Math. Struct. in Comp. Science (2013), *vol.* 23, *pp.* 934–943. © Cambridge University Press 2013
doi:10.1017/S0960129512000072

Postlude: seamless composition and integration – a perspective on formal methods research

AZER BESTAVROS, ASSAF KFOURY and ANDREI LAPETS

Computer Science Department, Boston University,
Boston, Massachusetts 02215, U.S.A.
Email: {best;kfoury;lapets}@bu.edu

Received 15 January 2012

1. Introduction

Have formal methods in computer science come of age? While the contributions to this special issue of *Mathematical Structures in Computer Science* attest to their importance in the design and analysis of particular software systems, their relevance to the field as a whole is far wider. In recent years, formal methods have become more accessible and easier to use, more directly related to practical problems and more adaptable to imperfect and/or approximate specifications in real-life applications. As a result, they are now a central component of computer-science education and research.

But perhaps 'coming of age' is the wrong qualification for the success of a scientific endeavour. There will always be advances in mathematical logic, which is called 'formal methods' when tailored to specific computer science needs, leading to advances in reliable, safe and secure computing. Others have already noted 'the unusual effectiveness of [mathematical] logic' in computer science (Halpern *et al.* 2001; Davis 1988), while some have even gone as far as to tie the future of mathematical logic (or the 'best of it' according to some) to its ability to contribute to advances in computer science (Buss *et al.* 2001).

There are many research directions that will increase the impact of formal methods on computer science in significant and novel ways (Halpern *et al.* 2001; Buss *et al.* 2001). Focusing more narrowly on areas and domains of particular interest to us, we can propose at least two directions that, while complementary to the current state of the art, are separate from and beyond the topics covered in this special issue of the journal. We use the terms *integration* and *composability*, each in a specific context of our own, as shorthands to refer to these two directions, and to suggest relationships with cognate notions in other parts of computer science. We use the qualifier *seamless* to emphasise that the notions we refer to are distinct from other referents of these terms in the literature.

Fifteen years ago, E. M. Clarke and J. M. Wing wrote in a survey paper on formal methods (Clarke and Wing 1996) that, in order 'to attack harder and larger problems [...] further work needs to be done' in six 'fundamental areas', which they identified as:

(1) *composition*;
(2) *decomposition*;

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

(3) *abstraction*;
(4) *reusable models and theories*;
(5) *combinations of mathematical theories*; and
(6) *data structures and algorithms.*

These fundamental areas have lost none of their importance since 1996 and remain research priorities across almost all of computer science, and issues of *composability* and *integration* pervade them all.

In the next two sections, we outline two research challenges, one of composability and one of integration, each in a specific context drawn from our own recent research and teaching experience (Soule *et al.* 2011; Lapets and Kfoury 2010). We will try to clarify why the study and ultimate resolution of these challenges promise important breakthroughs in improving first the accessibility of formal methods and, ultimately, their applicability.

2. Composability and enforcing invariant properties in large-scale assemblies

Current practice in the design and implementation of large-scale, safety-critical systems follows a bottom-up approach: certification of desirable safety invariants of the full system must wait for the complete certification of the safety invariants of *all* its subsystems. For example, the development of real-time applications requires the use of real-time kernels so that timing properties at the application layer (top) can be established through knowledge and/or tweaking of much lower-level system details (bottom), such as worst-case execution or context-switching times (Deng and Liu 1997; Lim *et al.* 1995; Regehr 2002), specific scheduling and power parameters (Aydin *et al.* 2001; Pouwelse *et al.* 2001; Schmidt *et al.* 1998; Stankovic 2000), among many others.

While justifiable in some instances, this vertical approach does not lend itself well to emerging practices in the design of complex large-scale systems – namely, the assembling of various subsystems into a whole by system builders who may not possess the requisite expertise or knowledge of the internals of these subsystems (Krafzig *et al.* 2004). The latter alternative can be viewed as a *horizontal* and *incremental* approach to system design and implementation, which has significant advantages in terms of scalability and modularity. However, it also poses a major and largely unmet challenge with respect to verifiable trustworthiness – namely, how to certify formally that the system as a whole will satisfy specific safety invariants, and to determine formal conditions under which it will continue to do so as it is augmented, modified or subjected to local component failures.

To make our proposal more concrete, though still quite generic, we will express it in terms of the requirements for designing large-scale safety-critical interconnections of what we call *constrained-flow networks* (CFNs), each CFN being a producer, consumer or regulator of a *flow* of objects characterised by a set of variables and associated constraints that spell out what constitutes its safe operation. A larger CFN is obtained from assembling a large number of smaller CFNs.

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

2.1. *Requirement: strongly typed domain-specific languages*

An appropriate response to this challenge is to define a formal framework that, from its inception, incorporates three inter-related features:

(A) the ability to pursue system design and analysis without having to wait for missing (or broken) smaller CFNs to be inserted (or replaced);

(B) the ability to abstract away details through the retention of only the salient variables and constraints as we transition from smaller to larger CFNs; and

(C) the ability to leverage diverse, unrelated theories to derive properties of smaller CFNs, provided they share a common language at their interfaces.

Feature (A) dictates the presence of *holes* (or placeholders) in the required formalism to accommodate CFNs, provided they respect safety conditions at hole interfaces (that is, boundaries). Features (B) and (C) call for some sort of *type theory* that is suitable for the formulation of safety conditions at interfaces between CFNs and between CFNs and holes. The required type theory will need to enforce desirable safety properties as invariants across the full assembly of CFNs, and, moreover, must be *compositional* in the sense explained below.

Taking a cue from other parts of computer science, this amounts to the definition of a *strongly typed domain-specific language* (DSL). To be clear, we do not claim that a *single* strongly typed DSL will work equally well for all application domains; we anticipate the development of several such DSLs, each depending on one or more application domain and on the desired invariants to be enforced by the associated type theory, but all having features (A), (B) and (C).

We have made an initial attempt in this direction, where the components of large assemblies are CFNs regulated by linear constraints (Soule *et al.* 2011; Bestavros and Kfoury 2011; Kfoury 2011). However, far more research is called for: in different application domains and with different safety constraints on components and their interconnections.

2.2. *Seamless composability*

Feature (A) in these DSLs should allow system designers to interchange components, and assemble them in any order – this calls for a type theory that supports *compositional type inference*. Our use of the word *compositional* shares many attributes with identically named notions in other parts of computer science. However, *composability* and *modularity* are frequently conflated in the literature, but in our sense, they are distinct: compositionality implies modularity, but not *vice versa.*

A good example of the difference (and the conflation by some) is provided by type inference for the language Standard **ML**. The usual type inference for **ML**, as supported by the Hindley–Milner type system, is syntax directed, and thus modular, but it is not seamlessly *compositional* since it imposes the *order* in which different parts of a program are to be analysed. Indeed, polymorphism in **ML** is introduced by the **let** construct, the

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

typing of which is usually expressed by a rule of the form

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma, x : \forall \vec{\alpha}.\tau \vdash N : \tau'}{\Gamma \vdash (\textbf{let } x = M \textbf{ in } N) : \tau'}.$$

Type inference here follows a strict order:

(1) A type τ is inferred for program fragment M.
(2) If (1) succeeds, a type τ' is inferred for fragment N assuming that variable x is assigned a generalised type (or type scheme) $\forall \vec{\alpha}.\tau$.
(3) If (1) and (2) succeed, a type is inferred for fragment (**let** $x = M$ **in** N).

We would need a different type system for **ML** to support compositional type inference seamlessly, one which does not impose the order in which the different parts of a program are analysed. Such a type system is possible, though it is somewhat more complicated than the Hindley–Milner type system – it is based on the notion of *principal typing*, which is more powerful than the standard notion of a *principal type* for **ML** programs (Jim 1995; Kfoury and Wells 2004; Carlier *et al.* 2004; Ancona and Zucca 2004; Ancona *et al.* 2004).

Standard **ML** type inference is thus modular, but it is not seamlessly compositional. This conflation is also found elsewhere in the literature: for examples of analyses that are described as 'modular', but are not 'seamlessly compositional' in our sense, see Grumberg and Long (1991), Hsiung and Cheng (2003) and Li *et al.* (2005), among others.

3. Seamless integration of automated assistants for formal reasoning

Researchers have produced a large number of automated and semi-automated assistants for formal reasoning: parsers, evaluators, proof-authoring systems, software verification systems, interactive theorem provers and model-checkers, among scores of other tools. While there have been notable successes in bringing the power of these tools to bear on the development of safe and secure software and hardware, these leading-edge advances remain underutilised by practitioners. Two factors contribute to this state of affairs.

First, practitioners are frequently faced with a bewildering number of choices, which are not always mutually compatible or simultaneously usable. These range from the domain-specific (for example, the Chaff SAT solver[†]) to the generic (for example, the Isabelle/HOL theorem prover[‡]), and from the lightweight (for example, the Alloy[§] instance finder) to those requiring a relatively high level of training in formal methods (for example, the Coq proof assistant[¶]). Practitioners are not offered any standard formalisms, environments or tools enabling them to use multiple assistants either simultaneously or sequentially for different components, or in situations where only partial information about the system is available, or to compose the results and analyses produced by these components in predictable or safe ways.

[†] See http://www.princeton.edu/~chaff/software.html for details.
[‡] See *Isabelle* at http://www.cl.cam.ac.uk/research/hvg/isabelle/ for details.
[§] See *alloy: a language and tool for relational models* at http://alloy.mit.edu/alloy/ for details.
[¶] See *The Coq Proof Assistant* at http://coq.inria.fr/ for details.

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

Second, while these tools are built on many common concepts of mathematics and accepted norms of formal reasoning (for example, soundness and completeness, and the interplay between syntax and semantics), each requires mastery of its own syntactic conventions and applicability conditions, making their simultaneous and/or interactive use within the same application domain difficult, and even beyond many system builders' knowledge and training in formal methods. Furthermore, no infrastructures have been developed enabling domain experts, let alone end users, to assemble tool libraries that allow the tools to accommodate and conform more closely to the semantics of the practitioner's application domain.

3.1. *Degrees of integration*

The problems arising from the wide range of choices and the inaccessibility of the tools can be mitigated – the first by *integrating* separate components with one another, and the second by making the integration *seamless* with the processes and environments used by practitioners in their specific domains. Users can either be provided with automated help in managing any distinctions between integrated components, or the distinctions can be entirely hidden 'under the hood'.

When end users employ multiple automated assistants, it is usually within a single context in the form of the application domain they are working in. Thus, one way to approach the design of interfaces that support the integration of two or more automated assistants is to identify a subset of the end user's domain and create a mapping between that domain and the various features of the assistants in question. This mapping can have different degrees of automation and different degrees of formal rigour.

At one extreme, there is no integration. In many cases, the impediment to the simultaneous and/or interactive use of two or more automated assistants is actually independent of their mathematical properties. In order to use a particular automated assistant, end users must learn the new and unfamiliar collection of constructs (that is, the syntax) and associated semantics made available by the assistant. End users must then learn to translate the familiar concepts and structures from their own domain into this new representation, and must also learn to interpret its output within the context of their own domain. If the end users wish to employ multiple assistants when engaging in formal reasoning tasks (for example, they may want to use a *model checker* to generate counterexamples while exploring variants of a formal statement, and then a *proof verifier* to author a verifiably correct version of that statement), the effort is multiplied at least by the number of assistants they wish to employ (and, potentially, by the number of pair-wise combinations of those assistants).

At the other extreme, we could create a new formal system for the user's domain or a subset of it, with its own domain-appropriate semantics, together with *sound* and possibly *complete* automated mappings for that new formalism to and from subsets of two or more underlying automated assistants. Such an integration scheme would qualify as *seamless*: the user would not need to understand much about the underlying assistants, and the assistants would enable the user to perform relevant tasks automatically while working within the new all-encompassing formal system.

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

Between these two extremes would be an integration scheme that is still seamless and in which a common syntax is created for multiple systems, but no common formal semantics is defined. Such an intermediate integration is probably the most likely to succeed in practice, if only because it can be calibrated according to the end users' needs and training in formal methods.

3.2. *Lightweight integrated environments*

We use the term *integrated environment*, or *environment*, for an infrastructure that integrates multiple automated assistants. The degree of integration may fall anywhere along the spectrum described in the previous section. Somewhere along this spectrum lies a middle region that still qualifies as seamless, but trades off the costs of building a rigorous integration scheme requiring an all-encompassing formalism against the marginal benefits it provides.

What we call *lightweight* integration schemes will probably be found in this middle region. In this context, our use of the term *lightweight* in reference to the integration of automated assistants for formal reasoning reflects the motivation and usage of the term by other groups. The term *lightweight* has been introduced by some proponents who favour shifting the focus to more practical concerns, including end-user accessibility and partial domain applicability of automated assistants (Saiedian 1996; Woodcock *et al.* 2009). The challenge is to extend this *lightweight* notion to integrated *environments* of two or more underlying automated assistants.

A lightweight approach requires an end user to expend little effort in setting up or becoming familiar with the interface (Jackson 2006). Any integrated environment should be able to support incremental construction of partial (that is, underspecified or incomplete) formal arguments and models, and be able to provide early, relatively quick and fully formal feedback, even for partial arguments or models (Saiedian 1996; Woodcock *et al.* 2009). Such an environment should enable the use of the same seamless interface for complementary tasks (such as 'counterexample search' and 'proof verification'), which may use substantially different underlying automated assistants. Our own use of the term is extended to explicitly include some *accessible* or *user-friendly* environments possessing interfaces that use conventions, representations and semantics familiar to the target community of end users, and that provide proactive and interactive automated guidance to the end users and explains the capabilities of the components constituting that environment. However, we do not claim that a user-friendly environment must necessarily be lightweight; other ways of supporting accessibility may exist.

Many efforts are currently being undertaken to provide a more conventional and user-friendly interface for proof authoring and verification systems, although there is not much focus on integration. The need for natural interfaces, regardless of the underlying verification procedure, has been recognised to varying degrees within the context of formal proof authoring by the designers of the `Tutch` proof checker (Abel *et al.* 2001), the Scunak mathematical assistant system (Brown 2006), the ForTheL language and SAD proof assistant (Verchinine *et al.* 2008), the EPGY Theorem-Proving Environment

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

(McMath *et al.* 2001), the ΩMEGA proof verifier (Siekmann *et al.* 2002), and others. The MathLang project (Kamareddine and Wells 2008) represents a long-term effort that aims to make natural language an input method for mathematical arguments and proofs.

3.3. *Instantiation and evaluation*

Whether an integrated environment just provides a consistent syntax (with associated translations to different underlying automated assistants) or an entire formal system with its own semantics, its accessibility within a community of end users will depend on its ability to support a variety of familiar input and output formats, as well as semantic concepts within the end user's application domain. In many cases, assembling a sufficiently broad collection of formats and definitions is beyond the capabilities of a small group of expert designers; furthermore, any such assembled collection is likely to become out of date over time. Thus, it is essential that there exist appropriate infrastructures for collaboratively assembling and maintaining databases of syntactic constructs and definitions (or appropriate translations or relations) for each of the automated assistants within an integrated environment. On the end-user side of an integrated environment, large databases should be exposed using interactive interfaces that can guide the user in learning what constructs can be understood, and how choosing between alternative constructs can affect feasibility and performance.

Ultimately, efforts to address this problem should at least consider and learn from work in the area of general-purpose and common-sense ontology assembly and management (Panton *et al.* 2006; Chklovski 2005; Chklovski and Gil 2005; Liu and Singh 2004). For the end-user side, while an indexed database of syntactic idioms (and, more generally, logical propositions (Cairns and Gow 2007)) is a natural starting point, real-time keyword-based lookup techniques for programming environments developed by the *user-interface design* (UID) community (Han *et al.* 2009; Little and Miller 2007) provide a means for further improving the usability of a system. Search tools exist that allow users to browse Haskell expressions by type in a context (Katayama 2006) or library (Mitchell 2008). Another example is the automation proof assistant Matita (Asperti *et al.* 2007), which is heavily based on an integrated search engine.

In order to avoid inadvertently making unfounded assumptions about the fitness and usefulness of syntaxes, graphical interfaces and underlying formal structures for a community of end users, it is essential that they be evaluated using some sort of disciplined, or even empirical, methodology. There have been attempts along these lines in the context of programming language design (Hanenberg 2010a; Hanenberg 2009). These were motivated by a perceived lack of attention in the software engineering community to the empirical study of usability issues (Hanenberg 2010b). Trials with actual school-age students were conducted when evaluating the EPGY Theorem-Proving Environment (McMath *et al.* 2001). Similar considerations must also be made in the design of automated systems and integrated environments. Work by the *human-computer interaction* (HCI) and UID communities should also be taken into consideration and used in evaluating the accessibility of systems.

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

References

Abel, A., Chang, B. and Pfenning, F. (2001) Human-readable machine-verifiable proofs for teaching constructive logic. In: Egly, U., Fiedler, A., Horacek, H. and Schmitt, S. (eds.) *PTP '01: IJCAR Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs*, Siena, Italy.

Ancona, D. and Zucca, E. (2004) Principal typings for Java-like languages. In: *POPL '04: Proceedings of the 31st ACM SIGPLAN–SIGACT symposium on Principles of Programming Languages*, ACM 306–317.

Ancona, D., Damiani, F., Drossopoulou, S. and Zucca, E. (2004) Even more principal typings for Java-like languages. In: *ECOOP Workshop on Formal Techniques for Java Programs.*

Asperti, A., Coen, C. S., Tassi, E. and Zacchiroli, S. (2007) User interaction with the Matita proof assistant. *Journal of Automated Reasoning* **39** (2) 109–139.

Aydin, H., Melhem, R. and Moss, D. (2001) Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In: *Proceedings of EuroMicro Conference on Real-Time Systems* 225–232.

Bestavros, A. and Kfoury, A. (2011) A Domain-Specific Language for Incremental and Modular Design of Large-Scale Verifiably-Safe Flow Networks. In: Proceedings of IFIP Working Conference on Domain-Specific Languages (DSL 2011). *Electronic Proceedings in Theoretical Computer Science* **66** 24–47.

Brown, C. E. (2006) Verifying and Invalidating Textbook Proofs using Scunak. In: Borwein, J. M. and Farmer, W. M. (eds.) Mathematical Knowledge Management: Proceedings 5th International Conference, MKM 2006. *Springer-Verlag Lecture Notes in Computer Science* **4108** 110–123.

Buss, S. R., Kechris, A. S., Pillay, A. and Shore, R. A. (2001) The prospects for mathematical logic in the twenty-first century. *The Bulletin of Symbolic Logic* **7** (2) 169–196.

Cairns, P. and Gow, J. (2007) Integrating Searching and Authoring in Mizar. *Journal of Automated Reasoning* **39** (2) 141–160.

Carlier, S., Polakow, J., Wells, J. B. and Kfoury, A. J. (2004) System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In: Schmidt, D. (ed.) Programming Languages and Systems: 13th European Symposium on Programming, ESOP 2004. *Springer-Verlag Lecture Notes in Computer Science* **2986** 294–309.

Chklovski, T. (2005) Towards managing knowledge collection from volunteer contributors. In: *Proceedings of AAAI Spring Symposium on Knowledge Collection from Volunteer Contributors (KCVC05)*, AAAI.

Chklovski, T. and Gil, Y. (2005) Improving the design of intelligent acquisition interfaces for collecting world knowledge from web contributors. In: *K-CAP '05: Proceedings of the 3rd international conference on Knowledge capture*, ACM 35–42.

Clarke, E. M. and Wing, J. M. (1996) Formal methods: State of the art and future directions. *ACM Computing Surveys* **28** (4).

Davis, M. (1988) Influences of mathematical logic on computer science. In: Herkin, R. (ed.) *The Universal Turing Machine: A Half-Century Survey*, Oxford University Press 315–326.

Deng, Z. and Liu, J. W.-S. (1997) Scheduling real-time applications in an open environment. In: *Proceedings of the 18th IEEE Real-Time Systems Symposium*, IEEE Computer Society Press 308–319.

Grumberg, O. and Long, D. E. (1991) Model checking and modular verification. *ACM Transactions on Programming Languages and Systems* **16** (3) 843–871.

Halpern, J. Y., Harper, R., Immerman, N., Kolaitis, P. G., Vardi, M. Y. and Vianu, V. (2001) On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic* **7** (2) 213–236.

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

Han, S., Wallace, D. R. and Miller, R. C. (2009) Code completion from abbreviated input. In: *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society 332–343.

Hanenberg, S. (2009) Costs of using untyped programming languages – first empirical results. In: *13th IFAC Symposium on Information Control Problems in Manufacturing (Track Advanced Software Engineering)*, Moscow, Russia.

Hanenberg, S. (2010a) Doubts about the positive impact of static type systems on programming tasks in single developer projects – an empirical study. In: D'Hondt, T. (ed.) ECOOP 2010 – Object-Oriented Programming: Proceedings 24th European Conference. *Springer-Verlag Lecture Notes in Computer Science* **6183** 300–303.

Hanenberg, S. (2010b) Faith, hope, and love – a criticism of software science's carelessness with regard to the human factor. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA/SPLASH)* 933–946.

Hsiung, P.-A. and Cheng, S.-Y. (2003) Automating formal modular verification of asynchronous real-time embedded systems. In: *VLSI Design, 2003, Proceedings, 16th International Conference* 249–254.

Jackson, D. (2006) *Software Abstractions: Logic, Language, and Analysis*, The MIT Press.

Jim, T. (1995) What are principal typings and what are they good for? Technical Memorandum MIT LCS TM 532.

Kamareddine, F. and Wells, J. B. (2008) Computerizing Mathematical Text with MathLang. *Electronic Notes in Theoretical Computer Science* **205** 5–30.

Katayama, S. (2006) Library for systematic search for expressions. In *AIC '06: Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications*, World Scientific and Engineering Academy and Society (WSEAS) 381–387.

Kfoury, A. (2011) The Denotational, Operational, and Static Semantics of a Domain-Specific Language for the Design of Flow Networks. In: *Proceedings of SBLP 2011: Brazilian Symposium on Programming Languages.*

Kfoury, A. J. and Wells, J. B. (2004) Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science* **311** (1-3) 1–70.

Krafzig, D., Banke, K. and Slama, D. (2004) *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*, Prentice Hall.

Lapets, A. and Kfoury, A. (2010) A User-friendly Interface for a Lightweight Verification System. In: Proceedings of UITP'10: 9th International Workshop On User Interfaces for Theorem Provers. *Electronic Notes in Theoretical Computer Science* **285** 29–41.

Li, H. C., Krishnamurthi, S. and Fisler, K. (2005) Modular verification of open features using three-valued model checking. *Automated Software Engineering* **12** (3) 349–382.

Lim, S.-S. *et al.* (1995) An accurate worst case timing analysis for risc processors. In: *Proceedings of the 15th IEEE real-time systems symposium* 97–108.

Little, G. and Miller, R. C. (2007) Keyword programming in Java. In: *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ACM 84–93.

Liu, H. and Singh, P. (2004) Conceptnet – a practical commonsense reasoning tool-kit. *BT Technology Journal* **22** (4) 211–226.

McMath, D., Rozenfeld, M. and Sommer, R. (2001) A Computer Environment for Writing Ordinary Mathematical Proofs. In: Nieuwenhuis, R. and Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning: Proceedings 8th International Conference, LPAR 2001. *Springer-Verlag Lecture Notes in Computer Science* **2250** 507–516.

Mitchell, N. (2008) Hoogle overview. *The Monad.Reader* **12** 27–35.

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press

Panton, K. *et al.* (2006) Common Sense Reasoning – From Cyc to Intelligent Assistant. In: Cai, Y. and Abascal, J. (eds.) Ambient Intelligence in Everyday Life. *Springer-Verlag Lecture Notes in Computer Science* **3864** 1–31.

Pouwelse, J., Langendoen, K. and Sips, H. (2001) Dynamic voltage scaling on a low-power microprocessor. In: *Mobile Computing and Networking – Mobicom* 251–259.

Regehr, J. (2002) Inferring scheduling behavior with Hourglass. In: *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, USENIX Association 143–156.

Saiedian, H. (ed.) (1996) An invitation to formal methods. A 'roundtable' of short articles by several authors. *IEEE Computer* **29** (4) 16–30.

Schmidt, D. C., Levine, D. L. and Mungee, S. (1998) The design of the TAO real-time object request broker. *Computer Communications* **21** 294–324.

Siekmann, J. H., Benzmüller, C., Fiedler, A., Meier, A. and Pollet, M. (2002) Proof Development with OMEGA: sqrt(2) Is Irrational. In: Baaz, M. and Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning 9th International Conference, LPAR 2002. *Springer-Verlag Lecture Notes in Computer Science* **2514** 367–387.

Soule, N., Bestavros, A., Kfoury, A. and Lapets, A. (2011) Safe Compositional Equation-based Modeling of Constrained Flow Networks. In: Cellier, F. E., Broman, D., Fritzson, P. and Lee, E. A. (eds.) *Proceedings of EOOLT 2011: 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Linköping University Electronic Press.

Stankovic, J. A. (2000) Vest: A toolset for constructing and analyzing component based embedded systems. In: Henzinger, T. A. and Kirsch, C. M. (eds.) Embedded Software: Proceedings First International Workshop, EMSOFT 2001. *Springer-Verlag Lecture Notes in Computer Science* **2211** 390–402.

Verchinine, K., Lyaletski, A., Paskevich, A. and Anisimov, A. (2008) On Correctness of Mathematical Texts from a Logical and Practical Point of View. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M. and Wiedijk, F. (eds.) Intelligent Computer Mathematics – Proceedings: 9th International Conference, AISC 2008; 15th Symposium, Calculemus 2008; and 7th International Conference, MKM 2008. *Springer-Verlag Lecture Notes in Computer Science* **5144** 583–598.

Woodcock, J., Larsen, P. G., Bicarregui, J. and Fitzgerald, J. (2009) Formal methods: Practice and experience. *ACM Computing Surveys* **41** (4) 19:1–19:36.

https://doi.org/10.1017/S0960129512000072 Published online by Cambridge University Press