Appears in *Proceedings of VMCAI2013*, Springerlink. The final publication is available at http://link.springer.com/chapter/10.1007%2F978-3-642-35873-9_18. Its DOI is 10.1007/978-3-642-35873-9_18.

Tool Integration with the Evidential Tool Bus*

Simon Cruanes[1], Gregoire Hamon[2], Sam Owre[2], and Natarajan Shankar[2]

[1] Ecole Polytechnique, Palaiseau, France
[2] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
`simon.cruanes.2007@polytechnique.org`
`{Hamon, Owre, Shankar}@csl.sri.com`

Abstract. Formal and semi-formal tools are now being used in large projects both for development and certification. A typical project integrates many diverse tools such as static analyzers, model checkers, test generators, and constraint solvers. These tools are usually integrated in an *ad hoc* manner. There is, however, a need for a tool integration framework that can be used to systematically create workflows, to generate claims along with supporting evidence, and to maintain the claims and evidence as the inputs change. We present the Evidential Tool Bus (ETB) as a tool integration framework for constructing claims supported by evidence. ETB employs a variant of Datalog as a metalanguage for representing claims, rules, and evidence, and as a scripting language for capturing distributed workflows. ETB can be used to develop assurance cases for certifying complex systems that are developed and assured using a range of tools. We describe the design and prototype implementation of the ETB architecture, and present examples of formal verification workflows defined using ETB.

Keywords: Formal techniques, Hybrid techniques, Certification, Tool integration, Workflow

1 Introduction

Formal techniques such as model checking, static analysis, and theorem proving are playing an increasingly prominent role in the design and analysis of complex hardware and software systems [16,26]. These techniques are typically employed within workflows that use multiple tools and techniques. Such a workflow might employ a combination of test case generation and static analysis to identify bugs, synthesis to generate correct code, and verification to establish correctness properties. Since formal techniques are typically used to achieve a high degree of assurance for certifying the validity of the software, it is important to extract evidence for any formal claims associated with the software. This evidence should be explicit so that it can be examined, replayed, and maintained

* This work was supported by NSF Grant CSR-EHCS(CPS)-0834810 and NASA Cooperative Agreement NNX08AY53A. [**Note to Program Committee:** The paper addresses issues of hybrid methods, tool integration, and certification that are central themes for VMCAI. We hope it will be considered within scope for this reason. A tarball of the ETB implementation can be made available for evaluation on request.]

even as the constituent components are modified. The idea of an Evidential Tool Bus (ETB) was proposed by Rushby [24]. We have recently designed and implemented an architecture for ETB as a distributed framework for integrating diverse tools into coherent workflows for producing claims supported by explicit evidence. We motivate the need for such an architecture and describe the design, implementation, and application of ETB.

Recent years have witnessed an explosion in the growth of formal verification tools. We now have a range of powerful formal tools for specialized tasks, but a typical application of formal techniques employs multiple tools. Examples of such integrated formal techniques include

1. *Counterexample-guided abstraction refinement (CEGAR)* combining model checking and abstraction, which is itself typically implemented using a satisfiability solver [7].
2. *Concolic execution* combining symbolic evaluation with a constraint solver for generating test cases [25].
3. *Hardware verification toolchains* that exploit different representations including hardware description languages and Boolean representations such as and-inverter graphs, binary decision diagrams, and conjunctive normal form [5].
4. *Deductive program verification* to generate proof obligations that are then discharged using a theorem prover [19].
5. *Invariant generation* combining static analysis, templates, dynamic analysis, k-induction, and property-directed reachability [8,13,4].

These examples employ several tools within a script to produce formal claims that either verify software properties or produce concrete counterexamples. Quite often, these tools have specific platform requirements so that any framework for combining them must support multiple diverse platforms.

Software and system certification is another motivation for ETB as a framework for the construction and maintenance of arguments and evidence. A formally supported software design lifecycle includes several phases from requirements and specification to design and implementation to testing and integration. These phases are connected to each other through formal and semi-formal claims. The resulting system is often accompanied by an assurance case [3], which is *"a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a systems properties are adequately justified for a given application in a given environment."* Such an assurance case is an argument that employs claims that are supported by evidence, where some of the evidence is the result of applying individual tools.

The main purpose of ETB is the production of claims supported by arguments based on evidence. Some sub-claims in the argument can be established by external tools. This yields several desiderata that guide the design of the evidential tool bus.

1. ETB should be extensible with new claim forms and rules of argumentation.
2. It should admit new external tools (including human oracles) through an API to produce claims and generate queries.

3. New workflows should be definable as scripts.
4. It should be possible for a completed development to be replayed and checked, relative to assumptions.
5. The resulting argument should explicate the tools and assumptions on which it depends.
6. There should be support for maintaining an argument even as the underlying tools and inputs change.

The most important design principle for ETB is *semantic neutrality.* The framework should not be biased toward any specific tools, languages, models, or applications. Any semantic interpretation is provided solely by the tools. The ETB merely provides the plumbing and book-keeping so that tools can communicate with each other to exchange services while recording the resulting claims and evidence.

Given these considerations, we have designed ETB as a distributed client-server architecture where the coordination between services is defined using a variant of the Datalog programming language [1,6]. We use Datalog both as a metalanguage for representing claims, inference rules, and arguments, as well as a scripting and coordination language for capturing distributed workflows. Datalog was initially conceived as a database query language, but in recent years, it has been used for other interesting applications such as static analysis, declarative networking, and verification [**?**,**?**]. In our variant of Datalog, claims are expressed using Datalog predicates. We incorporate external tool invocations through *interpreted* predicates. These external tools are available as services provided by one or more servers on the ETB network. At these servers, tool *wrappers* are defined to map Datalog queries of these predicates to the corresponding tool invocation and bind the results back to variables in the query. A typical interpreted query would be $\mathit{minisatCheck}(Formula, Result)$, which invokes the MiniSAT [12] SAT solver on the given formula to bind the variable *Result* to either `sat` or `unsat`. Claims can also be expressed using uninterpreted predicates. For example, we might have a more generic satisfiability query given by $\mathit{satisfiable}(Formula, Result)$ which can be defined to invoke one of several SAT solvers such as MiniSAT. Datalog programs are defined as Horn clauses, where the head atom is expressed using an uninterpreted predicate. A subset of these program clauses can be identified as *inference rules* that are admissible as steps within an argument, while the remaining clauses are used as scripts to direct the workflow.

ETB features a distributed architecture for processing Datalog queries. An ETB network is a fully-connected graph of ETB servers operating on a local area network. ETB nodes or networks running outside of the local network can be integrated through proxy servers. Each server advertises the tool services that it offers on the network. The ETB network offers a client API that can be used to define, initiate, and monitor computations on the ETB network as well as to maintain the structure of the network itself. An ETB network can be used simultaneously by several clients. Each server also maintains a claims table and a version-controlled file system. The claims table monitors the status of queries and records claims together with the associated inference steps. The claims are about data, some of which is maintained in files. It is important to note that the claims are actually about (the contents of) file versions, so that a claim about one version may not be valid about other versions of the same file. ETB can be used to maintain the validity of claims and arguments as file versions are modified.

Failure is a pervasive problem with distributed applications, and ETB is no exception. ETB server nodes can go down, communication links can fail, and tool invocations can trigger errors. Such erroneous computations might corrupt the data in files. We have designed the ETB engine to allow computations to be tracked. Errors lead to ETB claims that can be logged or can trigger notifications. A version-controlled file system ensures that corrupted files are not committed, and that clean versions of files can be recovered.

ETB has applications beyond supporting workflows for formal verification and certification. ETB can be used as a middleware platform for orchestrating distributed computations that involve the production and maintenance of claims. Examples include any scientific workflow where it is useful to maintain information about the provenance of files. Another such application is a distributed `make` (described in Section 4) that keeps system builds synchronized with changes to the source code files. Here, claims about the relationship between source and binary files, as well as configuration details, are used to direct the build process. Unlike the Unix utility `make`, the ETB version is sensitive only to file contents and not their write dates. Another application is in tying together tools that run only on specific platforms, e.g., Windows, Linux, iOS, and Android. ETB can also be used to distribute tasks such as regression testing that can be easily parallelized and distributed across a network of servers.

It is important to identify the problems that are not addressed by ETB. It is meant for coarse-grained integration of tasks over a distributed network, and not for fine-grained integration as in a Nelson–Oppen combination [20] of inference procedures. The integration in ETB is especially suited to workflows where the individual tools are employed on large tasks where the inputs and outputs can be saved to and read from files. In such workflows, much of the data saved in files is part of the evidence that has to be preserved along with any associated claims for certification purposes. While ETB can be used as a general-purpose framework for coarse-grained distributed computing, the main advantage it offers over task distribution frameworks like Condor [27] or Hadoop is its ability to retain evidence together with reproducible claims. ETB, by itself, is not a semantic interchange framework like PROSPER [10], ToolBus [9], or Veritech [14] where different tools interact through a common interchange language, but it could be used to implement such a framework. ETB has some similarity to service-oriented architectures such as ETI/JETI [18] or Orc [17] which can also be used to define workflows, but unlike ETB, these do not support the construction of version-controlled, evidence-based arguments. Dedalus[2] is a time and space-aware declarative language that extends Datalog for declarative networking and computation. It explicitly embeds a discrete notion of time to perform atomic updates on the set of extensive set of atoms considered true at any moment. Unlike ETB, whic supports the exchange of files and tool services, the operational semantics of Dedalus only supports the exchange of Datalog atoms between nodes ("locations"). ETB is also different from metalogical frameworks like LF [15], Isabelle [21], and Twelf [22]. These frameworks are used to represent formal object logics and support proof construction. In contrast, ETB is neutral about syntactic representations and semantic interpretations and focuses purely on the interaction between tools through scripts, queries, and claims. The design of ETB has been influenced by ideas from these projects, but the focus has been on semantically neutral, evidence-based tool integration.

We outline the use of Datalog as the metalanguage of ETB in Section 2, and describe the ETB architecture in Section 3. In Section 4, we illustrate the workings of ETB with some examples, and conclude with a discussion of future work in Section 5.

2 Datalog as a Metalanguage

Datalog was initially introduced as a deductive database language, but in recent years, it has been employed as a declarative framework for other applications such as program analysis and networking. We describe how ETB uses a variant of Datalog to represent data, claims, queries, inference rules, workflows, and arguments. We illustrate our variant of Datalog with an example. The key ideas in the representation of ETB artifacts are

1. Data in the ETB consists of JSON objects, file handles, and tool session handles.
2. An ETB atom is an n-ary predicate applied to n arguments that are either data objects or variables.
3. A claim is a ground (i.e., fully instantiated) atom or its negation.
4. A query atom is a partially instantiated atom where some of the arguments can be variables. A query is a sequence of query atoms.
5. Information in the ETB is exchanged in the form of *queries* and *claims* about these data objects. Files related to the queries and claims are also exchanged across the ETB network.
6. External tools are integrated into ETB through *interpreted* predicates.
7. Scripts are ETB programs built from *derivation rules*, i.e., Horn clauses where the head atom contains an *uninterpreted* predicate.
8. *Inference rules* are a subset of the derivation rules that are used to construct *exportable* proofs of claims from those of sub-claims. Such proofs may need to satisfy constraints on the subset of external tools and rules that can be used.
9. An ETB proof is a tree of claims: each claim follows from the subclaims by an inference rule.
10. A query triggers backward chaining on derivation rules to instantiate a workflow. A successful computation yields a set of claims instantiating the query, and each such a claim is supported by one or more derivations.
11. Forward chaining is employed on inference rules to derive claims and construct exportable proofs.

12. Multiple derivations (including exportable proofs) can be saved. These derivations are used to recreate claims when the input data has changed. Multiple exportable proofs are also useful in strengthening the validity of a claim.

We describe each of these aspects of ETB's use of Datalog as a metalanguage.

Data. The data processed by ETB can include programs, transition systems, formulas, files, contexts, models, test cases, analysis results, and tool sessions. ETB provides no built-in interpretation of the data beyond what is provided by these tools. Among the tools in the ETB are translators from one data representation to another. For example, an ETB tool might translate formulas generated by a hardware description language to the input language of a model checker. ETB data is represented as a JSON object, i.e., a Boolean, number, structure, or array. File and tool handles are represented as JSON structures. File handles can contain metadata, but are uniquely identified by the hash of the file contents. This ensures that any claims about files are about their contents rather than their file IDs or other metadata. Tool sessions are also similarly identified along with any metadata, but are uniquely identified by a session ID and a hash of the session state that is maintained by the tool wrapper. ETB can be extended with new forms of data, but the above forms are sufficient for most applications.

Claims. The notion of a claim is central to ETB. The whole point of ETB is to produce a replayable argument for a claim that is supported by evidence. Such an argument can include sub-claims generated by external tools. Claims are given in the form of judgements that are applied to data. A claim is a ground ETB atom of the form $p(d_1, \ldots, d_n)$, where p is an n-ary predicate and $d_1, \ldots, d_n$ is a sequence of n data elements. A predicate p can either be an *uninterpreted* predicate or an *interpreted* predicate. The latter class of predicates is mapped to tool invocations, whereas the former corresponds to conventional Datalog predicates. In describing these claims, we make reference to external tools such as the PVS theorem prover, the Yices SMT solver, and the SAL model checker. Simple examples of judgements used to make claims include:

1. $\mathit{pvsFormula}(P)$: P is a PVS formula
2. $\mathit{pvs2Yices}(P, Y)$: Y is the Yices translation of PVS formula P
3. $\mathit{yicesModel}(Y, C)$: C is a model for Yices formula Y
4. $\mathit{yicesUnsat}(Q)$: Q is an unsatisfiable Yices context
5. $\mathit{salModule}(M)$: M is a SAL transition system module
6. $\mathit{invariant}(M, P)$: P is an invariant property of module M

The actual claim predicates might take additional arguments such as tool version or auxiliary arguments.

Most of the predicates in the above list of claims are interpreted, i.e., they are directly implemented by invoking external tools. A predicate like *invariant* is uninterpreted in that it is defined by means of ETB rules. In Datalog, the predicates are partitioned as (extensional) database predicates and (intentional) defined ones. In ETB, the extensional predicates are treated as interpreted since the database can be regarded as an external tool. Another side-effect of using interpreted predicates is that we can recover the power of Prolog-style unification through external tool invocations. Datalog treats the data elements as unstructured, whereas Prolog can use unification to decompose a term of the form $f(X, Y)$. With interpreted predicates, we can replace a goal formula of the form $p(f(X, Y))$ by $p(Z)$, $\mathit{decompose}_f(Z, X, Y)$, where the interpreted predicate $\mathit{decompose}_f$ holds exactly when $Z = f(X, Y)$. A program clause of the form $p(f(X, Y))$:- $q(X, Y)$ can be replaced by $p(Z)$:- $\mathit{decompose}_f(Z, X, Y), q(X, Y)$.

Claims in ETB are not restricted to logical assertions. For example, a claim can provide statistical information such as the percentage of coverage provided by a test suite. Claims can even be speculative, as in the assertion that a formula is a potential invariant as generated by a tool like Daikon [13].

Queries. A query is a non-empty sequence of partially instantiated ETB atoms of the form $p(a_1, \ldots, a_n)$ where some arguments a_i can be variables. If $\{X_1, \ldots, X_n\}$ is the set of free variables in the query sequence $A_1, \ldots, A_k$, and K is the Datalog program, then a Datalog computation searches for a set of instantiations of the form $[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n]$, where each b_i is a data object, such that $K \implies A_1[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n] \wedge \cdots \wedge A_k[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n]$. When this is the case, we can assert $A_i[X_1 \mapsto b_1, \ldots, X_n \mapsto b_n]$ as a claim, for i in $\{1 \ldots k\}$.

Queries in ETB are used to trigger computations to establish the corresponding claims. Typical queries might include

1. Translation from one language, e.g., PVS to Yices: $\mathit{pvs2Yices}(`x + y < 3', Y)$
2. Type Checking: $\mathit{pvsTypecheck}('x : \mathit{real}, y : \mathit{int}', '(x + y)/(x - y)', T, Q)$, where the first argument is the context, and the output T is the type of the expression, and output Q is the set of proof obligations, e.g., $x - y \neq 0$.
3. Satisfiability checking: *yicesUnsat*(*filehandle*), where the file handle is provided by some other tool, e.g., the translator from PVS.

Scripts. An ETB program is collection of Horn clauses, where each clause is either a derivation rule of the form H :- C or an inference rule of the form $H \Leftarrow C$, where H is an atom, and C is a list of atoms such that every free variable in H also occurs free in some atom of C. In ETB, the predicate symbol for the head atom, the head predicate, must be uninterpreted, and the set of clauses in which p occurs as the head predicate constitutes the definition of p. As in Prolog and Datalog, variables are represented by identifiers where the leading character is in upper case, whereas the leading character for identifiers for constants is in lower case. For example, one could write the following reachability script for a transition system using a BDD package that checks if some bad state satisfying P is reachable by a transition system M. In this script, we first extract the initialization predicate I and the transition relation N from module M, and then check if P is reachable with I and N. The predicate *reachable* is defined to construct the BDD versions of I, N, and P as L, W, and Q, and to invoke the predicate *bddReachable* on these. The latter predicate is defined to iteratively compute the image of W with respect to L as J, and then check the reachability of Q from J.

$$\begin{aligned}
\mathit{reach}(M,P) &\text{ :- } \mathit{init}(M,I), \mathit{transition}(M,N), \mathit{reachable}(I,N,P).\\
\mathit{reachable}(I,N,P) &\text{ :- } \mathit{bdd}(I,L), \mathit{bdd}(N,W), \mathit{bdd}(P,Q), \mathit{bddReachable}(L,W,Q).\\
\mathit{bddReachable}(L,W,Q) &\text{ :- } \mathit{bddAnd}(L,Q,R), \mathit{bddNonempty}(R).\\
\mathit{bddReachable}(L,W,Q) &\text{ :- } \mathit{bddImage}(W,L,J), \mathit{bddReachable}(J,W,Q).
\end{aligned}$$

Semantics. We describe the distributed evaluation of ETB queries in Section 3.2. We focus here on the semantics of the Datalog variant used by ETB. With external tool invocations, the data domain is no longer finite. For example, a simple successor relation $\mathit{succ}(x,y)$ which holds exactly when $y = x + 1$ would generate an infinite set. We assume that the interpreted queries are *bounded*, so that $p(a,X)$ binds X to a list of instantiations for X. The processing of interpreted queries through a tool wrapper can trigger back-chaining, so that the query $p(a,X)$ can introduce the query $q(b)$. Our semantics is proof-theoretic. Given a program $\mathit{\Pi}$ and a query atom Q, the meaning $\mathit{\Pi}[\![Q]\!]$ we find every instance $\underline{Q}$ of Q for which there is a SLD-resolution proof of $\underline{Q}$ from $\mathit{\Pi}$. There can be unboundedly many such instances, as for example for the query $\mathit{lt}(0,y)$ in the program below.

$$\begin{aligned}
\mathit{lt}(x,y) &\text{ :- } \mathit{succ}(x,y).\\
\mathit{lt}(x,y) &\text{ :- } \mathit{lt}(x,z), \mathit{lt}(z,y).
\end{aligned}$$

In such cases, the computation diverges.

Inference Rules. Some of the clauses in the ETB programs can be designated as inference rules that are sanctioned for use in ETB proofs. For example, in the inference rules below, P is asserted to be an invariant for a transition module with initialization I and transition relation N, if it is implied by another invariant, or it is an inductive invariant. The premises of the rule involving the predicates *implies* and *inductiveInvariant* can be established by other rules or by means of tool invocation.

$$\begin{aligned}
\mathit{invariant}(I,N,P) &\Leftarrow \mathit{invariant}(I,N,Q), \mathit{implies}(Q,P).\\
\mathit{invariant}(I,N,P) &\Leftarrow \mathit{inductiveInvariant}(I,N,P).
\end{aligned}$$

Inference rules are ETB program clauses and can also be used as scripts. Other examples of inference rules include

1. Counterexample traces: A valid trace is one that leads from an initial state to an error state on zero or more computation steps.
2. Abstraction: If one program is an abstraction of a (concrete) program, then the concrete counterpart of a property of the abstract program holds of the concrete one.
3. Refinement: A concrete program is a refinement of an abstract one if any concrete computation step can be simulated by the abstract program through

the simulation relation. Separate inference rules can be specified for the different forms of refinement.
4. Termination: A `while`-loop terminates if there is a ranking function on the variables that decreases with each execution of the loop. Other termination techniques, e.g., disjunctive well-foundedness of the transitive closure of the transition relation, can also be specified [**?**].

Proofs. A proof is a derivation, a tree whose root node is a claim that is an instance of the head atom of an inference rule, where the corresponding instances of the body atoms are proved by the subtrees. Typically, an interpreted claim is the leaf node of a proof, but it is possible for the proof of an interpreted claim to also have sub-proofs corresponding to queries dynamically generated during the tool invocation. In this case, the validity of the inference step relative to the sub-proofs is checked by the tool itself. In some cases, the tool might be somewhat nondeterministic so that the proof cannot be exactly replicated. In such cases, the proof is merely used as a template for reconstructing the steps in the justification. ETB proofs are thus a subset of ETB derivations where the proof steps only employ program clauses that are designated as inference rules. The proof construction feature has not yet been implemented in ETB.

2.1 An Example: Iterated k-Induction

Given a transition system $M = \langle I, N\rangle$ with initialization I and transition N, and a predicate P, we want to prove that P is an invariant by finding the smallest k such that P is k-inductive. The inference rule for demonstrating that P is an invariant of M is given below. It asserts that one possible way of demonstrating the P is an invariant of M is by establishing that P is a k-inductive. (There are other possible ways, such as by showing that P is entailed by some other invariant Q.)

$$\begin{aligned} \mathit{invariant}(M,P) \Leftarrow\ & \mathit{transitionSystem}(M),\\ & \mathit{initialization}(M,I),\\ & \mathit{transition}(M,N),\\ & \mathit{kInductive}(I,N,P,K). \end{aligned}$$

The inference rule for k-inductivity decomposes the claim into the subclaims that P holds for the first K steps (i.e., the bounded model checking claim *bmc* holds of I, N, P, K), and that if P holds for any K states in a computation, then it holds for the $K+1$st state of the computation.

$$\begin{aligned} \mathit{kInductive}(I,N,P,K) \Leftarrow\ & \mathit{bmc}(I,N,P,K),\\ & \mathit{inductionStep}(N,P,K). \end{aligned}$$

The above inference rules are used to build ETB proof, but the actual value of K is found by a script defining the predicate *iterInductive* that tries each value K starting from 1 up to some upper bound U using the tool invocation

$in\_range(K, 1, U)$. We can also define script for proving $invariant(M, P)$ by invoking the script for *iterInductive* to find a suitable K for some fixed upper bound, e.g., 5.

$$\begin{array}{rl} iterInductive(I, N, P, K, U) \text{ :- } & in\_range(K, 1, U), kInductive(I, N, P, K). \\ invariant(M, P) \text{ :- } & transitionSystem(M), \\ & initialization(M, I), \\ & transition(M, N), \\ & iterInductive(I, N, P, K, 5). \end{array}$$

The predicates *bmc* and *inductionStep* invoke Yices, by first invoking an interpreted predicate *yicesBmcFormula* to construct the bounded model checking query formula F, and then invoking the interpreted predicate *yicesUnsat* to determine if F is unsatisfiable using Yices.

$$\begin{array}{rcl} bmc(I, N, P, K) & \Leftarrow & yicesBmcFormula(I, N, P, K, F), \\ & & yicesUnsat(F). \end{array}$$

The induction step generates the k-induction formula F using the interpreted predicate *yicesInductionFormula*, and checks for unsatisfiability invoking the interpreted predicate *yicesUnsat* as above.

$$\begin{array}{rcl} inductionStep(N, P, K) & \Leftarrow & yicesInductionFormula(N, P, K, F), \\ & & yicesUnsat(F). \end{array}$$

We omit the details of the construction of the Yices formulas for BMC and k-induction. If $kInductive(I, N, P, k)$ succeeds for some k, then forward chaining on the rule of inference for *invariant* yields the claim $invariant(M, P)$ with a proof using the inference rules.

3 The Architecture of ETB

ETB is based on a distributed client-server architecture. An ETB cluster consists of a fully connected network of ETB servers as shown in Figure 1. Each server maintains a version controlled file system, specifically Git, and tables of claims and queries. It also advertises a set of services and subscribes to a list of claim forms. A server can itself be a proxy for other ETB clusters, thus serving as a bridge between two or more ETB clusters. An ETB cluster is operated through a client which offers an application programming interface (API). Through a client, a user can configure and program the server, connect other servers to the cluster, add new content, initiate the processing of queries, and monitor the progress of a computation.

We describe some of the key features of the ETB architecture: the software stack, the server, the client API, and the mechanics of distributed query processing in ETB. The current ETB prototype is implemented in Python.

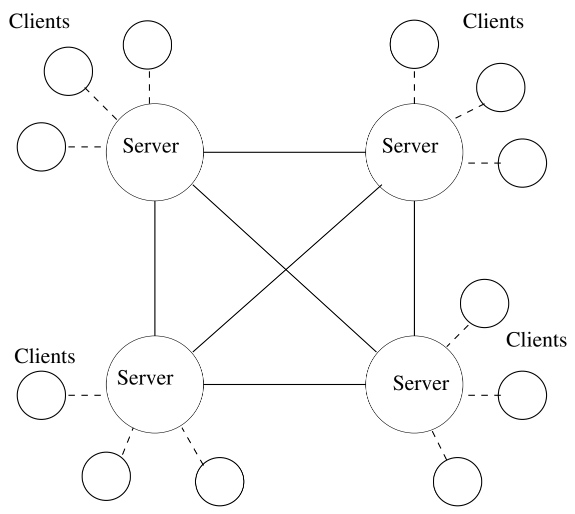

Fig. 1. The ETB Client-Server Architecture

3.1 The ETB Software Stack

The ETB stack consists of the following layers:

1. **Network:** Maintains the connectivity status of the ETB cluster and shares information about tool wrappers and cluster-wide ETB programs.
2. **File:** Each server operates a Git repository from which files/directories are synchronized with servers as needed.
3. **Session:** Servers exchange claims and queries, as needed to invoke external tools.

The Network Layer. Basic to the network layer is a heartbeat. Each server periodically broadcasts a heartbeat message to the other servers in the network. This heartbeat is used to maintain the membership of the network; in case of a server failing, other servers will delete it from the cluster. Servers also share information about the predicate symbols they can interpret, e.g., a server on a Linux machine on which Yices [11] is installed may declare that it can interpret *yicesUnsat*.

When two servers are in the same cluster, they can access one another through the network. XML-RPC calls are used for all communication between servers. For instance, the heartbeat messages are remote invocations of a `ping()` method.

The File Layer. Each server has a local, private Git repository that it uses to store files involved in claims. If several versions of a file are involved in claims, each such version is stored, and referred to by its unique SHA1 hash. The purpose of this storage is two-fold:

1. To replay derivations and proofs of claims, which may require restoring old versions of files
2. Sharing files with other servers by extracting the content of the file (indexed by its SHA1 hash) and sending it through the network layer.

Using Git in this way ensures the *immutability* of claims involving the contents of files. If we state that `yicesUnsat`(*filehandle*), and want to replay it later, the file may have been deleted or modified. However, since the contents of the file is saved in the Git repository, and the claim actually contains the hash of the file, we can create the original file again to replay Yices on it. File contents can therefore be seen as constant values indexed by their SHA1 hash even though the file itself might have been modified in the repository.

The Session Layer. The session layer supports query processing in order to produce claims and evidence. Each server executes scripts locally and only invokes the other servers for external tools. The queries corresponding to the tools are sent to a suitable server using XML-RPC. The target server uses the File layer to obtain the relevant files. The processing of the query at the target server can recursively invoke other queries on the ETB network. Any resulting claims are returned to the source server along with the derivation. The claims are also broadcast to servers that have registered subscriptions associated with this claim. The source server then obtains the files associated with the resulting claims. The source server can choose a target server based on load parameters or other factors. If the target server fails during the processing of a query, the query can be retargeted to a different server. Each server has a consistent copy of the ETB scripts, but a server can also have local scripts where only the service provided by these scripts is exported.

3.2 Query Processing in ETB

We present the details on the operational semantics of query processing through the evaluation of ETB rules. We call *goal* any ETB atom, possibly with variables. Claims are ground atoms with an associated with a *derivation*.

Figure 2 contains a set of *inference rules* that are applied to the query set $\mathcal{S}$ and the claims table T in a nondeterministic order until no rule affecting $\mathcal{S}$ and $\mathcal{T}$ is applicable; the set $\mathcal{S}$ is then *saturated* and contains clauses and justifications that can be used to construct derivations from the rules supporting the claims in $\mathcal{T}$.

- The rule Claim transfers unit clauses to the claims table.
- The rule Application instantiates a rule from $\mathcal{R}$ to the set $\mathcal{S}$ if the rule may help solving the first goal (g_1) of some clause in S.
- The rule Resolution removes the first goal g_1 if $g_1\sigma$ is a claim. It is really a unit resolution inference rule. Note that the justification $c\sigma[c']$ captures the resolution step.
- The rule Interpretation schedules the interpretation of an interpreted atom g_1 if it occurs as the first premise of a clause in $\mathcal{S}$. Intuitively, a tool

Claim
$$\frac{(c : h \text{ :- }) \in \mathcal{S}}{\mathcal{R}, T, \mathcal{S} \Downarrow T + (c : h), \mathcal{S}}$$

Application
$$\frac{(c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \qquad r \equiv a \text{ :- } b_1, ..., b_k \in \mathcal{R} \qquad \exists \sigma . a\sigma = g_1\sigma}{\mathcal{R}, T, \mathcal{S} \Downarrow T, \mathcal{S} + (r\sigma : a\sigma \text{ :- } b_1\sigma, ..., b_k\sigma)}$$

Resolution
$$\frac{(c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \qquad \exists \sigma . (c' : g_1\sigma) \in T}{\mathcal{R}, T, \mathcal{S} \Downarrow T, \mathcal{S} + (c\sigma[c']) : h\sigma \text{ :- } g_2\sigma, ..., g_n\sigma)}$$

Interpretation
$$\frac{(c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \qquad g_1 \text{ interpreted} \qquad I(g_1) = \Sigma, H, Q \qquad Q \subseteq T}{\mathcal{R}, T, \mathcal{S} \Downarrow T + \{(f \text{ :- }^{g_1} Q) : f | f \in H\} + \{(g_1\sigma \text{ :- } Q) : g_1\sigma | \sigma \in \Sigma\}, \mathcal{S}}$$

Query
$$\frac{\begin{array}{c} (c : h \text{ :- } g_1, ..., g_n) \in \mathcal{S} \\ g_1 \text{ interpreted} \qquad q \text{ query generated during interpretation of } g_1 \end{array}}{\mathcal{R}, T, \mathcal{S} \Downarrow T, \mathcal{S} + (g_1 : q)}$$

Fig. 2. Rules for query processing

invocation solves this goal in the hope that it will add claims matching g_1, that will in turn remove g_1 by the Resolution rule. All claims matching g_1 generated by the interpretation I are added to T when all the claims in Q corresponding to recursive queries have been established in T. A tool can also introduce additional claims H to T, and each additional claim f is tagged with the derivation (f :- ^{g_1}Q) to indicate that it was derived during the invocation g_1 and is conditional on Q.

– The rule Query allows new queries to be added to $\mathcal{S}$ during the evaluation of an interpreted atom g_1 by the corresponding external tool.

The rules have the form: $\mathcal{R}, T, \mathcal{S} \Downarrow T', \mathcal{S}'$, meaning that a set of rules $\mathcal{R}$, a set of established claims T, and a set of processing clauses $\mathcal{S}$ produce in one a step a new set of established claims T' and a new set of processing clauses $\mathcal{S}'$. Established claims in T are pairs of a ground claim h and a justifying clause h :- $g_1, ..., g_n$. Processing clauses in $\mathcal{S}$ are pairs of a justifying clause and a clause (the justifying clause will be used to turn derived unit clauses into claims). Claims that are already in the claims table $\mathcal{T}$ are reused, as in tabled logic programming [23], and are not recomputed.

This calculus is trivially sound (the rule Resolution is an instance of unit resolution). It is also complete for well-formed first-order clauses (such that all variables of the head of a clause also occur in the body). Any SLD-resolution refutation proof can be transformed to a derivation in the ETB calculus.

3.3 Tool Wrappers

Tools are connected to the ETB by means of thin wrappers. Wrappers are written in Python and and are short, simple programs that map queries to tool invocations. A wrapper for a tool is a Python class that defines one or more predicates to be interpreted by the ETB. Predicates are given by a triplet: a name, a predicate signature, and a Python function. The name and signature of the predicate are shared among all ETB nodes, informing other nodes that the ETB can now interpret this predicate. The function is local, and is executed when the predicate needs to be interpreted

Predicate Signatures A predicate signature is defined by the following grammar:

predicate signature	sig ::=	$arg\_spec \mid sig, arg\_spec$
argument specification	arg_spec ::=	$sign\, arg : type$
argument type	$type$ ::=	`value` | `file` | `files`
argument sign	$sign$ ::=	$+ \mid -$

The signature associate modes and types to each argument of the predicate. Types are either `value`, which indicates a JSON value, or `file`, which indicates an ETB file references, or `files`, in which case the argument is a list of files. The mode associated to each argument in the signature represents inputs and outputs: a + argument is an input to the predicate and is required to be ground when the predicate is invoked. A - argument indicates an output and can be either a constant or a variable at invocation: if it is a constant, the predicate is expected to check that the predicate holds for this particular value, if it is a variable, the predicate is expected to returns one or several substitution for this variable.

For example, the signature for a predicate invoking the `yices` SMT solver on a file containing a formula and returning either of the strings “sat” or “unsat” could be:

```
yices :: +input: file, -result: value
```

The predicate *yicesUnsat* used earlier can be defined as a separate wrapper or as an uninterpreted predicate using the `yices` predicate.

Implementation of a Wrapper As said before, wrappers are written in Python. More precisely, a wrapper is a Python class that inherits from the class `etb.wrapper.Tool`. This class provides a decorator `predicate` which is used to declare particular methods of the class as predicates as well as give the predicate signature. A template wrapper for the `yices` SMT solver is the following:

```
class Yices(Tool):
    @predicate('+input: file, -result: value')
    def yices(self, formula, result):
      ...
```

It declares the predicate `yices` with the signature we saw previously.

The code of the wrapper then typically calls the tool on the input arguments, gets the results from the tool and translates them for the ETB. The results returned to the ETB when the query succeeds, include a (possibly empty) list of substitutions, additional ground claims, and claims associated with ETB queries dynamically generated during the tool invocation.

The `etb.wrapper.Tool` class provides utilities that help in interpreting and producing ETB values. The `etb.wrapper.BatchTool` class extends it with utilities to make it easy to call external batch tools. Using these utilities, we can write the code of the `yices` wrapper as follows:

```
@Tool.predicate("+input: file, -result: value")
def yices(self, input, result):
    return self.run(result, 'yices', input['file'])
```

The wrapper takes as output argument a value `result`, which can be either a constant or a variable. If it is a constant, the wrapper checks that `yices` produces the same value, and returns an empty substitution representing success. If it is a variable, it returns a substitution binding the variable to `yices` output. If the `yices` command fails for any reason (e.g. `yices` is not found, or the file is not a valid `yices` file, etc.), it returns failure (no substitution), as well as an additional error claim.

3.4 The Client API

Interaction with an ETB network is done through a client. The client can exchange files with the ETB repository. It can also initiate queries, monitor the progress of a query, and access the substitutions and claims established by query evaluation.

The basic client included with ETB is a read-eval-print interaction loop which can interpret scripts interacting with the ETB. Specialized clients can be built for specific applications — a GUI client has been implemented for a hardware analysis project. A shell client that uses the ETB as a backend can be easily built.

A client connects to one server of an ETB network, and communicates with that node using XML-RPC calls. Clients can therefore be written in any language that supports XML-RPC. The main functions of the client API to the ETB are presented in Figure 3 – other functions not listed here enable the client to manage the state of the ETB and observe a visual representation of running queries.

4 ETB Examples

In this section, we present two simple application of the ETB. The first is a distribute implementation of `make` that can also handle multiple platforms, and the second is a naïve AllSAT script. ETB is currently being used in a couple of projects: one, to integrate various hardware analysis tools, and another, to combine model-based design/analysis capabilities.

Method	**Description**
ref = `put_file(`*src*`,` *dst*`)`	Put the file *src* as *dst* on the ETB, returns a reference to the destination.
`get_file(`*srcref*`,` *dst*`)`	Get the file corresponding to the reference *srcref* from the ETB and copy it locally under *dst*.
id = `eval(`*claim*`)`	Create a new query looking for solution for *claim*, returns a query ID.
answers = `wait_query(`*id*`)`	Wait for the query *id* to complete and returns its set of answers.
answers = `query_answers(`*id*`)`	Returns the current set of answers for the query *id*.
claims = `query_claims(`*id*`)`	Returns the current set of claims generated as part of the query *id*.
ret = `connect(`*host*`,` *port*`)`	Connect the remote ETB node running at *host:port* with the local ETB.

Fig. 3. The ETB Client API

4.1 A Simple Distributed Make

In our first example, we use the ETB to build a binary executable from source C files as would typically be done using `make` or similar tools.

C Compiler Wrappers We first define a wrapper for `gcc`. This wrapper declares two predicates, one to compile a C source file into an object file, the other one to link several object files together into a binary.

```
@Tool.predicate("+src: file, +dependencies: files, -obj: file")
def gcc_compile(self, src, deps, obj):
   dst = os.path.splitext(src['file'])[0] + '.o'
   self.callTool('gcc', '-c', '-o', dst, src['file'])
   objref = self.fs.put_file(dst)
   return self.bindResult(obj, objref)
```

The first wrapper takes a source file, a list of files that the compilation depends on, typically header files, and returns an object file. We first create the output file name, then run `gcc`. Finally, we put the generated object file on the ETB and bind the output variable to this file. Note that the dependencies do not appear in the command that is run, but appear in the claim, and as such the files are copied to the server that handles the compilation together with the source file.

```
@Tool.predicate("+ofiles: files, +exename: value, -exe: file")
def gcc_link(self, ofiles, exename, exe):
   filenames = [ r['file'] for r in ofiles ]
   args = [ 'gcc', '-o', str(exename) ] + filenames
   self.callTool(*args, env=env)
   exeref = self.fs.put_file(str(exename))
   return self.bindResult(exe, exeref)
```

The second predicate is similar to the first one. It takes a list of objects files and produces a binary executable.

A rule to build a program. We can now use these wrappers to build a program. Our example is built from three C source files: two independent components with associated header files, and a main program using these two components.

The following rule compiles each source file in turn, then links them:

```
main(Src1, H1, Src2, H2, Main, Name, Exe) :-
  gcc_compile(Src1, [H1], Obj1),
  gcc_compile(Src2, [H2], Obj2),
  gcc_compile(Main, [H1, H2], ObjMain),
  gcc_link([Obj1, Obj2, ObjMain], Name, Exe).
```

The rule defines a predicate `main` which takes as argument the three source files and two header files. It also takes the name of the binary and the binary itself. To build an actual binary, we just need to make an ETB query applying actual files to the rule.

Our example does not take into account the architecture on which the ETB node is running, or the version of the compiler used. This kind of information, as well as additional compilation flags, or other metadata can be tracked using the ETB.

4.2 AllSAT on top of Yices

In our second example, we build a tool that can generate all the solution satisfying a formula from a SAT solver. The idea is straightforward: ask the solver if the formula is satisfiable, if it is, ask the solver to generate a solution, add a clause blocking this solution to the formula, and iterate. Each iteration finds a new solution until they have all been found.

Assuming that we have two predicates `sat` and `unsat` that can establish whether a formula is satisfiable, and return a solution if it is, we can write derivation rules for a predicate `allsat` as follows:

```
allsat(F, Answers) :- sat(F, M),
                      negateModel(F, M, NewF),
                      allsat(NewF, T),
                      cons(M, T, Answers).
allsat(F, Answers) :- unsat(F), Answers = nil.
```

We use the additional predicate `negateModel` that extends an existing formula with a blocking clause built from a solution, as well as utility predicates `cons` and `nil` to build the list of answers and `equal` to test for equality. `cons` and `nil` are interpreted predicates that bind their last argument to new identifiers (similar to dynamic allocation that "creates" new memory addresses) to identify list nodes.

Given a solver, and its chosen representation of a formula, we can instantiate the predicates `sat`, `unsat`, and `negateModel` to get a working implementation of `allsat`. In this example, we use the `yices` SMT solver, and its native input language. We write a single interpreted predicate that calls the solver on a file containing a formula, and returns either “sat” or “unsat” as well as a model if available:

```
@Tool.predicate("+formula: file, -result: value, -model: value")
def yices(self, formula, result, model):
    ...
```

From this, we can derive the two predicates `sat` and `unsat`:

```
sat(F, M) :- yices(F, S, M), equal(S, 'sat').
unsat(F) :- yices(F, S, M), equal(S, 'unsat').
```

5 Conclusions and Future Work

We have outlined our vision for the Evidential Tool Bus, and described the design, implementation, and application of the framework. We have implemented the basic components of our proposed design, including the server engine, the client and wrapper APIs, wrappers for Yices and SAL, and several small scripts. We have also built capabilities for single-stepping the execution and observing execution traces. A lot of work remains to be done in extending the implementation, developing applications, and optimizing the implementation based on these applications. These include

1. Wrappers for a range of verification tools and static and dynamic analysis tools, through the use of standardized languages front-ends. Many of these wrappers will, we hope, be contributed by third-party developers.
2. Scripts for abstraction, invariant generation, termination, test generation, code generation, synthesis, and result certification.
3. Support for ETB proofs using forward-chaining on inference rules.

We also plan to develop a network of ETB servers operating in the cloud, and to aggressively explore novel applications of tool integration and coordination that require the construction and custody of evidence.

In summary, ETB is a semantically simple framework for formal tool integration that yields replayable evidence for use in assurance cases. It uses a variant of Datalog as the metalanguage for representing claims, queries, workflows, and arguments. Tools are integrated as uninterpreted predicates through wrappers, and made available through a client-server network. Claims are established through distributed query processing over this network, and proofs are constructed from these claims. ETB will be made available in open source form for community-wide experimentation and enhancement.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, Massachusetts, 1995.

2. Alvaro, Peter and Marczak, William and Conway, Neil and Hellerstein, Joseph M. and Maier, David and Sears, Russell C. Dedalus: Datalog in Time and Space. Technical report, EECS Department, University of California, Berkeley, Dec 2009.
3. R. E. Bloomfield, P. G. Bishop, C. C. M. Jones, and P. K. D. Froome. *Adelard Safety Case Development Manual*. Adelard, 1998.
4. Aaron R. Bradley. Understanding IC3. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy*, volume 7317 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2012.
5. Robert K. Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.
6. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
7. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
8. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming (ESOP), Edinburgh, UK*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
9. Hayco de Jong and Paul Klint. Toolbus: The next generation. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 2852 of *Lecture Notes in Computer Science*, pages 220–241. Springer, 2002.
10. Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER toolkit. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–92, Berlin, Germany, March 2000. Springer-Verlag.
11. Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.
12. Een, Niklas and Sorensson, Niklas. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.
13. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.
14. Orna Grumberg and Shmuel Katz. Veritech: a framework for translating among model description notations. *STTT*, 9(2):119–132, 2007.
15. R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. In *IEEE Symposium on Logic in Computer Science*, Ithaca, NY, 1987.
16. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.

17. David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. In David Lee, Antonia Lopes, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems (29th FORTE / 11th FMOODS'09)*, volume 5522 of *Lecture Notes in Computer Science (LNCS)*, pages 1–25. Springer-Verlag (New York), Lisboa, Portugal, June 2009.
18. Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. Remote integration and coordination of verification tools in JETI. In *ECBS*, pages 431–436. IEEE Computer Society, 2005.
19. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca., 1981.
20. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
21. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. Isabelle home page: `http://isabelle.in.tum.de/`.
22. Frank Pfenning and Carsten Schürmann. Twelf - a meta-logical framework for deductive systems (system description). In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer-Verlag, 1999.
23. Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David Scott Warren, and Juliana Freire. XSB: A system for effciently computing WFS. In *Logic Programming and Non-monotonic Reasoning*, pages 431–441, 1997.
24. John M. Rushby. An evidential tool bus. In Kung-Kiu Lau and Richard Banach, editors, *7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK*, volume 3785 of *Lecture Notes in Computer Science*, pages 36–36. Springer, 2005.
25. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.
26. Natarajan Shankar. Automated deduction for verification. *ACM Computing Surveys*, 41(4):20:1–20:56, October 2009.
27. Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.