Towards Accessible Integration and Deployment of Formal Tools and Techniques

Andrei Lapets and Rick Skowyra and Azer Bestavros and Assaf Kfoury
Computer Science Dept.
Boston University
Boston, Massachusetts 02215
Email: lapets@bu.edu, skowyra@bu.edu, best@bu.edu, kfoury@bu.edu

***Abstract*—Computer science researchers in the programming languages and formal verification communities, among others, have produced a variety of automated assistance and verification tools and techniques for formal reasoning. While there have been notable successes in utilizing these tools on the development of safe and secure software and hardware, these leading-edge advances remain largely underutilized by large populations of potential users that may benefit from them. In particular, we consider researchers, instructors, students, and other end users that may benefit from instant feedback from lightweight modeling and verification capabilities when exploring system designs or formal arguments.**

We describe Aartifact, a supporting infrastructure that makes it possible to quickly and easily assemble interacting collections of small domain-specific languages, as well as translations between those languages and existing tools (e.g., Alloy, SPIN, Z3) and techniques (e.g., evaluation, type checking, congruence closure); the infrastructure also makes it possible to compile and deploy these translators in the form of a cloud-based web application with an interface that runs inside a standard browser. This makes more manageable the process of exposing a limited, domain-specific, and logistically accessible subset of the capabilities of existing tools and techniques to end users. This infrastructure can be viewed as a collection of modules for defining interfaces that turn third-party formal modeling and verification tools and techniques into plug-ins that can be integrated within web-based interactive formal reasoning environments.

I. Introduction

Computer science researchers in the programming languages and formal verification communities, among others, have produced a variety of automated assistance and verification tools and techniques for formal reasoning: parsers, evaluators, proof-authoring systems, software verification systems, interactive theorem provers, model-checkers, static analysis methods, and so on. While there have been notable successes in utilizing these tools on the development of safe and secure software and hardware, these leading-edge advances remain largely underutilized by large populations of potential users that may benefit from them, including researchers, instructors, and students working in particular application domains. The limited use of formal tools and techniques in particular application domains has been acknowledged and sometimes addressed (e.g., in the design of distributed systems [1], [2], software engineering and programming [3], and real-time systems [4]).

More broadly, we view the issues as falling into the categories of domain-specific *accessibility* and *integration*: (1) different tools and techniques require different platforms and infrastructures, and have different syntaxes and semantics that must be adapted and understood within particular application domains; (2) practical characteristics make certain tools and techniques more appropriate for some domain-specific tasks than others, and many common, even simple tasks may require the simultaneous or even integrated use of multiple tools and techniques. The need for accessible integration of existing tools and techniques has been recognized in general [5]–[8] as well as within the context of a variety of application domains [9]–[12].

Practical integration of existing tools and techniques for application domains does not require only sound theoretical underpinnings in the form of formal frameworks that can encompass (or provide interface principles for) existing tools and techniques. Successful dissemination of integrated systems will require the creation of infrastructures for implementing and deploying domain-specific integrated environments that incorporate existing tools and techniques (as noted in more ambitious proposals along these lines [8]). In fact, lowering the bar for creating integrated environments may more immediately contribute to the accessibility of existing tools and techniques, which may in turn encourage more widespread utilization thereof. Such an infrastructure would also make it easier for tool designers to deliver existing or novel tools to target communities by providing a back-end interface for integrated environments to which they can connect their tools.

It is worth noting that the work presented in this paper does not constitute a complete or universal infrastructure for assembling integrated environments of tools; the authors do not view a universal framework or interface standard as a goal that is practical, attainable, or sustainable in the short term. Researchers specialize and work within particular application domains, and this naturally leads to the continual creation of new and distinct vocabularies, standards, languages, representations, tools, and techniques. Thus, the adopted assumption is that new tools and techniques with incompatible input/output representations and interfaces will inevitably be created in the future; in this work we present a prototype infrastructure designed to make assembly and deployment of integrated environments less burdensome under these circumstances.

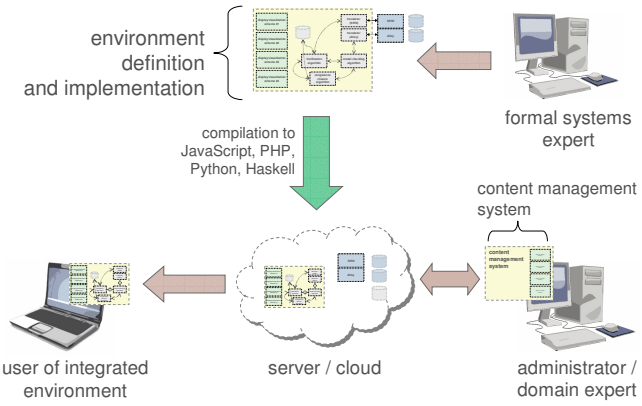

Fig. 1. Overall organization of the infrastructure.

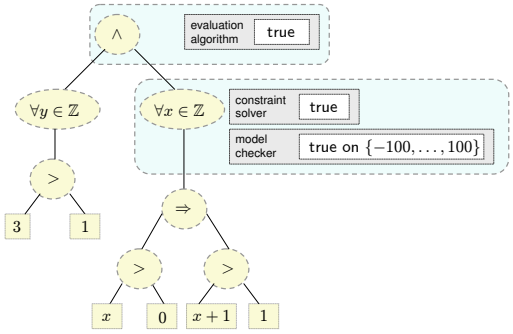

Fig. 2. An illustration of an abstract syntax tree corresponding to the logical formula "$(\forall y \in \mathbb{Z},\ 3 > 1) \wedge (\forall x \in \mathbb{Z},\ (x > 0 \ \Rightarrow\ x + 1 > 1))$" with multiple values (produced by different components) for some subexpressions.

II. Infrastructure for Instantiating Accessible Integrated Environments

To support rapid assembly and deployment of potential domain-specific solutions the designs of which take into account the issues raised in Section I, we introduce Aartifact [13], an infrastructure for implementing, instantiating, and delivering over the web an accessible integrated formal reasoning environment. This infrastructure is comprised of a collection of components that support the tasks that must be performed by three possible user roles (actual users may have more than one role): (1) formal systems experts who implement automated formal verification and analysis algorithms or techniques, as well as translators for underlying formal tools; (2) application domain expert administrators who instantiate libraries, decide which components and libraries are available to end users in the environment at any given time, and author content that may put into context the tasks in which the end-user may engage (e.g., homework assignments, tutorials, documentation, and so on); and (3) end users who use the environment to engage in formal reasoning tasks. The overall organization is presented in Figure 1.

A. Formal Systems Experts and Component Implementation

It is the responsibility of formal systems experts to provide implementations of common formal analysis and verification algorithms (e.g., evaluation, monomorphic type checking, congruence closure computation, resolution, unification, and so on), as well as appropriate translations for external systems or components (e.g., translations from a particular syntax for network protocols and protocol properties specified in a particular logic to an appropriate Alloy, SPIN, PRISM, or Z3 syntax). The role of the formal systems expert is somewhat similar to that of a staff member of a "tool repository" envisioned in more long-term proposals for integration efforts [8]. In order to support formal systems experts in this task, it is necessary to provide a language that: (1) allows them to easily specify a *single* definition for syntactic expressions (formulas and terms) for a particular domain-specific environment's input language without having to also define an abstract syntax and implement parsers that transform the concrete syntax into an abstract syntax and pretty-printers for displaying the abstract syntax or emitting it as input to underlying or integrated back-end tools; (2) allows them to specify algorithms using concise pattern-matching syntax (adopted from functional languages such as ML and Haskell) that operate on the abstract syntax; and (3) allows them to define and manage a collection of algorithms that are interdependent and can invoke one another.

Aartifact depends on a simple, specialized, custom-built programming language, Informl [14], that can be compiled to Haskell, Python, PHP, and JavaScript. A parser generator implemented using Informl makes it possible to concisely define parsers that can be compiled to any of these target platforms and languages. This makes it possible to easily define small domain-specific languages. Informl's features, including pattern-matching syntax, make it possible to define concise translations of abstract syntax instances generated by the parser.

For a given implementation of a domain-specific environment that employs the Aartifact infrastructure, all component algorithms and translations must be transformations that are defined on a subset of the input language supported by the environment. Any or all of the algorithms can then be applied to all subexpressions of any expression tree parsed from the formal input provided by the end user. Each subexpression can then be annotated with the results of applying various components to that subexpression, as illustrated in Figure 2. Environment implementers can then combine various Aartifact interface features to display this information to users in different ways (including formats that are accompanied by widgets that allow users to explore or filter the results).

Because the definition of any component algorithm is allowed to invoke any other algorithm, it is possible to construct a dependency graph between component algorithms, such as the one illustrated in Figure 3. If the dependency graph is a directed acyclic graph, it is possible to provide an ordering for algorithms that ensures convergence. Otherwise, it is necessary to either allow the algorithms to iterate until convergence or specify a bound on the number of iterations.

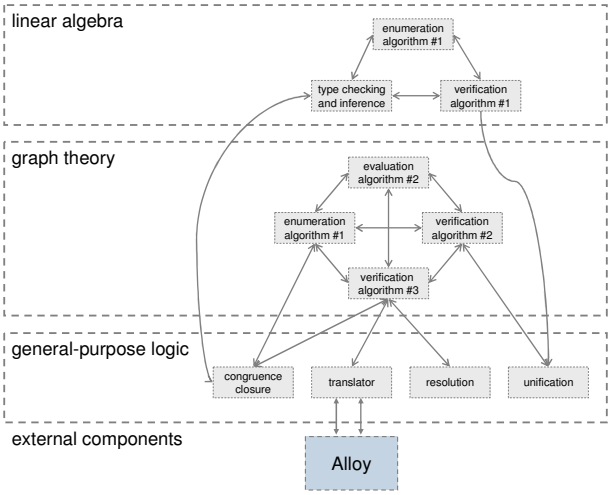

Fig. 3. Example of a component algorithm dependency graph.

B. Application Domain Experts and Instantiation

It is the job of the application domain expert administrator to instantiate a library of formal facts that will be seen by the user (and any corresponding definitions that are appropriate for underlying or integrated tools and techniques). Off-the-shelf open source *content management systems* (CMS) such as Drupal and MediaWiki can be extended to support these tasks. The current state of these features has been addressed in earlier work [15], [16].

C. End users and a Web-based Interactive Environment

End users can use any standard browser capable of running JavaScript applications to run an integrated environment constructed using the components of the Aartifact infrastructure. From the end-user's perspective, the environment provides a way to input formal definitions, arguments, and/or algorithms using a domain-specific syntax. Within a particular environment instantiation, the functionality provided by the integrated components is exposed using a variety of JavaScript visualization widgets that include: (1) friendly, formatted output of the formal argument, including highlighting of syntactic and logical errors, or other properties derived through automated analysis; (2) lists of formal expressions, including the propositions available in a library, facts derived by one or more component algorithms, and translations of the input into syntaxes for particular underlying systems and tools; (3) interactive controls for starting and stopping inference, evaluation, enumeration, and verification components that have been implemented to allow reporting of partial results.

III. Instantiating Integrated Environments

Assembling and instantiating a domain-specific integrated environment using the Aartifact infrastructure (or components thereof) usually involves at least the following steps:

(1) installation of the Aartifact and Informl modules and skeleton files on a web server (that has at least PHP or Python installed if any server-side tools are being integrated);
(2) specification of a grammar for the syntax of the domain-specific input language (using a format supported by the Informl modules that automatically generate parsers, abstract syntax data structure definitions, and pretty-printers);
(3) installation of one or more underlying tools on the server (unless they are available as web services elsewhere on the web);
(4) implementation (using Informl) of one or more techniques (i.e., algorithms) that operate on the syntax specified in step (2) above, or translations from the syntax specified in step (2) above into either API calls or to the actual concrete syntax of each underlying tool (in most cases, API calls that can be made in underlying languages such as Python could be made in Informl implementations with little or no modification);
(5) implementation (using Informl) of translations from tool outputs into domain-specific output, if it is necessary.

The Aartifact and Informl modules are designed to make steps (2), (4), and (5) above more manageable by requiring very little redundant work, and by providing abstractions such as algebraic data types and pattern matching over algebraic data types, features that are drawn from typed functional programming languages.

Components of the Aartifact infrastructure have been employed in assembling accessible integrated environments for two application domains: classroom instruction [17] and design of distributed systems [18]. A larger collection of tools, including SMT solvers [19], have also been integrated to some degree using the infrastructure.

A. Simple Example

As a simple example illustrating what might be involved in using the infrastructure to assemble an environment, we consider a domain with a simple input language: the set of boolean formulas without quantifiers.

Figure 4 presents an example of a grammar specification that might be used in this case (corresponding to task (2) in the enumeration of integration steps above). Unlike some existing parser and grammar specification formats, the supported grammar specification format does not require the user to specify any information redundantly in more than one place (e.g., terminals, how whitespace is handled, and so on). The purpose of this design is to support an approach that makes it possible to quickly define a grammar in one pass and then immediately begin interactively debugging the actual environment implementation and making incremental changes without a lot of overhead. The format provides some specialized constructs for handling arbitrarily long sequences of non-terminals, whitespace, indentation, line breaks, and regular expressions, and delimiters for comments and string literals. The parser module can automatically generate parsers, abstract syntax data structure definitions, and pretty-printers from such a specification.

```
'Top ::=
  'Top       | '1[StmtLine]

'StmtLine ::=
  'StmtLine  | ''_ 'Stmt

'Stmt ::=
  'Assert    | assert 'Formula

'Formula ::=
  'And       | 'Formula and 'Formula
  'Or        | 'Formula or 'Formula
  'Not       | not 'Formula
             | ( 'Formula )
  'True      | true
  'False     | false
             | 'Variable

'Variable ::=
  'Variable  | ''rgxnk/\b[a-z0-9_]+\b/
```

Fig. 4. Example of a simple grammar definition.

```
function assert(s)
  if s is Assert(e)   | return eval(e)

function eval(f)
  if f is And(f1,f2) | return eval(f2) and eval(f2)
  if f is Or (f1,f2) | return eval(f2) or eval(f2)
  if f is Not(f)     | return not eval(f)
  if f is True       | return true
  if f is False      | return false
```

Fig. 5. Simple evaluation algorithm defined using Informl.

Figure 5 presents an example of a very simple evaluation algorithm for the subset of formulas in the input language defined in Figure 4 that do not contain variables. Such an algorithm might be implemented as part of task (4) in the enumeration of integration steps above. This example illustrates the algebraic data types and pattern-matching constructs available in Informl; these can also be used when defining translations between the input syntax and the input/output syntax of integrated tools. Note also that the semantics of this algorithm implementation (i.e., its behavior when compiled) is beyond what is customarily expected: at compile time, it is possible to specify that the result returned by each recursive call within this implementation should also be automatically associated with the abstract syntax data structure node on which the algorithm was invoked.

B. Classroom Instruction

A web-based interactive formal verification environment that runs in a standard web browser was assembled; it incorporates several standard techniques: basic evaluation of expressions, monomorphic type inference, basic verification of a subset of logical formulas in propositional and first-order logic using unification, and an algorithm for computing congruence closures. The environment can be seamlessly integrated with a web page of lecture materials and homework assignments. Lectures and assignments can contain within them formal arguments (including both complete examples and problems to be completed) that can be interactively constructed and automatically verified in the environment. As a student interactively constructs a formal argument (as in Figure 6), the

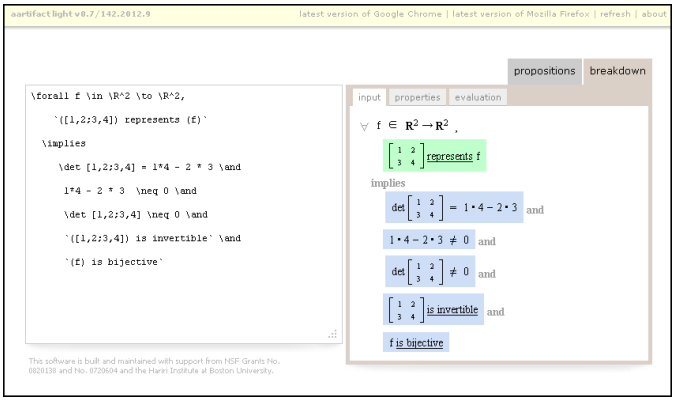

Fig. 6. Screen capture of end-user interface being used to assemble and verify a formal argument in an algebra course.

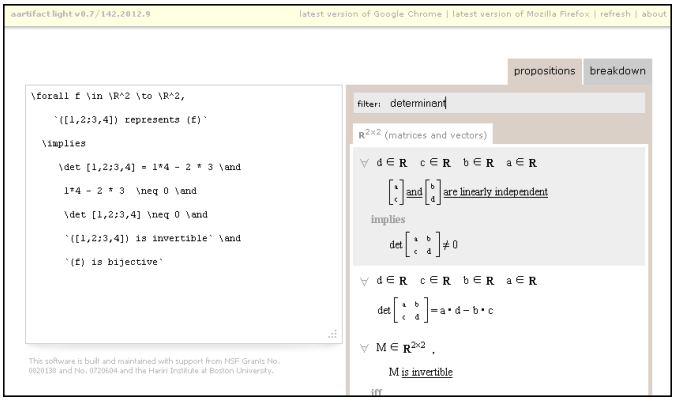

Fig. 7. Screen capture of end-user interface being used to explore the library of formal propositions.

environment can provide instant verification feedback based on the results of one or more underlying integrated verification algorithms. The environment also provides an explicit library of logical definitions and formulas that students can utilize to complete assignments (shown in Figure 7). Implementations of this environment, as well as individual components such as management tools for lecture notes, have been deployed in multiple computer science and mathematics courses on topics in linear algebra [17] and abstract algebra.

C. Distributed System Design

The aim of the Verificare project [18] is to provide a language and tools for specifying and analyzing real-world, formally disparate properties of distributed systems without requiring prior knowledge of any formal logics or languages. Verificare provides an integrated environment (as shown in Figure 8) for which the input DSL is VML, a lightweight modeling language designed to permit rapid, iterative development of system specifications. A library of formal requirements that end users may want to check against their models is also provided. Users can define models using VML, and then can select which properties they want to verify for that model. The model definition and the underlying definition of the properties are then translated into SPIN (i.e., Promela)

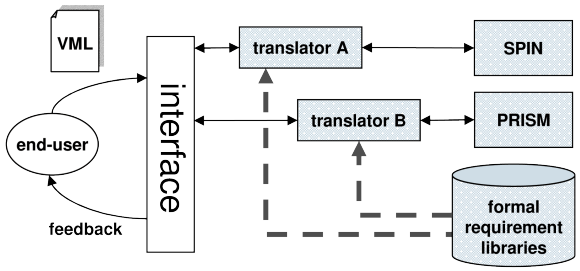

Fig. 8. Verificare integrated environment overview.

and PRISM syntax, and fed into instances of these tools running on a server. Outputs from the underlying tools are interpreted for the user in terms of the visible properties. Future plans include implementation of a translator for the Alloy Analyzer. Verificare has been applied [18] to software-defined networking examples involving OpenFlow.

IV. Related Work

This work is at least tangent to several areas of research, including the integration of multiple formal tools and techniques, and the deployment, practical accessibility, and usability within application domains of existing tools and techniques.

This work seeks to provide an infrastructure for integrating and multiple formal tools. There has been substantial interest in the model checking community in automating the translation of a model to multiple formal systems. PRISM [20] draws on numeric methods for linear system solving, as well as both symbolic and explicit state model checking libraries, to check properties of probabilistic systems. The Symbolic Analysis Laboratory (SAL) [21] is a suite of formal methods for checking properties of concurrent systems, including multiple model checkers, a type checker, and several simulation tools. The AVVANTSSAR project [22] is a platform for protocol security analysis, incorporating constraint solving, symbolic model checking, refinement libraries, and automated interaction with the Isabelle theorem prover. Cryptol [23] is a domain-specific language and tool suite that provides automated verification capabilities for cryptographic algorithms.

This work aims to address the disincentives to utilizing automated formal reasoning assistance systems by integrating multiple systems within a web-based accessible environment. We share motivation with, are inspired by, and incorporate ideas from related efforts to address practical usability in the formal systems communities. Some aim to provide interfaces that have a familiar syntax [24]–[27]; some aim to make optional the need to provide explicit references to the formal facts being used within the individual steps of a formal argument [28]–[30]; some aim to eliminate steep learning curves [31]–[33]; some aim to reduce the logistical difficulties of utilizing automated formal reasoning assistance systems [32], [34]. We are inspired by search mechanisms for libraries of formal facts [35]–[37] and programming language constructs [38], as well as keyword-based lookup mechanisms for programming environments [39], [40]. Providing the functionality of a formal reasoning environment within a browser is a goal that has been adopted by some projects [34], though that work focuses on delivering the look and functionality of a single existing proof assistant. The rise4fun website [41] provides a simple web protocol for integrating existing tools into the rise4fun web interface; the work presented in this paper focuses not only on providing a web interface for existing tools that can be made available as a web service, but also on providing modules for quickly defining new domain-specific languages, implementing corresponding translations to and from existing tools, and even implementing techniques and algorithms that can run directly within a standard web browser.

Our work is distinguished by its focus on the practical issues of tool and technique integration and deployment as a way to address the accessibility issues that may affect the adoption of existing tools and techniques. This can be viewed as an attempt to provide a general-purpose infrastructure (or collection of modules) for assembling the kinds of accessible formal reasoning environments proposed and prototyped in our earlier work [15]–[17]. Thus, the work is not focused on providing a rigorous mathematical framework for integration of techniques, but on developing an infrastructure that supports and accelerates other integration and deployment efforts (which may focus on the soundness of their integration techniques independently of this infrastructure, as is the case with Verificare [18]).

V. Status, Availability, and Future Work

The current prototype of the Aartifact infrastructure (including the underlying Informl modules) is available online [13], [14]. It is currently available as a collection of modules and not yet as a self-installing package or toolkit; documentation is sparse. Efforts to assemble a well-documented package are ongoing. The modules have not been tested extensively on a variety of different platforms, and some debugging and adjustment may be required to make the tools function. A few instantiated environments have been deployed on web servers running Linux. Prototype environments assembled and instantiated using the infrastructure are either under development [18] or have been deployed in the classroom [17].

A more sophisticated infrastructure for creating accessible integrated environments might provide components that can collect data that can improve the performance of the integrated environment given the application domain's distinct characteristics. For example, a web application deployed on the cloud and employed by many users simultaneously must maintain a certain level of responsiveness given the user load. Actual data showing which underlying tools and techniques are employed more often (or less often), and their computational cost and response time might inform how future versions of the integrated environment are deployed on the cloud (e.g., how many instances of a cloud-based virtual server running each underlying tool must be deployed given an expected user load).

It is also possible to explore ways to predict and compare the response time of each of the integrated tools and techniques

on various classes of input. These classes might be defined in terms of characterization and metric functions (e.g., based on their syntactic structure and depth, or in the case of logical formulas, the number of variables and quantifiers, and so on [42]), and the response time or completeness of the underlying tools for each submitted input could be recorded. This data can then be presented to users so that they can make intelligent trade-offs, or it can be used to automatically choose the most suitable integrated technique based on the user's priorities (e.g., completeness or response time) by incorporating machine learning techniques, as has been in work on integrating SMT solvers [42].

REFERENCES

[1] W. Schulte, "Why doesn't anyone use formal methods?" in *Integrated Formal Methods*. Springer, 2000, pp. 297–298.

[2] D. Parnas, "Really rethinking'formal methods'," *Computer*, vol. 43, no. 1, pp. 28–34, 2010.

[3] 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, 2010.

[4] M. Bersani, C. Furia, M. Pradella, and M. Rossi, "Integrated modeling and verification of real-time systems through multiple paradigms," in *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*. IEEE, 2009, pp. 13–22.

[5] R. Büssow and W. Grieskamp, "A modular framework for the integration of heterogeneous notations and tools," in *Proc. of the 1st Intl. Conference on Integrated Formal Methods–IFM*, vol. 99, 1999.

[6] J. Rushby, "Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving," *Theoretical and Practical Aspects of SPIN Model Checking*, pp. 1–11, 1999.

[7] S. Katz and O. Grumberg, "A framework for translating models and specifications," in *Integrated Formal Methods*. Springer, 2002, pp. 145–164.

[8] T. Hoare and J. Misra, "Verified software: Theories, tools, experiments vision of a grand challenge project," in *VSTTE*, ser. Lecture Notes in Computer Science, B. Meyer and J. Woodcock, Eds., vol. 4171. Springer, 2005, pp. 1–18.

[9] V. Nepomniaschy, N. Shilov, E. Bodin, and V. Kozura, "Basic-real: integrated approach for design, specification and verification of distributed systems," in *Integrated Formal Methods*. Springer, 2002, pp. 69–88.

[10] B. Ellis and A. Ireland, "An integration of program analysis and automated theorem proving," in *Integrated Formal Methods*. Springer, 2004, pp. 67–86.

[11] M. Baldamus and J. Schröder-Babo, "p2b: A translation utility for linking promela and symbolic model checking (tool paper)," *Model Checking Software*, pp. 183–191, 2001.

[12] J. Bowen, "Combining operational semantics, logic programming and literate programming in the specification and animation of the verilog hardware description language," in *Integrated Formal Methods*. Springer, 2000, pp. 277–296.

[13] (2013) Aartifact. [Online]. Available: http://www.aartifact.org/

[14] (2013) Informl. [Online]. Available: http://www.informl.org/

[15] A. Lapets and A. Kfoury, "A user-friendly interface for a lightweight verification system," *Electronic Notes in Theoretical Computer Science*, vol. 285, no. 0, pp. 29 – 41, 2012, proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066112000242

[16] A. Lapets, "User-friendly Support for Common Concepts in a Lightweight Verifier," in *Proceedings of VERIFY-2010: The 6th International Verification Workshop*, Edinburgh, UK, July 2010.

[17] ——, "Accessible Integrated Formal Reasoning Environments in Classroom Instruction of Mathematics," in *Proceedings of HCSS 2012*, Washington, D.C., USA, May 2012.

[18] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury, "Verifiably-Safe Software-Defined Networks for CPS," in *Proceedings of the 2nd ACM International Conference on High Confidence Networked Systems (HiCoNS 2013)*, Philedelphia, PA, USA, April 2013.

[19] A. Lapets and S. Mirzaei, "Toward Lightweight Integration of SMT Solvers," CS Dept., Boston University, Tech. Rep. BUCS-TR-2012-017, December 2012. [Online]. Available: http://www.cs.bu.edu/techreports/pdf/2012-017-smt-integration.pdf

[20] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-time Systems," in *23rd International Conference on Computer Aided Verification*. Springer, 2011, pp. 585–591.

[21] L. D. Moura, S. Owre, and N. Shankar, "The SAL Language Manual," SRI International, Tech. Rep. 650, 2003.

[22] L. Vigano, "AVANTSSAR Validation Platform v2," Tech. Rep. 216471, 2011. [Online]. Available: http://www.avantssar.eu/pdf/deliverables/avantssar-d4-2.pdf

[23] L. Erkök and J. Matthews, "Pragmatic equivalence and safety checking in cryptol," in *Proceedings of the 3rd workshop on Programming languages meets program verification*, ser. PLPV '09. New York, NY, USA: ACM, 2008, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1481848.1481860

[24] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, "The Fortress Language Specification Version 1.0," March 2008. [Online]. Available: http://research.sun.com/projects/plrg/fortress.pdf

[25] F. Kamareddine and J. B. Wells, "Computerizing Mathematical Text with MathLang," *Electronic Notes in Theoretical Computer Science*, vol. 205, pp. 5–30, 2008.

[26] M. M. Wenzel, "Isabelle/isar - a versatile environment for human-readable formal proof documents," Ph.D. dissertation, Institut für Informatik, Technische Universität München, 2002.

[27] P. Rudnicki, "An overview of the Mizar project," in *Proceedings of the 1992 Workshop on Types and Proofs for Programs*, 1992, pp. 311–332.

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

[29] C. E. Brown, "Verifying and Invalidating Textbook Proofs using Scunak," in *MKM '06: Mathematical Knowledge Management*, Wokingham, England, 2006, pp. 110–123.

[30] J. H. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet, "Proof Development with OMEGA: sqrt(2) Is Irrational," in *LPAR*, 2002, pp. 367–387.

[31] A. Asperti, C. S. Coen, E. Tassi, and S. Zacchiroli, "User interaction with the matita proof assistant," *Journal of Automated Reasoning*, vol. 39, no. 2, pp. 109–139, 2007.

[32] D. Jackson, "Alloy: a lightweight object modelling notation," *Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

[33] R. M. Burstall, "Proveeasy: helping people learn to do proofs," *Electr. Notes Theor. Comput. Sci.*, vol. 31, pp. 16–32, 2000.

[34] C. Kaliszyk, "Web interfaces for proof assistants," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 2, pp. 49–61, 2007.

[35] P. Cairns and J. Gow, "Integrating Searching and Authoring in Mizar," *Journal of Automated Reasoning*, vol. 39, no. 2, pp. 141–160, 2007.

[36] G. Bancerek and J. Urban, "Integrated semantic browsing of the Mizar Mathematical Library for authoring Mizar articles," in *MKM*, 2004, pp. 44–57.

[37] T. Hallgren and A. Ranta, "An extensible proof text editor," in *LPAR '00: Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 70–84.

[38] N. Mitchell, "Hoogle overview," *The Monad.Reader*, vol. 12, pp. 27–35, November 2008.

[39] 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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 332–343.

[40] 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*. New York, NY, USA: ACM, 2007, pp. 84–93.

[41] (2013) rise4fun: a community of software engineering tools. [Online]. Available: http://rise4fun.com/

[42] M. A. Aziz, A. Wassal, and N. Darwish, "A Machine Learning Technique for Hardness Estimation of QFBV SMT Problems (Work in progress)," in *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories*, Manchester, UK, 2012, pp. 56–65.