

---

# The Evidential Tool Bus

---

Computer Science Laboratory  
SRI International

## 1 Introduction

The rising diversity of verification tools—proof assistants, model checkers, satisfiability solvers, predicate abstractors, to name a few—can be seen as both a testament to the health of formal methods, and as an impediment to their widespread adoption. In particular, in such a rich ecosystem, the process of making an enlightened choice about the best combination of tools for a given verification task can, in itself, be fairly problematic. This choice bears even more weight considering that there is no guarantee that formal developments in a given system can be later ported to other systems. Solutions to this problem often come as *ad hoc* implementations: mainly, translators between proof assistants [11, 14, 15, 6], and integration of solvers, model-checkers, and decision procedures into proof assistants [18, 9]. These approaches all have in common the fact that they are at the same time fragile, because any change in the source or target implementation will break the translation; and expensive to establish and maintain, since they require in-depth expertise of the systems involved.

The novel concept of a *formal tool bus* takes a different approach towards composition and interoperability, by relying on asynchronous message passing between standalone formal verification tools. The tools behave as *distributed agents* that can either publish a formula they wish to see proved, or answer such a request with some evidence attesting of their success. Agents register the services they provide, as well as the syntax and semantics of their logical language, to a *facilitator*, that takes care of the lower-level parts of the connection. The Formal Methods group at SRI International has developed a formal tool bus called Evidential Tool Bus (Tool Bus or ETB for short) [17], basing it on the industrial-strength distributed framework Open Agent Architecture [13], and starting with the connection of the Yices SMT-solver [8], the SAL model-checking suite [3], and the PVS proof assistant [16]. By combining formal verification tools in a distributed framework, the formal tool bus architecture aims at facilitating the elaboration of powerful, flexible, and interoperable tool chains.

## 2 Rationale

The capabilities of the Evidential Tool Bus had to meet several criterions and requirements.

- Semantical integration : as opposed to operational integration. This entails providing an ontological framework for the definition of the various logics and representations used by the connecting tools.
- Plug-in architecture : for easy feature development. A new component of the Tool Bus should be implementable with only minimal knowledge of its internal workings.
- Networked accessibility : for distributed connection of tools. This feature should not be implemented at the expense of safety. It induces the use of lazy data sharing to allow resource and bandwidth economy.
- Scripting capability : providing chaining facilities. This includes forward chaining from existing judgements, as well as backward chaining through the generation of proof obligations.
- Control capability : providing component oversight, and both process and data garbage collection.
- Resource discovery functionality : for human discovery of services.
- Evidence generation : to allow trusted verification of proofs.

Verification problems including, but not limited to, the contents of the repository of the Verified Software Grand Challenge [12] will make the basis of the ETB testing suite.

### 3 Architectural Overview

The ETB is developed as an overlay on top of the Open Agent Architecture (OAA) [13]:

“In a distributed agent framework, we conceptualize a dynamic community of agents, where multiple agents contribute services to the community. When external services or information are required by a given agent, instead of calling a known subroutine or asking a specific agent to perform a task, the agent submits a high-level expression describing the needs and attributes of the request to a specialized Facilitator agent. The Facilitator agent will make decisions about which agents are available and capable of handling sub-parts of the request, and will manage all agent interactions required to handle the complex query.”

In particular, the ETB leverages the Facilitator component of the OAA to guide component interactions.

Figure 1 provides an overview of the structure of the ETB.

### 4 The Facilitator

The Facilitator component in the ETB is provided by the Open Agent Architecture. The following attributes and method calls are used to interface with this subsystem (subscripts denote arities):

`OAA_AGENT_NAME` : a protected static String containing the identifier of the agent.

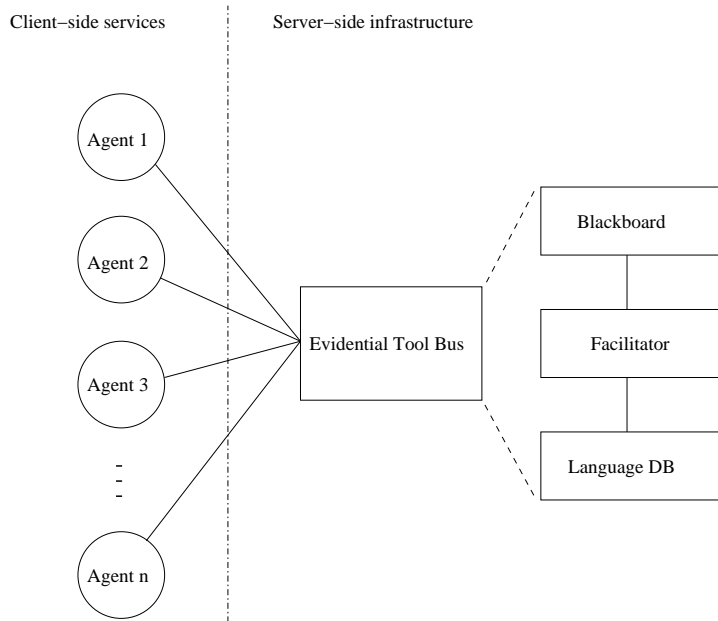


Figure 1: The architectural design of the Tool Bus ecosystem.

`OAA_SOLVABLES` : a **protected static String** holding the list of methods in the class that are registered as invocable from other agents. The list is constructed using the symbols '[' , ']' and ',' .

`registerSolvable1` : a **public void** method taking in argument the agent instance and registering its capabilities with the Facilitator.

`OaaConnection` : this class takes care of the lower-levels of interaction with the Facilitator. It defines the following methods:

`connect0` : a **public boolean** method which connects the instance to the Facilitator using the default host and port.

`solve3` : a **public boolean** method taking in argument an `IclTerm` goal, an `IclList` of parameters and an `IclList` of answers. In response to this method, the Facilitator will invoke all registered methods that correspond to the goal, using side-effects to modify the list of answers.

## 5 The Blackboard

## 6 The Language Database

In order to be able to teach and share music, European scholars starting with Guido d'Arezzo in the 9th century have developed a common notation format, based on a five-line staff and predefined note values, to which they adjoined symbols defining the interpretation that needed to be made of these notations, such as the key, key signature, and time signature. Similarly, the Evidential Tool Bus needs a mechanism akin to musical scores and keys to ensure the correctness of the communication between its agents. The

language database serves this purpose: it registers the various *syntax* and *semantics* of agents, thus defining the form of the judgements exchanged on the bus. Such a declaration is carried within a file bearing the “.edf” extension. In this section we detail the contents of such a file, presenting the symbolic representation that is used to represent the syntactic and semantical components.

## 6.1 Syntax

As the exchanges over the bus are carried in XML format, their syntax can be defined as an XML dialect, i.e., an extension of XML that obeys a predefined grammar. Coincidentally, a number of different languages have been designed to express these grammars. In the Evidential Tool Bus, we reuse one of these syntax definition language, called RELAX NG [5], to define the syntax of agent statements.

As a running example, we use an XML encoding of the DIMACS format used to represent boolean formulas in conjunctive normal form. So, for instance, the formula

$$(A \vee \neg C) \wedge (A \vee C \vee \neg A)$$

has the equivalent DIMACS representation

```
c A sample .cnf file.
p cnf 3 2
1 -3 0
2 3 -1 0
```

where the line starting with a `c` is a comment, the line starting with a `p` declares the form of the formula (conjunctive normal form), the number of variables it contains (3) and the number of clauses (2). Each line thereafter specifies a clause, with each literal being assigned a number, the ‘-’ character denoting negation, and ‘0’ closing each line. An equivalent XML encoding can easily be derived:

```
<?xml version="1.0"?>
  <comment>A sample .cnf file</comment>
  <formula format="cnf" variable_num="3" clause_num="2">
    <clause>
      <atom name="1"/>
      <neg><atom name="3"/></neg>
    </clause>
    <clause>
      <atom name="2"/>
      <atom name="3"/>
      <neg><atom name="1"/></neg>
    </clause>
  </formula>
```

The RELAX NG language allows the description of the grammar of this encoding, using notations popularized by the Backus-Naur form meta-syntax, in which language patterns are specified using simple typographic conventions. So for instance, an atom can be defined as an `atom` element that has a `name` attribute:

```
dimacs-atom = element atom {attribute name {text}}
```

This declaration is called a *named pattern*, where the '=' symbol binds its left-hand side, the `dimacs-atom` identifier, to its right-hand side, the `element` construction. The contents of the `name` attribute is specified to be simple text, but more precise datatypes can be used. Here, we would like to specify that atom names are integers, thus the previous declaration can be refined into:

```
dimacs-atom = element atom {attribute name {xsd:integer}}
```

where `xsd:integer` represents the integer datatype. Then, using the '|' symbol, a literal can be defined as either an atom, or its negation:

```
dimacs-literal = {  
    dimacs-atom  
    | element neg {dimacs-atom}  
}
```

Now that the grammar of literals is properly defined, we would like to enlarge the definition to clause elements, which contain one or more children that are `literal` elements. This is quite straightforward, using the previously named pattern:

```
dimacs-clause = element clause {dimacs-literal +}
```

where the '+' symbol denotes the "one or more" concept. Finally, a DIMACS formula can be defined as the ordered succession of:

- zero or more `comment` elements,
- a `formula` element featuring the `format`, `variable_num`, and `clause_num` attributes, and one or more `clause` elements.

this leads us to introduce the ',' symbol for ordered conjunction, as well as the `start` keyword that names the root of the grammar, resulting in the following RELAX NG definition.

```
default namespace = "http://etb.csl.sri.com/ns/dimacs"  
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"
```

```
start = {  
    element comment {text} *,  
    element formula {  
        attribute format      {"cnf"},  
        attribute variable_num {xsd:integer},  
        attribute clause_num   {xsd:integer},  
        dimacs-clause +  
    }  
}
```

```
dimacs-clause = element clause {dimacs-literal +}
```

```
dimacs-literal = {  
    dimacs-atom  
    | element neg {dimacs-atom}  
}
```

```
dimacs-atom = element atom {attribute name {xsd:integer}}
```

where the contents of the `format` attribute are defined as an enumerative datatype with a single member `cnf`. The header `default namespace` contains a URI that acts as a unique identifier for the grammar, and the `datatypes` declaration imports the XML Schema datatype library, binding its namespace to the `xsd` identifier<sup>1</sup>.

This notation defines a well-formed RELAX NG grammar for DIMACS formulas. The Evidential Tool Bus, however, uses *judgements* as the atoms of its communication procedure, i.e., statements of the form  $\Gamma \vdash \Delta$ . In our example, we assume that both antecedent and consequent are sets of DIMACS formulas formula, thus resulting in statements such as:

$$(A \vee \neg B) \wedge B, (\neg B \vee \neg C) \vdash (A \vee \neg C) \wedge (A \vee C \vee \neg A)$$

Hence we encapsulate the aforementioned definition of formulas within a simple judgement definition, yielding the final syntactic definition.

```
default namespace = "http://etb.csl.sri.com/ns/dimacs"  
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"
```

```
start = judgement
```

```
judgement = element sequent {  
  attribute structure {"set"}  
  & element antecedent {dimacs-formula} *  
  & element consequent {dimacs-formula} *  
}
```

```
dimacs-formula = {  
  element comment {text} *,  
  element formula {  
    attribute format {"cnf"},  
    attribute variable_num {xsd:integer},  
    attribute clause_num {xsd:integer},  
    dimacs-clause +  
  }  
}
```

```
dimacs-clause =  
  element clause {dimacs-literal +}
```

```
dimacs-literal = {  
  dimacs-atom  
  | element neg {dimacs-atom}  
}
```

```
dimacs-atom =  
  element atom {attribute name {xsd:integer}}
```

---

<sup>1</sup> Rather than redefining its own, possibly incompatible, datatypes, RELAX NG reuses the facilities already provided by the XML Schema library that predates it.

<i>element identifier</i>	element declaration
<i>attribute identifier</i>	attribute declaration
<i>pattern *</i>	zero or more quantifier
<i>pattern +</i>	one or more quantifier
<i>pattern ?</i>	zero or one quantifier
<i>pattern1   pattern2</i>	disjunction
<i>pattern1 , pattern2</i>	in-order conjunction
<i>pattern1 &amp; pattern2</i>	out-of-order conjunction

Table 1: Common RELAX NG grammar constructs.

Table 1 provides a quick digest of the mostly used RELAX NG constructs. More information, operators, patterns and datatypes can be found in the RELAX NG reference documents, books such as [20], and examples that reside in the `lib/edf/` directory of the Evidential Tool Bus installation.

## 6.2 Semantics

With the previous paragraphs outlining how a RELAX NG grammar can be used to describe the syntactic characteristic of the exchange languages in the ETB, this section now addresses the problem of specifying the semantic contents of such languages. In effect, this means describing how to interpret the various judgements of the language; for instance, judgements in classical propositional sequent calculus enjoy the following property: *if, for all sets of propositions  $\Gamma$  and  $\Delta$ , and for all propositions  $A$  and  $B$ ,  $\Gamma, A \vdash B, \Delta$ , then  $\Gamma \vdash A \Rightarrow B, \Delta$* . The semantics of the ‘ $\vdash$ ’ relation are often given in the form of such inference rules. In [1], Avron proposes an equivalent definition, also proceeding by case analysis on the different kinds of logical connectors, but clearly separating the asymmetric reflexivity and transitivity rules from the bidirectional connector rules. For instance, reusing our example from propositional sequent calculus, the following table summarizes the equations defining the judgement relation in that setting:

Inference	Equivalence
$\Gamma, A \vdash A, \Delta$	
<i>if <math>\Gamma, A \vdash \Delta</math> and <math>\Gamma \vdash A, \Delta</math> then <math>\Gamma \vdash \Delta</math></i>	
<i>if <math>G, A \vdash B, \Delta</math> then <math>\Gamma \vdash A \Rightarrow B, \Delta</math> ;</i> <i>if <math>\Gamma \vdash A, \Delta</math> and <math>\Gamma, B \vdash \Delta</math> then <math>\Gamma, A \Rightarrow B \vdash \Delta</math></i>	$G, A \vdash B, \Delta$ <i>iff</i> $\Gamma \vdash A \Rightarrow B, \Delta$

Proving that both representations are equivalent requires orienting the bidirectional rules, yielding two inferences; then using the transitivity rule to restore the subformula property for the affected inference. In the case of the previous example, the equivalence

$$G, A \vdash B, \Delta \text{ iff } \Gamma \vdash A \Rightarrow B, \Delta$$

becomes

$$\text{if } G, A \vdash B, \Delta \text{ then } \Gamma \vdash A \Rightarrow B, \Delta \text{ and if } \Gamma \vdash A \Rightarrow B, \Delta \text{ then } G, A \vdash B, \Delta$$

where the first of these rules has the subformula property, the second does not. However, using the second rule, the judgement  $\Gamma, A \Rightarrow B \vdash A \Rightarrow B, \Delta$ ,

true by reflexivity, entails  $\Gamma, A \Rightarrow B, A \vdash B, \Delta$ . This in turn, implies  $\Gamma, A \Rightarrow B \vdash \Delta$ , by double application of the transitivity rule, and assuming the two judgements  $\Gamma \vdash A, \Delta$  and  $\Gamma, B \vdash \Delta$  hold. Therefore the inference rule *if  $\Gamma \vdash A, \Delta$  and  $\Gamma, B \vdash \Delta$  then  $\Gamma, A \Rightarrow B \vdash \Delta$*  can be deduced from the bidirectional implication rule, the reflexivity, and transitivity rules. Conversely, this inference rule entails the provability of  $\Gamma, A, A \Rightarrow B \vdash B, \Delta$ , allowing us to demonstrate  $\Gamma, A \vdash B, \Delta$  from  $\Gamma \vdash A \Rightarrow B, \Delta$ , using the transitivity rule.

Avron's representation allows for a compact formulation of the semantics of the judgement relation, where instead of two inference rules, only one equivalence relation is associated with each connector. It also eliminates the redundancy of having the reflexivity and transitivity rules embedded in half of the connector inference rules.

This representation is used to provide the semantics of the judgements exchanged on the Tool Bus. Taking advantage of RELAX NG's annotation mechanism, statements about the semantic nature of the connectives are added on top of the syntax definition mechanism. This takes place through the language's ability to embed XML statements into the grammar definitions, and the use of a predefined namespace called the *SRI Core Logic Initiative* (SCLI), providing the meta-logical structure for these statements. For instance, the negation connector in the syntactic definition of a DIMACS literal can be annotated with its corresponding bidirectional rule  $\Gamma \vdash \neg A, \Delta$  *iff*  $\Gamma, A \vdash \Delta$ :

```
dimacs-literal = {
  element-atom
  | [
    sc:property [
      sc:iff [
        sc:cr [
          sc:lhs [sc:context ["G"]]
          sc:rhs [formula [clause [neg [sc:formula ["atom"]]]]
                sc:context ["D"]]
        ]
        sc:cr [
          sc:lhs [sc:context ["G"]
                formula [clause [sc:formula ["atom"]]]]
          sc:rhs [sc:context ["D"]]
        ]
      ]
    ]
  ]
  element neg {
    dimacs-atom
  }
}
```

where the identifier `sc` is bound to the SRI Core Logic Initiative namespace. While having this kind of annotations in XML format allows it to inherit the language's advantages, conciseness clearly is not one of them. Hence we defined a pretty-printing scheme to allow for a more readable representation of the semantic annotations. Using this scheme, the formalization of the negation connector reads:

```

dimacs-literal = {
  element-atom
  | element neg {
    # scli: G |- formula(clause(neg(atom))), D
    <==> G, formula(clause(atom) |- D
    dimacs-atom
  }
}

```

where ‘<==>’ denotes the *iff* meta-equivalence construct, ‘|-’ is used to build judgements respectful of the syntactic definition of sequents as per section 6.1, and the letters G and D, with an optional digit following, are reserved for context identifiers. The annotations can then be extended to the whole DIMACS grammar definition file, as follows:

```

default namespace = "http://etb.csl.sri.com/ns/dimacs"
datatypes xsd = "http://www.w3.org/2001/XMLSchema-datatypes"

```

```

# scli: revised      2008-06-12
# scli: status       experimental
# scli: shelf-life   2008-12-31

```

```

start = judgement

```

```

judgement = element sequent {
  attribute structure {"set"}
  & element antecedent {dimacs-formula} *
  & element consequent {dimacs-formula} *
}

```

```

dimacs-formula = {
  element comment {text} *,
  element formula {
    # scli: G |- formula(C1,...,Cn), D
    <==> G |- formula(C1), D
    && ...
    && G |- formula(Cn), D
    attribute format {"cnf"},
    attribute variable_num {xsd:integer},
    attribute clause_num {xsd:integer},
    dimacs-clause +
  }
}

```

```

dimacs-clause =
  element clause {
    # scli: G |- formula(clause(A1,...,An)), D
    <==> G|- formula(clause(A1)),
    ...,
    formula(clause(An)), D
    dimacs-literal +
  }

```

	Pretty-print	RELAX NG XML
Judgement	-	sc:cr
Equivalence	<==>	sc:iff
Conjunction	&&	sc:and
Disjunction		sc:or
Universal quantification	all(.).	sc:all
Existential quantification	ex(.).	sc:ex
Substitution	.[<-.]	sc:subs
Context variables	G, G1, ..., D, D1, ...	sc:context
Formula variables	A, B, t, ...	sc:formula

Table 2: SRI Core Logic constructs.

```

dimacs-literal = {
  element-atom
  | element neg {
    # scli: G |- formula(clause(neg(atom))), D
    <==> G, formula(clause(atom)) |- D
    dimacs-atom
  }
}

dimacs-atom =
  element atom {
    attribute name {xsd:integer}
  }

```

where ‘&&’ represents the meta-conjunction construct *and*. The fields **revised** and **shelf-life** are dates in YYYY-MM-DD format, and the **status** is one of **official**, **experimental**, **private**, or **obsolete**. These headers are directly inspired by the ones found in the OpenMath Content Dictionaries [4].

The SCLI meta-framework relies on classical first-order logic. While a comprehensive discussion of this design decision is out of the scope of this manual, it suffices to note that first-order logic is a trusted, well-understood formalism, and if need be, can easily be extended with paradigms from computational facilities [7] to proof term generation [10, 19, 2]. Table 2 summarizes the SCLI notations; more examples can be found in the `lib/edf/` directory of the Evidential Tool Bus distribution.

### 6.3 Schematic

Figure 2 provides an overview of the language definition processing flow. The resulting files are stored in the language database as triples, associated with the corresponding syntax or semantics tag, and agent name.

## References

- [1] Arnon Avron. Simple consequence relations. *Information and Computation*, 92(1):105–139, 1991. 7

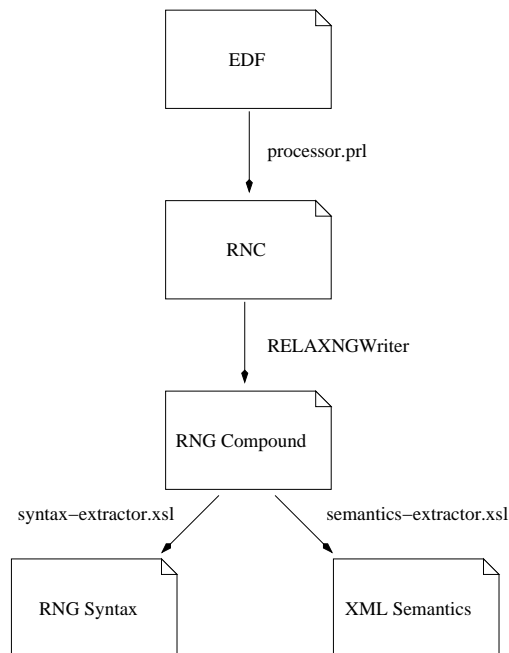


Figure 2: Processing flow for “.edf” data files.

- [2] Franco Barbanera and Stefano Berardi. A symmetric lambda calculus for classical program extraction. *Information and Computation*, 125(2):103–117, 1996. 10
- [3] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Natarajan Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center. 1
- [4] Stephen Buswell, Olga Caprotti, David Carlisle, Mike Dewar, Marc Gaëtano, and Michael Kohlhase. *The OpenMath Standard*. The OpenMath Society, 2.0 edition, June 2004. 10
- [5] James Clark. RELAX NG specification. Organization for the Advancement of Structured Information Standards (OASIS), Committee Specification, December 2001. 4
- [6] Ewen Denney. A prototype proof translator from HOL to Coq. In Mark Aagaard and John Harrison, editors, *Proc. 13th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2000. 1
- [7] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo, revised version. Rapport de Recherche 4861, Institut National de Recherche en Informatique et en Automatique, July 2003. 10
- [8] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert Jones,

editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 2006. 1

- [9] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2006. 1
- [10] Hugo Herbelin. *Séquents qu'on Calcule*. PhD thesis, Université Paris VII, January 1995. 10
- [11] Douglas Howe. Importing mathematics from HOL into Nuprl. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Proc. 9th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 1996. 1
- [12] Cliff Jones, Peter O’Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006. 2
- [13] David Martin, Adam Cheyer, and Douglas Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128, January 1999. 1, 2
- [14] Pavel Naumov, Mark-Oliver Stehr, and José Meseguer. The HOL/NuPRL proof translator: A practical approach to formal interoperability. In Richard Boulton and Paul Jackson, editors, *Proc. 14th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer-Verlag, 2001. 1
- [15] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *Proc. 3rd Int. Joint Conf. on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2006. 1
- [16] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992. 1
- [17] John M. Rushby. Harnessing disruptive innovation in formal verification. In *Proc. 4th IEEE Int. Conf. on Software Engineering and Formal Methods*, volume 0, pages 21–30. IEEE Comp. Soc. Press, 2006. 1
- [18] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV ’99*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer-Verlag, 1999. 1
- [19] Christian Urban. Strong normalisation for a gentzen-like cut-elimination procedure. In Samson Abramsky, editor, *Proc. 5th Int. Conf. on Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 415–429. Springer-Verlag, 2001. 10

[20] Eric van der Vlist. *RELAX NG*. O'Reilly & Associates, Inc., 2004. 7