Technical Report BUCS-TR-2012-001, Computer Science Department, Boston University

Seamless Composition and Integration: A Perspective on Formal Methods Research †

AZER BESTAVROS[1], ASSAF KFOURY[1] and ANDREI LAPETS[1]
[1] *Computer Science Dept., Boston University, Boston, Massachusetts 02215, USA.*

17 February 2012

Have formal methods in computer science come of age? While the contributions to this volume 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. 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 endeavor. There will always be advances in mathematical logic – a.k.a. 'formal methods' among computer scientists – 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 others have gone as far as tying 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 promote 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 which, while complementary to the current state-of-the-art, are separate from and beyond the topics covered in this volume. 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 emphasize that the notions to which we refer are distinct from other referents of these terms in the literature.

In a survey paper on formal methods 15 years ago (Clarke and Wing, 1996), E.M. Clarke and J.M. Wing wrote that, in order "to attack harder and larger problems [...]

† To appear in *Math. Struct. in Comp. Science.*
This work is supported in part by NSF award CCF-0820138.

further work needs to be done" in six "fundamental areas" which they identified thus: (1) *composition*, (2) *decomposition*, (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, 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., 2011a; Lapets and Kfoury, 2010). We try to clarify why the study and ultimate resolution of these two challenges hold the promise of important breakthroughs in the accessability of formal methods and, ultimately, their applicability.

1. Composability and Enforcing Invariant Properties in Large-Scale Assemblies

Current practices in the design and implementation of large-scale, safety-critical systems follow a bottom-up approach: certification of desirable safety invariants of the full system must wait complete certification of the safety invariants of *all* subsystems. For example, the development of real-time applications necessitates 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 s. 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). This latter alternative can be viewed as a *horizontal* and *incremental* approach to system design and implementation, which has significant merits with respect to scalability and modularity. However, it also poses a major and largely unmet challenge with respect to verifiable trustworthiness – namely, how to formally certify that the system as a whole will satisfy specific safety invariants and to determine formal conditions under which it will remain so, as it is augmented, modified, or subjected to local component failures.

To make our proposal more concrete, though still quite generic, we express it in terms of the requirements for designing large-scale safety-critical interconnections of what we may call *constrained-flow networks* (CFNs), each CFN being a producer, consumer, or regulator, of a *flow* of objects characterized 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.

Needed: Strongly-Typed Domain-Specific Languages. An appropriate response to this challenge is to define a formal framework which, 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, as long as they share a common language at their interfaces.

Feature (A) dictates the presence of *holes* (or placeholders) in the sought-after formalism, where CFNs can be placed as long as they respect safety conditions at hole interfaces (*i.e.*, boundaries). Features (B) and (C) call for a *type theory* of sorts, suitable for the formulation of safety conditions at interfaces between CFNs and between CFNs and holes. The required type theory is to enforce desirable safety properties as invariants across the full assembly of CFNs, which is moreover *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 sharing features (A), (B) and (C).

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

Seamless Composability. In relation to feature (A) in the sought-after DSLs, system designers should be able to interchange components and assemble them in any order, which in turn calls for a type theory that supports *compositional type inference*. What we call *compositional* shares many attributes with identically-named notions in other parts of computer science. In the literature, *composability* and *modularity* are frequently conflated. In our sense, the two are distinct: what is compositional is modular, 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**, supported by its so-called Hindley-Milner type system, is syntax-directed and therefore modular; it is not seamlessly *compositional*, because it imposes the *order* in which different parts of a program are analyzed. Indeed, polymorphism in **ML** is introduced by the **let** construct, the 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 generalized type (or type scheme) $\forall \vec{\alpha}.\tau$; and (3), if (1) and (2) succeed, a type is inferred for fragment $(\textbf{let } x = M \textbf{ in } N)$.

A different type system for **ML** is necessary to support compositional type inference seamlessly, one which does not impose the order in which the different parts of a program are analyzed. Such a type system, somewhat more complicated than the Hindely-Milner, is possible based on the notion of *principal typing* – more powerful than the standard notion of *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 therefore modular but not seamlessly compositional. The conflation is found elsewhere in the literature. For examples of analyzes that are qualified as "modular", though they are "seamlessly compositional" in our sense, see (Grumberg and Long, 1991; Hsiung and Cheng, 2003; Li et al., 2005), among others.

2. Seamless Integration of Automated Assistants for Formal Reasoning

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

First, practitioners are faced with a bewildering range of choices of not-always mutually compatible and not-always simultaneously usable alternatives – from domain-specific (*e.g.*, the Chaff SAT solver ("The Chaff SAT Solver," n.d.)) to generic (*e.g.*, the Isabelle/HOL theorem prover ("Isabelle: A Generic Proof Assistant," n.d.)), from lightweight (*e.g.*, the Alloy instance finder ("Alloy," n.d.; Jackson, 2006)) to requiring a relatively high level of training in formal methods (*e.g.*, the Coq proof assistant ("The Coq Proof Assistant," n.d.)). Practitioners are offered no standard formalisms, environments, or tools that would enable them to utilize multiple assistants simultaneously or across time for different components, use these assistants in situations where only partial information about the system is available, and compose the results and analyses produced by these components in predictable or safe ways.

Second, while these tools are built on many common concepts of mathematics and accepted norms of formal reasoning (*e.g.*, soundness and completeness, interplay between syntax and semantics, etc.), each requires mastery of its own syntactic conventions and applicability conditions, making their simultaneous and/or interactive use within a same application domain difficult, if not beyond many system builders' knowledge and training in formal methods. Furthermore, no infrastructures are provided that would allow even domain experts to assemble libraries for the tools that allow the tools to accommodate and conform more closely to the semantics of the practitioner's application domain.

Degrees of Integration. The myriad of choices and inaccessibility facing end-users 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 utilized by practitioners. Users are either provided automated aid in managing any distinctions between integrated components, or the distinctions are entirely hidden "under the hood".

When an end-user employs multiple automated assistants, this usually occurs within a single context, the application domain within which the end-user is working. Thus, one way to approach the design of interfaces that support the integration of two or more automated assistants is by identifying a subset of the end-user's domain and creating 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 rigor.

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 (*i.e.*, the syntax) and associated semantics made available by the assistant. End-users must then learn to translate the familiar concepts and structures in 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 (*e.g.*, 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, it is possible to create a new formal system for the user's domain or a subset thereof, with its own domain-appropriate semantics, and to create *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; the assistants would merely allow the user to perform certain tasks automatically while working within the new all-encompassing formal system.

An example of an integration scheme between these two extremes that is still seamless is one in which a common syntax is created for multiple systems, while no common formal semantics is defined. Such an intermediate integration is most likely to succeed in practice, if only because it can be calibrated according to end-users' needs and training in formal methods.

Lightweight Integrated Environments. We call an *integrated environment*, or *environment*, an infrastructure that integrates multiple automated assistants. The degree of integration may fall anywhere along the spectrum we described above. 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 thereof.

What we call *lightweight* integration schemes are likely found in this middle region. In this context, our use of the term *lightweight* in reference to integration of automated assistants for formal reasoning reflects motivation and usage of the term by other groups. The term *lightweight* has been introduced by some proponents who favor shifting the

focus to more practical concerns, including end-user accessibility and partial domain applicability of automated assistants (Saiedian (Ed.), 1996; Woodcock et al., 2009). The challenge is to extend this notion of *lightweight* 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 (*i.e.*, underspecified or incomplete) formal arguments and models, and is able to provide early, relatively quick, and fully formal feedback even for partial arguments or models (Saiedian (Ed.), 1996; Woodcock et al., 2009). Such an environment makes it possible to use the same seamless interface to engage in complementary tasks (*e.g.*, 'counterexample search' and 'proof verification') that may use substantially different underlying automated assistants. Our own usage 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 that explicates the capabilities of the components constituting that environment. However, note that 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 recognized to varying degrees within the context of formal proof authoring by the designers of the `Tutch` proof checker (Abel, Chang, and Pfenning, 2001), the Scunak mathematical assistant system (Brown, 2006), the ForTheL language and SAD proof assistant (Verchinine et al., 2008), the EPGY Theorem-Proving Environment (McMath et al., 2001), the ΩMEGA proof verifier (Siekmann et al., 2002), and others. The MathLang project (Kamareddine and Wells, 2008) represented a long-term effort that aims to make natural language an input method for mathematical arguments and proofs.

Instantiation and Evaluation. Whether an integrated environment provides nothing but a consistent syntax (with associated translations to different underlying automated assistants) or an entire formal system with its own semantics, its accessability within a community of end-users will hinge 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 collection of expert designers; furthermore, any such assembled collections are 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). Matita (Asperti et al., 2007) is a proof assistant the automation of 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 to evaluate these using some sort of disciplined or even empirical methodology. Attempts exist to take this sort of approach within the context of programming language design (Hanenberg, 2010b; Hanenberg, 2009), motivated by a perceived lack of attention in the software engineering community to the empirical study of usability issues (Hanenberg, 2010a). 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 be taken into consideration and utilized in evaluating the accessibility of systems.

References

Alloy: A Language and Tool for Relational Models. URL: `http://alloy.mit.edu/alloy/`.

The Chaff SAT Solver. URL: `http://www.princeton.edu/ chaff/software.html`.

The Coq Proof Assistant. URL: `http://coq.inria.fr/`.

Isabelle: A Generic Proof Assistant. URL: `http://www.cl.cam.ac.uk/research/hvg/isabelle/`.

A. Abel, B. Chang, and F. Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic. In U. Egly, A. Fiedler, H. Horacek, and S. Schmitt, editors, *PTP '01: IJCAR Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs*, Siena, Italy, 2001. URL: `citeseer.ist.psu.edu/abel01humanreadable.html`.

D. Ancona and E. Zucca. Principal typings for java-like languages. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 306–317, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. URL: `http://doi.acm.org/10.1145/964001.964027`.

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

A. Asperti, C. S. Coen, E. Tassi, and S. Zacchiroli. User interaction with the matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007. ISSN 0168-7433. URL: `http://dx.doi.org/10.1007/s10817-007-9070-5`.

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

A. Bestavros and A. Kfoury. A Domain-Specific Language for Incremental and Modular Design of Large-Scale Verifiably-Safe Flow Networks. In *Proc. of IFIP Working Conference on Domain-Specific Languages (DSL 2011), EPTCS Volume 66*, pages 24–47, Sept 2011.

C. E. Brown. Verifying and Invalidating Textbook Proofs using Scunak. In *MKM '06: Mathematical Knowledge Management*, pages 110–123, Wokingham, England, 2006.

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

P. Cairns and J. Gow. Integrating Searching and Authoring in Mizar. *Journal of Automated Reasoning*, 39(2):141–160, 2007. ISSN 0168-7433.

S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, volume 2986 of *LNCS*, pages 294–309. Springer-Verlag, 2004. ISBN 3-540-21313-9.

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

T. Chklovski and Y. Gil. 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*, pages 35–42, New York, NY, USA, 2005. ACM. ISBN 1-59593-163-5. URL: `http://doi.acm.org/10.1145/1088622.1088630`.

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

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

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

O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16, 1991.

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

S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4. URL: `http://dx.doi.org/10.1109/ASE.2009.64`.

S. Hanenberg. 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, June 3-5 2009.

S. Hanenberg. Faith, hope, and love - a criticism of software science's carelessness with regard to the human factor. In *Proceedings of OOPSLA/SPLASH*, Reno, Nevada, USA, 2010a.

S. Hanenberg. Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In *24th European Conference on Object-Oriented Programming (ECOOP)*, Maribor, Svlovenia, June 21-25 2010b.

P.-A. Hsiung and S.-Y. Cheng. Automating formal modular verification of asynchronous real-time embedded systems. *VLSI Design, International Conference on*, 0:249, 2003. ISSN 1063-9667.

D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN 0262101149.

T. Jim. What are principal typings and what are they good for? Tech. memo. MIT/LCS/TM-532, MIT, 1995.

F. Kamareddine and J. B. Wells. Computerizing Mathematical Text with MathLang. *Electronic Notes in Theoretical Computer Science*, 205:5–30, 2008. ISSN 1571-0661.

S. Katayama. Library for systematic search for expressions. In *AIC '06: Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications*, pages 381–387, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS). ISBN 960-8457-51-3.

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

A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004. URL: doi:10.1016/j.tcs.2003.10.032.

D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. ISBN 0131465759.

A. Lapets and A. Kfoury. A User-friendly Interface for a Lightweight Verification System. In *Proceedings of UITP'10: 9th International Workshop On User Interfaces for Theorem Provers*, Edinburgh, UK, July 2010.

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

S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An accurate worst case timing analysis for risc processors. In *IN IEEE REAL-TIME SYSTEMS SYMPOSIUM*, pages 97–108, 1995.

G. Little and R. C. Miller. Keyword programming in java. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 84–93, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. URL: http://doi.acm.org/10.1145/1321631.1321646.

H. Liu and P. Singh. Conceptnet — a practical commonsense reasoning tool-kit. *BT Technology Journal*, 22(4):211–226, 2004. ISSN 1358-3948. URL: http://dx.doi.org/10.1023/B:BTTJ.0000047600.45421.6d.

D. McMath, M. Rozenfeld, and R. Sommer. A Computer Environment for Writing Ordinary Mathematical Proofs. In *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*, pages 507–516, London, UK, 2001. Springer-Verlag. ISBN 3-540-42957-3.

N. Mitchell. Hoogle overview. *The Monad.Reader*, 12:27–35, November 2008.

K. Panton, C. Matuszek, D. Lenat, D. Schneider, M. Witbrock, N. Siegel, and B. Shepard. Common Sense Reasoning – From Cyc to Intelligent Assistant. In Y. Cai and J. Abascal, editors, *Ambient Intelligence in Everyday Life*, volume 3864 of *LNAI*, pages 1–31. Springer, 2006.

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

J. Regehr. Inferring scheduling behavior with hourglass. In *In Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pages 143–156, 2002.

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

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

J. H. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Proof Development with OMEGA: sqrt(2) Is Irrational. In *LPAR*, pages 367–387, 2002.

N. Soule, A. Bestavros, A. Kfoury, and A. Lapets. Safe Compositional Equation-based Modeling of Constrained Flow Networks. In *Proceedings of EOOLT 2011: 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Zurich, Switzerland, September 2011a.

N. Soule, A. Bestavros, A. Kfoury, and A. Lapets. Safe Compositional Equation-based Modeling of Constrained Flow Networks. In *Proc. of 4th Int'l Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Zürich, September 2011b.

J. A. Stankovic. Vest: A toolset for constructing and analyzing component based embedded systems. In *In Proc. EMSOFT01, LNCS 2211*, pages 390–402. Springer, 2000.

K. Verchinine, A. Lyaletski, A. Paskevich, and A. Anisimov. On Correctness of Mathematical Texts from a Logical and Practical Point of View. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 583–598, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85109-7. URL: `http://dx.doi.org/10.1007/978-3-540-85110-3_47`.

J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41:19:1–19:36, October 2009. ISSN 0360-0300. URL: `http://doi.acm.org/10.1145/1592434.1592436`.