Chord is a protocol and algorithm for a peer-to-peer distributed hash table first presented in 2001 [72]. The work identified relevant properties and provided informal proofs for them in a technical report. Chord has been implemented many times[5] and went on to win the SIGCOMM Test-of-Time Award in 2011. The original paper currently has over 12,000 citations on Google scholar and is listed by CiteSeer as the 9th most cited Computer Science article. In 2012, it was shown that the protocol was not correct [82].

2.3 How to deal with complexity

Computer science curricula teach students a combination of techniques to deal with complexity, the most prominent of which are decomposition, abstraction, reuse, automation, and analysis. Of these, abstraction, automation, and analysis lie at the heart of MDE. These principles have served us amazingly well. Examples include the development of programming languages in general, and Peter Denning's ground-breaking work on virtual memory in particular [15]. But, e.g., 'The Law of Leaky Abstractions'[6], the 'Automation Paradox' [22], and the Ariane 5 accident in 1996 [1] have also taught us that even these techniques must be used with care.

3 Developments and opportunities

> *"I have no doubt that the auto industry will change more in the next five–10 years than it has in the last 50"*
>
> Mary Barra, GM Chairman and CEO, January 2016 [24]

> *"Only 19% of [175] interviewed auto executives describe their organizations as prepared for challenges on the way to 2025"*
>
> B. Stanley, K. Gyimesi, IBM IBV, January 2015 [71]

Making predictions in the presence of exponential change is very difficult[7]. For instance, when asked to imagine life in the year 2000, 19th century French artists came up with robotic barbers, machines that read books to school children, and radium-based fireplaces[8]; when the concept of a personal computer

[4] http://www.zdnet.com/article/worldwide-cost-of-it-failure-revisited-3-trillion
[5] At least 8 implementations are listed at https://github.com/sit/dht/wiki/faq
[6] http://www.joelonsoftware.com/articles/LeakyAbstractions.html
[7] http://uday.io/2015/10/15/predicting-the-future-and-exponential-growth
[8] http://singularityhub.com/2012/10/15/19th-century-french-artists-predicted-the-world-of-the-future-in-this-series-of-postcards

was first discussed at IBM, a senior executive famously questioned its value[9]. However, predicting further accelerating levels of change appears to be a safe bet. Increasing amounts of software are very likely to come with that, meaning there should be lots of things to do for software researchers.

The following list is highly selective and meant to complement more comprehensive treatments such as [65]. Also, we will focus most on technology; however, as pointed out in [65], more technology is not always the answer.

3.1 Semantics engineering

Capturing the formal semantics of general purpose programming languages has been a topic of research for a long time, but the richness of these languages present challenges that limit a more immediate, practical application of the results contributing to a widespread belief that formal semantics are for theoreticians only. However, the recent interest in Domain Specific Languages (DSLs) appears to present new opportunities to leverage formal semantics. Compared to General Purpose Languages (GPLs), a DSL typically consists of a smaller number of carefully selected features. Often, semantically difficult GPL constructs such as objects, pointers, iteration, or recursion can be avoided; expressiveness is lost, but tractability is gained.

The literature contains some examples showing how this increased tractability can be leveraged to facilitate formal reasoning. For instance, automatic verifiers have been built for DSLs for hardware description [13], train signaling [18], graph-based model transformation [66], and software build systems [10].

However, the improved tractability of DSLs might also greatly facilitate the automatic generation of supporting tooling. Looking at how widely used techniques to describe the *syntax* of a language have become to generate syntax processing tools, the vision is clear: Use descriptions of the *semantics* of a language to facilitate the construction of semantics-aware tools for the execution and analysis of that language.

An inspiring example This idea has already been explored in the context of programming languages [52,6,77,28]) and modeling languages [19,43,53,83] to, e.g., implement customizable interpreters, symbolic execution engines, and model checkers. However, the work in [40], in which abstract interpreters for a language are generated automatically from a description of its formal semantics, shows that more is possible. Given a description of the operational semantics of a machine-language instruction set such as x86/IA32, ARM, or SPARC in a domain-specific language called TSL, and a description of how the base types and operators in TSL are to be interpreted "abstractly" in an abstract semantic domain, the TSL tool automatically creates an implementation of an abstract interpreter for the instruction set:

$$\text{TSL} : \text{concrete semantics} \times \text{abstract domain} \longrightarrow \text{abstract semantics}$$

[9] http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/personalcomputer

The abstract interpreter can then be used by different analysis engines (e.g., for finding a fixed-point of a set of dataflow equations using the classical worklist algorithm, or for performing symbolic execution) to obtain an analyzer that is easily retargetable to different languages. The tool offers an impressive amount of generality by supporting different instruction sets and different analyses. It has been used to build analyzers for the IA32 instruction set that perform value/set analysis, definition/use analysis, model checking, and Botnet extraction with a precision at least as high as manually created analyzers.

Lowering barriers, increasing benefit Recent formalizations of different industrial-scale artifacts including operating system kernels [35], compilers [38], and programming languages including C [17], JavaScript [57] and Java [4] provide some evidence that large-scale formalizations are becoming increasingly feasible. Efforts are underway to make the expression, analysis, and reuse of descriptions of semantics more scalable, effective, and mainstream [21,62,54]. Paired with the increasing maturity and adoption of language workbenches such as Xtext[10], this work may allow substantial progress on the road towards the automatic generation of semantics-aware tools such as interpreters, static analyzers, and compilers. Descriptions of semantics might one day be as common and useful as descriptions of syntax are today.

3.2 Synthesis

The topic of synthesis has been receiving a lot of attention recently. For most of these efforts, 'synthesis' refers to the process of automatically generating executable code from information given in some higher level form: Examples include the generation of code that manipulates many different artifacts (e.g., bitvectors [70], concurrent data structures [69], database queries [9], data reprentations [68], or spreadsheets [26]), gives feedback to students for programming assignments [68], or implements an optimizing compiler [8]. Some of these examples use a GPL, some use a DSL. The synthesis itself is implemented either using constraint solving or machine learning. Different proposals on how to best integrate synthesis into programming languages have been made and have targeted GPLs such as Java [49,31] and DSLs [75].

Given that abstraction, automation and analysis are central to MDE, synthesis certainly also is of interest to the modeling community and the work on synthesis and its applications should be followed closely. In [74,75], the idea of "solver-aided DSLs" is introduced. The paper presents a framework in which such DSLs can be created and illustrates its use with a DSL for example-based web scraping in which the solver is used to generate an XPath expression that retrieves the desired data.

MDE features a range of activities and situations which might potentially benefit from a little help from a solver capable of finding solutions to constraints. Could the idea of synthesis and the use of solvers facilitate, e.g.,

[10] https://eclipse.org/Xtext

– the development of models via extraction or autocompletion,
– the support for partial models with incomplete or uncertain information,
– the analysis of models,
– the refinement of models via, e.g., the generation of substate machines from interface specifications,
– the generation of correct, efficient code from models,
– the generation of different views from a model?

How could synthesis be leveraged in language workbenches that generate supporting tools such as analyzers and code generators, or in model transformation languages and engines that support different transformation intents [44]?

Some attempts to leverage synthesis for, e.g., model creation [36], transformation authoring [2], design space exploration [27] already exist, but the topic hardly seems exhausted. Indeed, some of the technical issues Selic mentions in [65] might be mitigated using synthesis including dealing with abstract, incomplete models, model transformation, and model validation.

3.3 Reconciling formal analysis and evolution

There is a fundamental conflict between analysis and evolution: As soon as the model evolves (changes), any analysis results obtained on the original version may be invalidated and the analysis may have to be rerun. Unfortunately, both seem unavoidable not just in the context of MDE, but software engineering in general.

Most analyses require the creation of supporting artifacts that represent analysis-relevant information about the model. For instance, software reverse engineering tools collect relevant information about the code in a so-called fact repository typically containing a collection of tuples encoding graphs [34]; most static analysis tools require some kind of dependence graph, and test case generation tools often rely on symbolic execution trees.

When the cost of the analysis rises, the motivation to avoid a complete reanalysis after a change and to leverage information about the nature of the change to optimize the analysis increases as well. In general, aspects of this topic are handled in the literature on impact analysis [39]; however, the analyses considered typically are either manual (comprehension, debugging) or rather narrow (regression testing, software measurement via metrics), and do not consider, e.g., static analyses or analyses based on formal methods.

Two approaches Assuming the analysis requires supporting artifacts, there are, in principle, at least two ways of reconciling analysis and evolution [33]:

1. Artifact-oriented (Figure 2): The goal here is to update the supporting artifact A_1 as efficiently as possible, but in such a way that it becomes fully reflective of the information in the changed program. To this end, the impact of the change Δ on the artifact original artifact A_1 is determined, and the parts of the artifact possibly affected are recomputed, while leaving parts known to be unaffected unchanged. Then, the updated artifact A_2 can be used as before

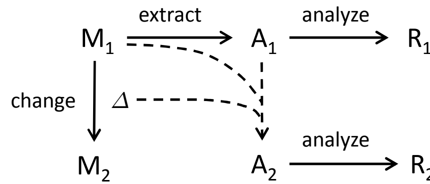

Fig. 2. Artifact-oriented approach to reconcile analysis and evolution. M_i, A_i, and R_i denote, respectively, a model, the artifact extracted from the model to support the analysis, and the analysis result

to perform all analyses it is meant to support. For instance, for analyses based on dependence graphs such as slicing or impact analysis, the parts of the graph affected by the change are updated and the result is used to recompute the result. Similarly, for a dead code analysis (or test case generation) using a symbolic execution tree (SET), affected parts of the tree would be updated to produce a tree corresponding to the changed program. In this approach, the savings come from avoiding the reconstruction of parts of the supporting artifact A_2.

2. Analysis-oriented (Figure 3): Here, the focus is on updating the result of the analysis as efficiently as possible, rather than the supporting artifact. To this end, the impact of the change Δ on the analysis result is determined, and the parts of the analysis that may lead to a different result due to the change are redone, ignoring any parts known to produce the same result. For instance, when impact analysis is used during regression testing, only tests for executions that were introduced by the change are run; tests covering unaffected executions are ignored [60]. In this approach, the focus is on reestablishing the analysis result R_2 as some combination $R_2 = op(\Delta, R_1, R_2')$ of the previous result R_1 and the partial result R_2'. E.g., an analysis-oriented optimization of the dead code analysis mentioned above (or test case generation) would use the most efficient means to determine dead code in (or test cases for) the affected parts and the construction of the full SET for the changed program may not be necessary for that; in this case, R_1 would be the dead code in (or test cases for) M_1 and the partial result R_2' would be the dead code in (test cases for) the parts of the model introduced by the change; the operation $op(\Delta, R_1, R_2')$ would return the union of R_2' and the dead code (test cases) in R_1 not impacted by the change. In this case, the savings come from avoiding unnecessary parts of the analysis.

Comparing the two approaches, we see an interesting tradeoff: The first approach does not speed up the analysis itself (only the update of the supporting artifact). However, it results in a complete supporting artifact (e.g., dependence graphs, SET) that can then be used for whatever analyses it supports (e.g., different static analyses for dependence graphs, and, test case generation, dead code analysis for SETs). Moreover, the result of the analysis of the changed model does not rely on the result of the analysis of the original program at all. The second approach speeds up the analysis itself, but since it focusses on the changed

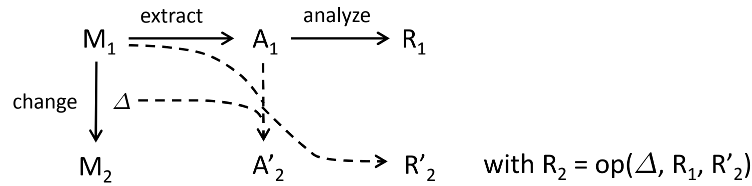

Fig. 3. Analysis-oriented approach to reconcile analysis and evolution. The analysis result R_2 for M_2 is obtained by combining the result for M_1 with the partial result R'_2

parts, it is partial only. E.g., the updated program can only be concluded to be free of dead code, if the second *and* the first analysis say so.

In sum, the second approach is more restricted compared to the first, but might well hold additional optimization potential. Recent research on program analysis using formal techniques has begun to explore these possibilities, and analysis-oriented approaches to optimize model checking [81] and symbolic execution [58] have been developed. Inspired by these proposals, we have developed prototypes that use both approaches to optimize the symbolic execution of Rhapsody statemachines [33]. Results indicate that both approaches are complementary and effective in different situations.

3.4 Open Science

In 2010, two Harvard economists published a paper entitled "Growth in a Time of Debt" in a non-peer reviewed journal which provided support for the argument that excessive debt is bad for growth. The paper was used by many policy makers to back up their calls for fiscal austerity. However, in 2013, the paper was shown to have used flawed methodology and to not support the authors' conclusions[11].

Reproducibility Examples of research producing doubtful results due to unintended or even intended flaws in the data or methodology have been going through the media recently and many disciplines have begun to investigate the reproducibility of their research results. For instance, a study in economics showed that 78% of the 162 replication studies conducted "disconfirm a major finding from the original study" [16]. A study focusing on research in Computer Systems [12], examined 601 papers from eight ACM conferences and five journals: of the papers with results backed by code, the study authors were able to build the system in less than 30 minutes only 32% of the time; in 54% of cases the study authors failed to build the code, but the paper authors said that the code does build with reasonable effort.

The U.S. President steps in However, it has been pointed out in prominent places that in many disciplines these days reproducibility means the availability

[11] A discussion of the paper and the controversy it caused can be found at https://en.wikipedia.org/wiki/Growth_in_a_Time_of_Debt

of programs and data [30,64,50]. In other words, since software, programming, and the use and manipulation of data plays such a central role in so many disciplines, some of the problems with reproducibility in other disciplines are due to limitations in programming, software, and the use and manipulation of data, that is, they are due to problems that the computing community is at least partially responsible for and should put on its research agenda[12]. About a year ago, the world's most powerful man has done exactly that with an executive order to create a "National Strategic Computing Initiative" which includes accessibility and workflow capture as central objectives [56].

A good start: encouraging artifact submission The research community has begun to adjust with, e.g., no less than four events devoted to reproducibility at the 2015 Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)[13], and Eclipse's Science Working Group announcing specific initiatives (Eclipse Integrated Computational Environment and Data Analysis Workbench). However, more should be done and promoting the value of artifact submission at workshops, conferences, and journal appears to be a good place to start. According to [12], 19 Computer Science conferences have participated in an artifact submission and evaluation process between 2011 and 2016, including PLDI'15, OOPSLA'15, ECOOP'15, and POPL'16, but more need to join. The availability of the artifacts that research is based and their integration into the scientific evaluation process should be the norm, not the exception.

3.5 Provenance

A topic closely related to open science and reproducibility is provenance. In general, data provenance refers to the description of the origins of a piece of data and the process by which it was created or obtained with the goal to allow assessments of quality, reliability, or trustworthiness. It has traditionally been studied in the context of databases, but has also been used for data found on the web or data used in scientific experiments. Domains of application include

- *science*, to make data and experimental results more trustworthy and experiments more reproducible,
- *business*, to demonstrate ownership, responsibility, or regulatory compliance and facilitate auditing processes, and
- *software development*, to aid certification and establish adherence to licensing rules.

OPM and PROV: metamodels and standards for provenance There are tools specifically devoted to the collection and representation of provenance

[12] Computers are even said to have "broken science", https://www.eclipsecon.org/na2016/session/how-computers-have-broken-science-and-how-we-can-fix-it

[13] http://sc15.sueprcomputing.org

data such as Karma[14] but also workflow engines supporting provenance such as Kepler[15]. Many of these tools support the Open Provenance Model (OPM), a data model (i.e., metamodel) for provenance information [51] based on directed, edge-labeled, hierarchical graphs with three kinds of nodes representing things (Artifact, Agent, and Process) and five kinds of edges representing causal relationships (used, wasGeneratedBy, wasControlledBy, wasTriggeredBy, and wasDerivedFrom). OPM graphs are subject to well-formedness constraints, can contain time information, and have inference rules (allowing, e.g., the inclusion of derived information via transitive edges) and operations (for, e.g., union, intersection, merge, renaming, refinement and completion) associated with them. A formal semantics of OPM graphs published recently views them as temporal theories on the temporal events represented in the graph [37], but does not account for Agents. OPM has been a major influence in the design of the PROV family of documents by the World Wide Web Consortium (W3C) [78] which not only defines a data model, but also corresponding serializations and other supporting definitions to enable the interoperable interchange of provenance information in heterogeneous environments such as the Web.

Open-ended opportunities There appears to be a lot of opportunity for researchers with background in graph transformation, formal methods, or modeling to advance the state-of-the-art in provenance. Many established topics (e.g., formal semantics, constraint solving, traceability, querying, language engineering for graphical DSMLs, and model management), but also emerging topics (e.g., the use of models and modeling to support inspection, certification and compliance checking [20,46,55] and data aggregation and visualization [76,42,48,41]) appear potentially relevant. Moreover, no approaches have been found to build models that allow the quantification of the quality or trustworthiness of data. In case of producer/consumer relationships, service level agreements guaranteeing data with a certain level of quality might also be of interest.

3.6 Open source modeling tools

The need to improve MDE tooling has been expressed before [65,80,32]. At the same time, significant efforts to develop industrial-strength open source modeling tools and communities that support and sustain them are currently being made. Sample tools include AutoFocus[16], xtUML[17], Papyrus[18][3], and PapyrusRT[19][59].

The development and availability of complete, industrial-strength open source MDE tools is a radical shift from past practices and presents both exciting opportunities and substantial challenges for everybody interested in MDE, regardless

[14] http://d2i.indiana.edu/provenance_karma
[15] https://kepler-project.org
[16] http://www.fortiss.org/en/about-us/alle-news/autofocus-3
[17] https://xtuml.org
[18] https://eclipse.org/papyrus
[19] https://www.eclipse.org/papyrus-rt

of whether they use the tools for industrial development, research, or education. Due to the importance of tooling to the success of MDE, this shift has the potential to provide a much-needed stimulus for major advances in its adoption, development, and dissemination.

4 Conclusion

> *"We can only see a short distance ahead, but we can see plenty there that needs to be done."*
>
> Alan Turing

As we continue to entrust more and more complex functions and capabilities to software, our ability to build this software reliably and effectively should increase as well. Much more work is needed to make this happen and this paper has suggested some starting points.

The fragmentation that plagues many research areas is harmful. Any scientific community should keep an open mind and remain willing to learn from others about existing and new problems and potentially new ways to solve them [67].

Acknowledgment

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by the Ontario Ministry of Research and Innovation (MRI).

References

1. Ariane 5 flight 501 failure, report by the inquiry board. `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`, 1996.
2. I. Baki and H. Sahraoui. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Transactions on Software Engineering and Methodology*, 2016. In print.
3. R. Barrett and F. Bordeleau. 5 years of 'Papyrusing' — migrating industrial development from a proprietary commercial tool to Papyrus (invited presentation). In *Workshop on Open Source Software for Model Driven Engineering (OSS4MDE'15)*, pages 3–12, 2015.
4. D. Bogdănaş and G. Roşu. K-Java: A Complete Semantics of Java. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.
5. S. Borowiec and T. Lien. AlphaGo beats human Go champ in milestone for artificial intelligence. *Los Angeles Times, March 12*, 2016.
6. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *ACM SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments (SDE'87)*, 1987.
7. R.N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, 2005.

8. A. Cheung, S. Kamil, and A. Solar-Lezama. Bridging the gap between general-purpose and domain-specific compilers with synthesis. In *Summit oN Advances in Programming Languages (SNAPL'15)*, 2015.
9. A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices*, 48(6):3–14, 2013.
10. M. Christakis, R.M. Leino, and W. Schulte. Formalizing and verifying a modern build language. In *International Symposium on Formal Methods (FM'14)*, 2014.
11. E.H. Chudler. Neuroscience for kids. `https://faculty.washington.edu/chudler/what.html`.
12. C. Collberg and T.A. Proebsting. Repeatability in computer systems research. *Communications of the ACM, Vol. 59 No. 3, Pages 62-69*, 59(3), 2016.
13. B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Workshop on Formal Techniques for Hardware and Hardware-like Systems*, 1998.
14. J.P. Potocki de Montalk. Computer software in civil aircraft. *Cockpit/Avionics Engineering*, 17(1):17–23, 1993.
15. P.J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3), 1970.
16. M. Duvendack, R.W. Palmer-Jones, and W.R. Reed. Replications in economics: A progress report. *Economics in Practice*, 12(2), 2015.
17. C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL12)*, pages 533–544, 2012.
18. J. Endresen, E. Carlson, T. Moen, K.-J. Alme, Ø. Haugen, G.K. Olsen, and A. Svendsen. Train control language – teaching computers interlocking. In *Computers in Railways XI*. WIT Press, 2008.
19. G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta-modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *International Conference on the Unified Modeling Language (UML'00)*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
20. D. Falessi, M. Sabetzadeh, L. Briand, E. Turella, T. Coq, and R.K Panesar-Walawege. Planning for safety standards compliance: A model-based tool-supported approach. *IEEE Software*, 29(3):64–70, 2012.
21. M. Felleisen, R.B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
22. D.E. Geer. Children of the magenta. *IEEE Computer*, September/October 2015.
23. R.L. Glass. Sorting out software complexity. *Communications of the ACM*, 45(11):19–21, 2002.
24. GM. GM chairman and CEO addresses CES. `https://www.gm.com/mol/m-2016-Jan-boltev-0106-barra-ces.html`, Jan 6, 2016.
25. K. Grimm. Software technology in an automotive company — major challenges. In *International Conference on Software Engineering (ICSE'03)*, 2003.
26. S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55:97–105, 2012.
27. M. Hendriks, T. Basten, J. Verriet, M. Brassé, and L. Somers. A blueprint for system-level performance modeling of software-intensive embedded systems. *Software Tools for Technology Transfer*, 18:21–40, 2016.
28. P.R. Henriques, M.J.V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the LISA system. *IEE Proceedings — Software*, 152:54–69, 2005.
29. T. Homer-Dixon. *The Ingenuity Gap*. Vintage Canada, 2001.

30. D.C. Ince, L. Hatton, and J. Graham-Cumming. The case for open computer programs. *Nature*, 482:485–488, 2012.
31. J. Jeon, X. Qiu, J.S. Foster, and A. Solar-Lezama. Jsketch: sketching for Java. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, 2015.
32. N. Kahani, M. Bagherzadeh, J. Dingel, and J.R. Cordy. The problems with Eclipse modeling tools: A topic analysis of Eclipse forums. Submitted, April 2016.
33. A. Khalil and J. Dingel. Incremental symbolic execution of evolving state machines. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, 2015.
34. H.M. Kienle and H.A. Mueller. Rigi — an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75:247–263, April 2010.
35. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Formal verification of an OS kernel. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP'09)*, pages 207–220. ACM, 2009.
36. A.S. Koksal, Y. Pu, S. Srivastava, R. Bodik, N. Piterman, and J. Fisher. Synthesis of biological models from mutation experiments. In *ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL'13)*, 2013.
37. N. Kwasnikowska, L. Moreau, and J. Van den Bussche. A formal account of the Open Provenance Model. *ACM Transactions on the Web*, 9, May 2015.
38. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), July 2009.
39. B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification, and Reliability*, 2012.
40. J. Lim and Th. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst.*, 35(1):4:1–4:59, April 2013.
41. M. Lima. Visual complexity website. `http://www.visualcomplexity.com/vc`.
42. M. Lima. *The Book of Trees: Visualizing Branches of Knowledge Hardcover*. Princeton Architectural Press, 2014.
43. Y. Lu, J.M. Atlee, N.A. Day, and J. Niu. Mapping template semantics to SMV. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'04)*, 2004.
44. L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G.M.K. Selim, E. Syriani, and M. Wimmer. Model transformation intents and their properties. *Software and Systems Modeling*, pages 1–38, 2014.
45. J. Markoff. The iPad in your hand: As fast as a supercomputer of yore. New York Times article based on interview with Dr. Jack Dongarra, May 9, 2011 2011. `http://bits.blogs.nytimes.com/2011/05/09/the-ipad-in-your-hand-as-fast-as-a-supercomputer-of-yore`.
46. A. Mayr, R. Plösch, and M. Saft. Objective safety compliance checks for source code. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, 2014.
47. J.C. McCallum. Memory prices (1957-2015). `http://www.jcmit.com/memoryprice.htm`, Accessed: 03/2016.
48. D. McCandless. Information is beautiful: Million lines of code. `http://www.informationisbeautiful.net/visualizations/million-lines-of-code`.

49. A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *International Conference on Software Engineering (ICSE'11)*, 2011.
50. D. Monroe. When data is not enough. *Communications of the ACM*, 58(12):12–14, 2015.
51. L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. van den Bussche. The Open Provenance Model Core Specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, June 2011.
52. P. Mosses. Sis: A compiler-generator system using denotational semantics. Technical Report 78-4-3, Dept. of Computer Science, University of Aarhus, 1978.
53. P. Muller, F. Fleurey, and J. Jézéquel. Weaving executability into object-oriented meta-languages. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'05)*, 2005.
54. D.P. Mulligan, S. Owens, K.E. Gray, T. Ridge, and P. Sewell. Lem: Reusable engineering of real-world semantics. *SIGPLAN Not.*, 49(9):175–188, August 2014.
55. S. Nair, J. de la Vara, A. Melzi, G. Tagliaferri, L. de la Beaujardiere, and F. Belmonte. Safety evidence traceability: Problem analysis and model. In *Requirements Engineering: Foundation for Software Quality*, 2014.
56. The President of the United States. Executive order: Creating a national strategic computing initiative. July 29, 2015. Available at: `https://www.whitehouse.gov/the-press-office/2015/07/29/executive-order-creating-national-strategic-computing-initiative`.
57. D. Park, A. Ştefănescu, and G. Roşu. KJS: A complete formal semantics of JavaScript. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.
58. S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, 2011.
59. E. Posse. PapyrusRT: modelling and code generation (invited presentation). In *Workshop on Open Source Software for Model Driven Engineering (OSS4MDE'15)*, 2015.
60. X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, 2004.
61. F. Riguzzi. A survey of software metrics. Technical Report DEIS-LIA-96-010, Università degli Studi di Bologna, 1996.
62. G. Roşu and T.F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
63. RTI. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-3, National Institute of Standards & Technology (NIST), May 2002.
64. R. Schuwer, M. van Genuchten, and L. Hatton. On the impact of being open. *IEEE Software*, 32:81 – 83, 2015.
65. B. Selic. What will it take? A view on adoption of model-based methods. *Software and System Modeling*, 11:513–526, 2012.
66. G.M.K. Selim, L. Lucio, J.R. Cordy, J. Dingel, and B.J. Oakes. Specification and verification of graph-based model transformation properties. In *International Conference on Graph Transformation (ICGT'14)*, pages 113–129, 2014.
67. S. Shapiro. Splitting the difference: The historical necessity of synthesis in software engineering. *IEEE Annals of the History of Computing*, 19(1), 1997.

68. R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Notices*, volume 48, pages 15–26. ACM, 2013.
69. A. Solar-Lezama, C. Jones, and R. Bodik. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, volume 43, pages 136–148. ACM, 2008.
70. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*, volume 40, pages 281–294. ACM, 2005.
71. B. Stanley and K. Gyimesi. Automotive 2025 – industry without borders. Technical report, IBM Institute for Business Value, January 2015. http://www-935.ibm.com/services/us/gbs/thoughtleadership/auto2025.
72. I. Stoica, R. Morris, D. Karger, F.M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 149–160, 2001.
73. J.A. Tainter. Complexity, problem solving, and sustainable societies. In R. Costanza, O. Segura, and J. Martinez-Alier, editors, *Getting Down to Earth: Practical Applications of Ecological Economics*. Island Press, 1996.
74. E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, 2013.
75. E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*, 2014.
76. E. Tufte. *Beautiful Evidence*. Graphics Press, 2006.
77. M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Oliver, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *International Conference on Compiler Construction (CC'01)*, volume LNCS 2027, 2001.
78. W3C Working Group. PROV-Overview: An overview of the PROV family of documents. In P. Groth and L. Moreau, editors, *W3C Working Group Note*. W3C, 2013.
79. D. Ward. Avsis system architecture virtual integration program: Proof of concept demonstrations. INCOSE MBSE Workshop, January 27 2013.
80. J. Whittle, J. Hutchinson, M. Rouncefield, and R. Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems (MODELS'13)*, 2013.
81. G. Yang, M. Dwyer, and G. Rothermel. Regression model checking. In *International Conference on Software Maintenance (ICSM'09)*, pages 115–124. IEEE, 2009.
82. P. Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):50–57, April 2012.
83. K. Zurowska and J. Dingel. *BM-FA 2009 – 2014: Revised Selected Papers*, chapter A Customizable Execution Engine for Models of Embedded Systems, pages 82–110. Springer, 2014.