Towards an Infrastructure for Integrated Accessible Formal Reasoning Environments

Andrei Lapets
Boston University
Boston, Massachusetts, USA
lapets@bu.edu

Richard Skowyra
Boston University
Boston, Massachusetts, USA
rskowyra@bu.edu

Christine Bassem
Boston University
Boston, Massachusetts, USA
cbassem@cs.bu.edu

Assaf Kfoury
Boston University
Boston, Massachusetts, USA
kfoury@cs.bu.edu

Azer Bestavros
Boston University
Boston, Massachusetts, USA
best@bu.edu

Abstract

Computer science researchers in the programming languages and formal verification communities have produced a variety of automated tools and techniques for assisting formal reasoning tasks. However, while there exist notable successes in utilizing these tools to develop safe and secure software and hardware, both leading-edge advances and basic techniques (such as model checking, state space search, type checking, logical inference and verification, computation of congruence closures, non-interference enforcement, and so on) remain underutilized by large populations of end-users that may benefit from them when they engage in formal reasoning tasks within their own application domains. This may be in part because (1) these tools and techniques are not readily accessible to end-users who are not experts in formal systems or are simply not aware of what is available and how it can be utilized, and (2) these tools and techniques are only valuable when used in conjunction with one another and with appropriate domain-specific libraries and databases.

Motivated by these circumstances, we present our ongoing efforts, built on earlier work in developing user-friendly formal verification tools, to develop an infrastructure for assembling user-friendly, interactive, integrated formal reasoning environments that can assist users engaged in routine domain-specific formal reasoning tasks. This infrastructure encompasses a programming language, compilers, and other tools for building up from components, instantiating with domain-specific formal content, and finally delivering such environments in the form of ready-to-use web-based applications that can run entirely within a standard web browser. We describe current efforts to use such instantiated environments in two application domains: classroom instruction of linear algebra and verifying the correctness of protocols.

1 Introduction

Computer science researchers in the programming languages and formal verification communities have produced a wide variety of automated tools and techniques for assisting formal reasoning tasks. However, while there exist notable successes in utilizing these tools to develop safe and secure software and hardware, both leading-edge advances and basic techniques (such as model checking, state space search, type checking, logical inference and verification, computation of congruence closures, non-interference enforcement, and so on) remain underutilized by large populations of end-users that may benefit from them when they engage in formal reasoning tasks within their own application domains. These populations include students and instructors, researchers, and experts working in particular application domains. This state of affairs may be a consequence of the fact that (1) many existing tools and techniques are not readily accessible to end-users who are not experts in formal systems or are not familiar with what is available and how it may be used, and that (2) these tools and techniques only become practically valuable when they can be used in conjunction with one another and with appropriate domain-specific libraries to engage in routine formal reasoning tasks.

Motivated by the current state of affairs, we present our ongoing efforts, built on earlier work in developing user-friendly formal verification tools [20, 19, 12], to develop a general-purpose infrastructure for defining, implementing, instantiating, and delivering user-friendly, interactive, integrated formal reasoning environments (henceforward we will refer to these as *environments* for concision) that can assist users engaging in routine domain-specific formal reasoning tasks. This infrastructure encompasses a programming language, compilers, user-interface features, and other tools for building up from components, instantiating with domain-specific formal content, and finally delivering such environments in the form of ready-to-use web-based applications that can run entirely within a standard web browser.

In this work we begin to define what we consider to be a minimal collection of tools and conventions that are needed to implement and support an accessible integrated environment for a chosen application domain. In doing so, we develop prototypes of practical tools and conventions for this underlying infrastructure. Furthermore, we begin to develop a framework and context in which to raise new questions and problems associated with defining and implementing accessible integrated environments.

Accessibility of formal reasoning tools and practical integration of such tools can be treated as two complementary but orthogonal issues, and these issues can be addressed separately. However, we believe (and attempt to illustrate in this work) that addressing them simultaneously compels each effort to reinforce the other. Also focusing on the accessibility of integrated components rather than only on the integration of the components leads us to create tools to support fast and unique implementations of integrated components that enable user-friendly and practical delivery over the web of the resulting automated formal assistance capabilities. Likewise, also focusing on the integration of components rather than only on the accessibility of individual algorithms or tools leads us to build general-purpose interface features that are not necessarily limited to the idiosyncratic strengths and weaknesses of particular tools and techniques (though they may expose and let users make tradeoffs between different tools based on their strengths and weaknesses). Thus, such interface features can be used for combinations of underlying components that may have different automated assistance capabilities and characteristics.

To illustrate more concretely our vision and to begin evaluating our efforts, we consider two use cases in which end-users benefit not just by having an environment with an accessible interface, but also by having the ability to *integrate* multiple tools and techniques.

2 Motivating Examples

We provide two use cases involving an accessible integrated formal reasoning environment in order to illustrate some of the key features that distinguish such an environment. While the two examples differ in many ways, the focus of this paper is on a *common infrastructure* that would support the creation of the kinds of environments described below. The first example is drawn from actual experience employing the latest prototype environment in the classroom.

The second represents a use case that we hope to support once the ongoing work is completed.

Classroom Instruction of Mathematics. Suppose that an instructor of an undergraduate linear algebra course in a mathematics department wants: to present in lectures examples of formal arguments (e.g., proofs of theorems and solutions to problems) in a standard syntax that can be processed automatically; to have students write their homework solutions while enjoying the benefits of an interactive, automatic verification tool that requires no special environment or setup procedure; and to grade homework automatically.

One possible solution might consist of the following components.

- An interactive web-based verification environment that allows a user (e.g., a student) to interactively construct a formal argument while receiving instant verification feedback based on the results of one or more integrated verification algorithms (e.g., an evaluation algorithm for expressions containing matrices, a simple logical inference algorithm based on computation of congruence closures, and a basic logical verification algorithm based on unification). Figure 1 illustrates how our own prototype environment appears within a standard web browser when it is used to author a basic linear algebra argument.

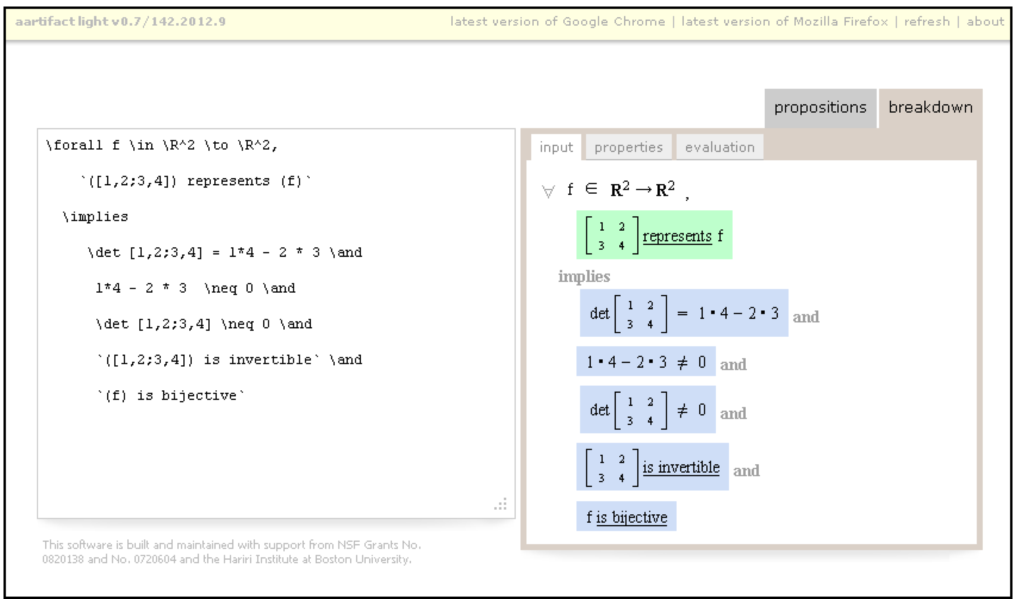

Figure 1: Screen capture of end-user interface.

- A content management system (CMS) for authoring and posting lecture notes online. This CMS should be extended with features that allow inclusion of formal arguments that are automatically verifiable. Students who view the lecture notes would be able to see both a “friendly” version of a formal argument, or its raw source (as illustrated in Figure 2). Students should be able to load any formal argument instantly into the environment (as illustrated in Figure 3), whether it is complete or, in the case of an exercise or assignment to be completed by students, incomplete.
- A tool for assembling and including in the lecture notes problems that students must solve within the interactive environment (including, for each problem, a subset of formal facts

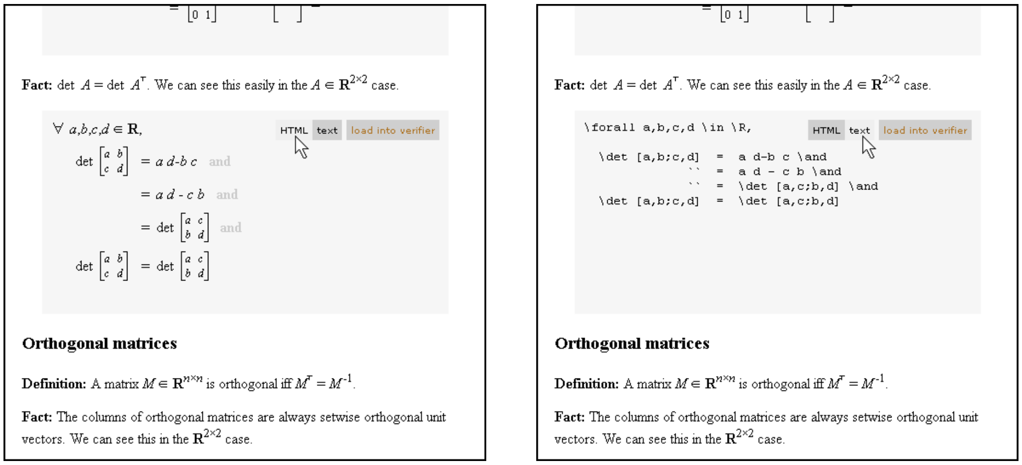

Figure 2: Screen captures of formal arguments integrated into online lecture notes.

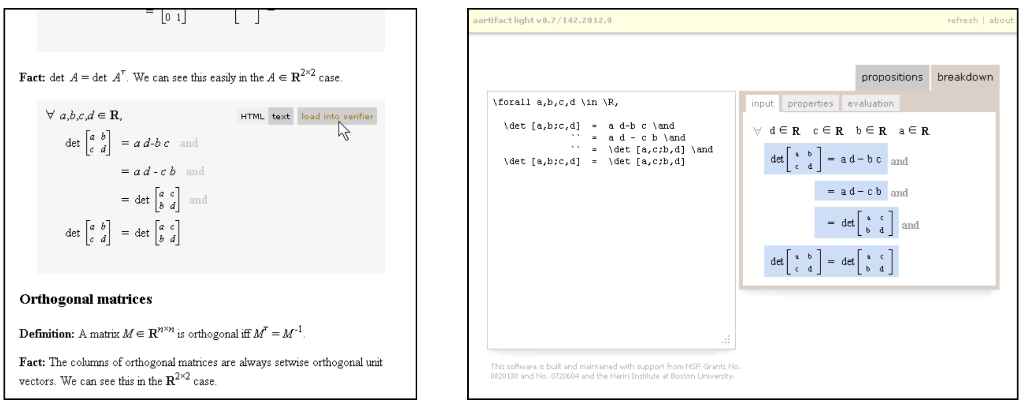

Figure 3: Loading a formal argument in the lecture notes into the environment.

that students may employ). An instructor should be able to use the CMS to assemble a collection of facts that may be used in a given assignment. When students load a particular assignment into the environment as in (2) above, they should be able to see, browse, and search using keywords the formal facts the instructor has made available for them to use in their solutions (as illustrated in Figure 4).

Designing Protocols. Suppose a protocol designer is defining a simple leadership election protocol over a unidirectional ring of processes, each of which has a unique numerical identifier. The process with the highest identifier should be elected leader at the end of a protocol execution. Each process first sends its own ID to its successor, then uses the following algorithm whenever it receives an ID from its predecessor (see Figure 5).

While a number of different properties can be checked automatically, consider two classes that the designer is interested in: relational, global properties and temporal/concurrency properties. The Alloy Analyzer [15], a model checker which accepts models and constraints written in the Alloy [13] language, can be used to checker the former. SPIN [11], an explicit-state model

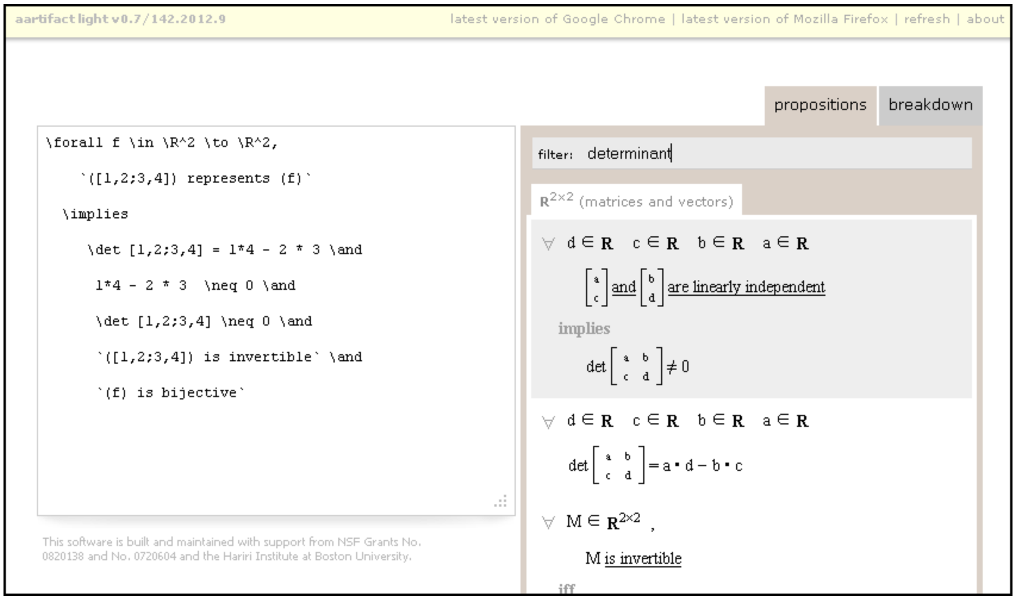

Figure 4: Screen capture of end-user interface.

```
At start, send own ID to successor
Each round, send one ID in pool to successor
If ID is received from predecessor, compare to own ID:
  Higher: add to ID pool
  Lower: drop
  Equal: elect self as leader and stop sending messages
```

Figure 5: An example of a protocol to elect the process with the highest ID in a set of processes.

checker that accepts models written in Promela and properties specified in Linear Temporal Logic (LTL) can be used to check the latter.

The correctness properties the designer wants to check using Alloy are:

- Is at most one process elected as leader?
- Is at least one process elected as leader?
- Is the process with the highest identifier always elected leader?
- Can a process ever have no IDs to send, but not be the leader?

The correctness properties the designer wants to check using SPIN are:

- Are deadlocks possible?
- Once a process is elected, are no other processes elected at any future time?
- Once a process is elected, is it elected at all future times?

The integrated environment provides a standard syntax for protocol designers that can be used to write down both the definition of a protocol and formulas describing its properties. Any protocol definition can be translated to both an equivalent Alloy syntax, and an equivalent SPIN syntax. Any formula specified by the designer can be converted to Alloy syntax, SPIN syntax, or both, depending on whether its structure fits each system. The above protocol, written in a syntax accommodated by the environment, is presented in Figure 6.

```
class Process:
    id: Pid
    predecessor: Process
    Link(predecessor, self, 1)
    successor: Process
    Link(self, successor, 1)
    pool: set Pid
    leader: Bool

    init: id -> successor

    act send(not leader):
        some Pid in pool -> successor

    act receive(predecessor -> rec_id):
        if (rec_id < id):
        elif (rec_id = id): leader = True
        elif (rec_id > id): pool.add(rec_id)

    act final(leader): accept

    invariant Ring:
        all p in Proccess:
          all q in Process in p.successor*

    invariant PredSucc:
        all p,q in Proccess:
          q = p.successor implies p=q.predecessor
```

Figure 6: Leadership election in a unidirectional ring

The two `Links` denote buffered message channels of size 1, which define the behavior of the `->` operator when it is used between a process' predecessor and itself or itself and its successor, respectively. If a Link is not explicitly defined, `->` defaults to synchronous message passing.

The `act` keyword denotes labeled, guarded actions that a protocol may take. The `final` action, for example, may be taken only if `leader` is true. Note that there is no particular semantics associated with the case that multiple guards are true simultaneously. Different formal tools handle this case in different ways, or may not even have a concept of concurrency.

The `invariant` keyword introduces required structure over the successor and predecessor pointers of a process. `Ring` enforces the topological constraint of the protocol (a unidirectional ring), and should be read as 'for all processes p, any process q is in the transitive closure (e.g. reachable via a finite chaining) of its successor pointer. `PredSucc` enforces the requirement that predecessor is the inverse of successor, and should be read as 'If q is the successor of p, then p is the predecessor of q'.

It is important to note that the above protocol model may not be fully expressible in all formal tools. Alloy, for example, has no concept of buffered channels, and would model message passing as changes to relations between identifiers and atoms in the Process set. In this case, any translation performed by the environment would be accompanied by an explicit message informing the user of the loss in detail.

The properties that the designer is interested in checking are expressed in a similar manner to invariants, as presented in Figure 7.

```
# At all times, is at most one process eventually elected as
leader? all p,q in Proccess: p.leader implies no q.leader

# At all times, is at least one process eventually elected as leader?
all p in Proccess: always eventually some p.leader

# Is the process with the highest identifier always elected leader?
all p,q in Process: p.leader implies no q: q.id > p.id

# Can a process ever have no IDs to send, but not be the leader?
some p in Process: p.pool={} and not p.leader

# Once a process is elected, is it elected at all future times?
all p in Process: p.leader implies always p.leader

# Once a process is elected, are no other processes elected
# at any future time?
all p,q in Process: p.leader implies never q.leader

#Are deadlocks possible?
some p in Process: always blocked
```

Figure 7: Protocol properties to be checked

Note that the syntax used in expressing these properties (largely first-order logic with some temporal operators) is translated, if possible, into the formal logic(s) supported by the component systems integrated within the environment. Finally, the deadlock property makes use of the `blocked` keyword. This denotes a state in which the process is either unable to act (i.e. no action guard is true) or is waiting on a blocking operation (e.g. sending a message on a link with a full buffer). Our infrastructure will translate as many of the above properties as possible to both Alloy and SPIN. Once verification is complete, the model checkers' results can be presented as counterexamples or verification statements to the user.

3 Infrastructure for Accessible Integrated Environments

We propose an infrastructure for implementing, instantiating, and delivering 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, 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 authors 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 8.

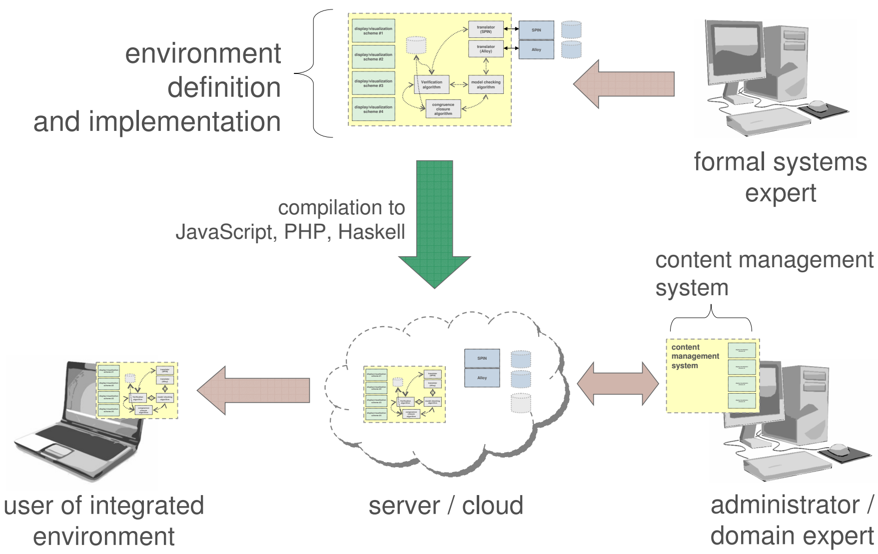

Figure 8: Overall organization of infrastructure.

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., monomorphic type checking, congruence closure computation, resolution, unification, and so on), as well as appropriate translations for external systems or components (e.g., translations of a particular syntax for protocols and protocol properties to an appropriate Alloy or SPIN syntax). In order to support formal systems experts in this task, it is necessary to provide a language that: (1) allows them to easily define a standard definition for expressions (formulas and terms) for a particular environment (parsers are generated automatically); (2) allows them to specify algorithms that operate on these formulas; and (3) allows them to define algorithms that are interdependent and can invoke one another.

All component algorithms and translations must be transformations that are defined on a subset of the expression space 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 argument provided by the user as input. Each subexpression can then be annotated with the results of applying various components to that subexpression, as illustrated in Figure 10. Environment implementers can then combine various 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 9. If the dependency graph is a DAG, 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.

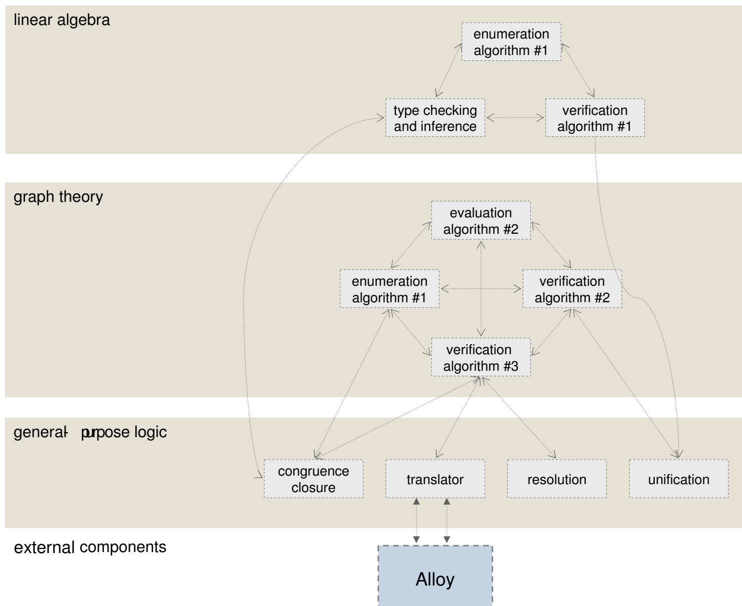

Figure 9: Example of a component algorithm dependency graph.

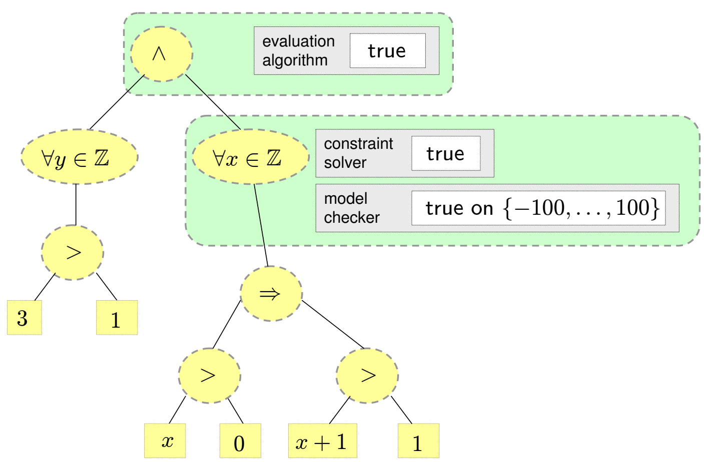

Figure 10: Abstract syntax tree for "$(\forall y \in \mathbb{Z},\ 3 > 1) \wedge (\forall x \in \mathbb{Z},\ (x > 0 \ \Rightarrow\ x + 1 > 1))$" with multiple values (produced by different components) for some subexpressions.

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 and can be used by the end-user in constructing formal arguments, and to provide other content or documentation that may incorporate examples of formal arguments. 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 our earlier work [20, 19].

End-users and a Web-based Interactive Environment. End-users can use any standard browser capable of running JavaScript applications to run the integrated environment. Two possible user experiences for particular instantiations of an integrated environment have been described in Section 4.

From the user's perspective, the environment provides a way to input automatically verifiable formal arguments using a conventional syntax similar to that of LaTeX. Within a particular environment instantiation, the functionality provided by the integrated components is exposed using a variety of JavaScript visualization widgets that include:

- friendly, formatted output of the formal argument, including highlighting of syntactic and logical errors, or other properties derived through automated analysis;
- lists of formal expressions, including the propositions available in the library, facts derived by one or more component algorithms, and translations of the input into syntaxes for particular underlying systems and tools;
- interactive controls for starting and stopping inference, evaluation, enumeration, and verification components that have been implemented to allow partial results.

4 Related and Future Work

This work incorporates ideas and techniques from multiple disciplines and areas of research. We review related work that shares one of two relevant features with this work.

Practical Usability of Automated Formal Systems. This work aims to address the disincentives to utilizing automated formal reasoning assistance systems by integrating multiple systems within an 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 [2, 17, 27, 24]; 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 [1, 5, 25]; some aim to eliminate steep learning curves [3, 14, 6]; some aim to reduce the logistical difficulties of utilizing automated formal reasoning assistance systems [16, 14]. We are inspired by search mechanisms for libraries of formal facts [7, 4, 9] and programming language constructs [22], as well as keyword-based lookup mechanisms for programming environments [10, 21]. Providing the functionality of a formal reasoning environment within a browser is a goal that has been adopted by some projects [16], though that work focuses on delivering the look and functionality of an existing proof assistant.

Use of Multiple Automated Formal Systems in Conjunction. This work seeks to provide an infrastructure for integrating and automating multiple formal tools. We draw on previous work in the model checking community, from which there has been substantial interest in automating the translation of a model to multiple formal systems. PRISM [18] 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) [23] is a suite of formal methods for checking properties of concurrent systems, including multiple model checkers, a type checker, and several simulation tools. The AVANTSSAR project [26] is a platform for protocol security analysis, incorporating constraint solving, symbolic model checking, refinement libraries, and automated interaction with the Isabelle theorem prover.

Cryptol [8] is a domain-specific language and tool suite that provides automated verification capabilities for cryptographic algorithms. In Cryptol, verification can be performed in a fully-automated manner, in which modern off-the-shelf SAT and SMT solvers are used to perform the verification, or in a semi-automated manner, in which the Cryptol-written theorems are translated into Isabelle/HOL to be manually constructed by the user. The Cryptol tool suite allows system designers to experiment their programs as their designs evolve, and provides them with the capability of generating C, C++, Haskell software implementations, or VHDL and Verilog HDL hardware implementations.

Our work is distinguished from the above by a stronger emphasis on accessibility for non-expert end-users, on delivery of functionality over the web, and on providing feedback for high-level partial formal arguments. We also explicitly address the task of implementing, extending, and instantiating integrated environments of both basic algorithms and systems.

Future Work. This work raises many new questions about how accessibility and integration can play complementary roles in delivering the benefits of automated formal reasoning assistance tools and techniques to end-users working in particular application domains. Further work is required to determine whether the supporting languages and tools are sufficiently flexible and expressive for integrating a wide variety of existing algorithms and automated systems.

As an example of one specific direction for future work, many other workflows and practical tasks could be better supported by CMS extensions. For example, a CMS used in classroom instruction may be extended with facilities that allow students to submit their completed formal arguments, or with logging tools that actually allow the instructor to see and play back an entire log showing how a student assembled his or her solution to a particular problem.

References

[1] 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.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0. March 2008.

[3] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

[4] Grzegorz Bancerek and Josef Urban. Integrated semantic browsing of the Mizar Mathematical Library for authoring Mizar articles. In *MKM*, pages 44–57, 2004.

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

[6] Rod M. Burstall. Proveeasy: helping people learn to do proofs. *Electr. Notes Theor. Comput. Sci.*, 31:16–32, 2000.

[7] Paul Cairns and Jeremy Gow. Integrating Searching and Authoring in Mizar. *Journal of Automated Reasoning*, 39(2):141–160, 2007.

[8] Levent Erkök and John Matthews. Pragmatic equivalence and safety checking in cryptol. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, pages 73–82, New York, NY, USA, 2008. ACM.

[9] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *LPAR '00: Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, pages 70–84, Berlin, Heidelberg, 2000. Springer-Verlag.

[10] Sangmok Han, David R. Wallace, and Robert 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.

[11] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2005.

[12] Vatche Ishakian, Andrei Lapets, Azer Bestavros, and Assaf Kfoury. Formal Verification of SLA Transformations. In *Proceedings of CloudPerf 2011: IEEE International Workshop on Performance Aspects of Cloud and Service Virtualization*, Washington, D.C., USA, July 2011.

[13] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[14] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[15] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, revised ed edition, 2012.

[16] Cezary Kaliszyk. Web interfaces for proof assistants. *Electronic Notes in Theoretical Computer Science*, 174(2):49–61, 2007.

[17] Fairouz Kamareddine and J. B. Wells. Computerizing Mathematical Text with MathLang. *Electronic Notes in Theoretical Computer Science*, 205:5–30, 2008.

[18] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *23rd International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[19] Andrei 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.

[20] Andrei Lapets and Assaf 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.

[21] Greg Little and Robert 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.

[22] Neil Mitchell. Hoogle overview. *The Monad.Reader*, 12:27–35, November 2008.

[23] Leonardo De Moura, Sam Owre, and N Shankar. The SAL Language Manual. Technical Report 650, SRI International, 2003.

[24] P. Rudnicki. An overview of the Mizar project. In *Proceedings of the 1992 Workshop on Types and Proofs for Programs*, pages 311–332, 1992.

[25] Jörg H. Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, and Martin Pollet. Proof Development with OMEGA: sqrt(2) Is Irrational. In *LPAR*, pages 367–387, 2002.

[26] Luca Vigano. AVANTSSAR Validation Platform v2. Technical Report 216471, 2011.

[27] Markus M. Wenzel. *Isabelle/Isar - A versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.