Invited paper, to be presented at ICFEM'05, Manchester, England, November 2005.

An Evidential Tool Bus*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. Theorem provers, model checkers, static analyzers, test generators... all of these and many other kinds of formal methods tools can contribute to the analysis and development of computer systems and software. It is already quite common to use several kinds of tools in a loose combination: for example, we might use static analysis and then model checking to help find and eliminate design flaws prior to undertaking formal verification with a theorem prover. And some modern tools, such as test generators, are built using model checkers, predicate abstractors, decision procedures and constraint solvers as components in tight combination.

But we can foresee a different kind of combination where many tools and methods are used in ad hoc combination within a single analysis. For example, static analysis might yield invariants that enable decision procedures to build a predicate abstraction whose reachable states are calculated as a BDD and then concretized to yield a strong invariant for the original system; the invariant then enables properties of the original system to be verified by highly automated theorem proving.

This sort of combination clearly requires an integrating platform—a *tool bus*—to connect the various tools together; but the capabilities required go beyond those of platforms such as Eclipse. The entities exchanged among clients of the bus—proofs, counterexamples, specifications, theorems, counterexamples, abstractions—have logical content, and the overall purpose of the bus is to gather and integrate *evidence* for verification or refutation.

In this paper I propose requirements for such an "evidential tool bus," and sketch a possible architecture.

1 Introduction

Early tools for formal methods were closed and monolithic "verification systems" that typically provided a tightly integrated environment for a given specification language (and sometimes an implementation language as well) together with a theorem prover that supported interactive proof of conjectures about specifications or programs. As specialized forms of analysis and deduction became highly efficient, some of the monolithic systems began to use them as components. For example, PVS has used an external BDD package to perform Boolean

* This work was supported by SRI International.

simplification and CTL model checking since 1995 [1], and an experimental extension uses MONA as an external procedure to decide WS1S formulas [2]. The Prosper project developed similar connections from HOL to external deductive components [3], and Isabelle also has capabilities of this kind [4].

Conversely, those developing tools for formal analysis of other languages sometimes used the monolithic tools as components. For example, Adams *et al* developed a tool for the Maple computer algebra system that uses PVS as a back end [5], while the LOOP project uses PVS as a back end in verification of Java programs [6].

Recognizing that some users desired access only to a subset of their capabilities, developers of monolithic systems began to open up their interfaces, and to make some of their components available separately. These modest adjustments led to larger changes. For example, ICS [7] began as a project to make the capabilities of the decision procedures in PVS available separately. However, once the decision procedures were available in isolation, they were applied to bigger problems than were encountered within an interactive proof, and much greater performance was therefore required. At the same time, it was realized that the decision procedures could be combined with a SAT solver to yield a more useful set of services (i.e., the ability to decide arbitrary Boolean combinations of terms in the decided theories, rather than just conjunctions) [8]. Several groups independently recognized the utility of such procedures for Satisfiability Modulo Theories (SMT) and explored different techniques for their construction, which are now being honed through competition [9]. Efficient SMT solvers enabled a new class of model checkers that operate over infinite domains [10] (so-called "infinite" bounded model checkers such as the `sal-inf-bmc` tool of SAL [11]), whose capabilities were then extended from refutation to verification by k-induction [12].

The case of SMT solvers illustrates how identification of independently useful capabilities within monolithic tools can lead to enhancement of those capabilities, which can in turn spur the development of new tools. Opening interfaces can have a similar impact. For example, the underlying engines of the SAL model checkers are programmed in a variety of languages, but are packaged as foreign functions that can be called from Scheme programs. The model checkers themselves are simply scripts written in Scheme that make use of the services provided through these functions. Users can write their own Scheme scripts and this has allowed a novel "witness" model checker [13] and a test generator [14] to be developed relatively easily. Other modern model checking suites also are extensible, although in different ways, and have open interfaces [15, 16].

It is natural to ask whether the development of scriptable interfaces and of independently useful verification components can be generalized to a "plug and play" environment in which verification systems and model checkers are deconstructed into components that can then be reassembled to undertake new formal analysis tasks. By analogy with the way tools are assembled in the embedded systems and hardware design fields, and with the Eclipse programming environment, we could call this hypothesized environment a "formal tool bus."

2 Formal Tool Bus Scenarios

The simplest conception of a formal tool bus is one that provides a way to invoke deductive services such as decision procedures or model checkers for *endgame* purposes. By "endgame" we mean that the deductive service is given a proof goal and either discharges it or reports failure. The model checker of PVS is used in just this way: if the model checker succeeds in verifying the property concerned, then PVS considers the current proof goal to be discharged; if not, then it simply reports "unproved" and the user must think again. This is different than the way the more tightly integrated proof services work in PVS: these either discharge the current goal, or return a collection of simplified goals that can separately be analyzed by further proof steps. The endgame approach is attractive for powerful deductive services because they are generally designed as "all or nothing" verifiers rather than as simplifiers, and it also decouples the deductive services from the details of the proof representation (e.g., whether it is natural deduction or sequent calculus) used by the tool that invokes it.

A tool bus for endgame purposes could be quite simple to construct: it would support some neutral representation (e.g., in XML) for communicating deductive queries from the "client" tools to the "server" endgame verifiers, with suitable translators to and from the native notations of the tools concerned. In addition to syntax, the endgame verifiers would publish the semantics of the proof queries they support and the translators could be responsible for checking these and possibly performing suitable reductions (e.g., if a quantified formula is sent to a tool that requires unquantified ones, the translator could either reject it, or perhaps Skolemize it). Attaching a client or server to the bus would be a matter of providing a suitable translator. Many classes of backend verifiers have standard input formats (mostly due to competitions): for example, almost all high-performance SAT and SMT solvers, first order theorem provers, and BDD packages accept the input format defined by their respective competitions. Thus, each class of backend verifiers could be supported by a single tool bus translator. The internal tool bus of SAL exploits this: its bounded model checker can use any of several SAT solvers, and its infinite bounded model checker can use any of several SMT solvers.

A simple tool bus of this kind could be quite useful. Imagine, for example, that we wish to support some formal analysis for the Stateflow language of the Matlab toolset [17] (Stateflow provides a combination of statechart and flowchart notations). An operational [18] or denotational [19] semantics for Stateflow naturally supports construction of a translator from Stateflow to a traditional state machine representation. Some elements of Stateflow are notoriously tricky—e.g., the "12 o'clock rule," which uses the graphical representation of a chart to resolve multiple enabled transitions from a junction—and are discouraged by some style guides. The Stateflow analyzer could check whether the given chart relies on the 12 o'clock rule by submitting the conjunction of the conditions on each pair of transitions from each junction to an endgame verifier to check for unsatisfiability (the disjunction of all conditions on each junction should also be checked for satisfiability). Properties of a well-defined Stateflow chart could similarly be checked by submitting the translated state machine, and a representation of the property concerned, to a suitable model checker attached to the tool bus.

An obvious limitation of the simple tool bus is that it makes no provision for any diagnostic output produced by an endgame verifier when it fails to prove the conjecture submitted to it: e.g., the counterexample produced by a model checker, or the satisfying assignments produced by SAT or SMT solvers (these are equivalent to counterexamples when the solvers are applied to negated formulas). A crude extension could return the diagnostic information in a form that is uninterpreted by the tool bus, but that could be displayed to the human user by the client concerned.

A more serious limitation of this simple conception of a formal tool bus is that it imposes a dichotomy between clients and servers, and assumes that all servers are endgame verifiers. There are several tools that perform valuable formal analyses that use more complex relationships among their deductive subcomponents. For example, software model checkers such as Blast [20] generally operate on a finite state representation of the given program that is constructed by predicate abstraction [21]. The predicate abstraction can be constructed eagerly using an SMT solver [22] and then analyzed with a conventional finite state model checker, or it can be constructed and examined lazily ("on the fly") using a kind of explicit state model checker. If the property is not verified for the abstracted program, then the abstract counterexample can be concretized to see if provides a counterexample to the original program. If it does, we have found a bug; if not, then analysis of the counterexample (e.g., using Craig Interpolation [23]) may suggest additional predicates that can be used to refine the abstraction. This process of abstraction and refinement can be iterated until verification, refutation, or exasperation is achieved.

If we desire a tool bus that can support construction of a software model checker of the kind outlined above from more basic components, then it is clear that we need more kinds of components that just endgame verifiers, and more kinds of data then just conjectures and proof judgments. Now, we could obviously extend the simple tool bus to provide these capabilities, but the passive nature of the bus would force the components that use it to build-in a lot of knowledge about each other. For example, the component that provides the top level of the software model checker would need to know in detail how to name and invoke the component that performs the predicate abstraction, while the abstraction component would similarly need to know how to invoke the services of an SMT solver; the tool bus would merely provide the data paths between these active components and adds little value. Might it not be better for components to describe the services they provide declaratively, and to let the tool bus become an active entity that brokers interaction between components on the basis of their declarations?

An active tool bus of this kind could resemble any of several architectures for coordination services or distributed programming (e.g., *The Information Bus* [24]). However, the data carried by a *formal* tool bus, and the components that attach to it, have a special character that should be exploited: the data are formulas in a logic, and the components mechanize logical inferences. The tool bus thus ties together formulas and inferences: it is a theorem prover!

3 The Tool Bus As Theorem Prover

That a tool bus should be a theorem prover is not as surprising as it may at first seem. Verification systems and model checking environments can both be seen as rather primitive and closed tool buses; verification systems naturally have a theorem prover as their integrating component, but modern model checking systems such as Cadence SMV also have a simple "proof assistant" as their top-level integrator [25]. The tool bus as theorem prover is just an extension to these simpler manifestations.

We envisage a formal tool bus manipulating formulas in some logic (the "Formal Tool Bus Logic" FTL). The FTL needs to be rich enough that it can define the kinds of entities we want to manipulate: state machines, abstractions, counterexamples, test cases, and so on. Predicates record judgments on these entities: e.g., $M \models I$ ("state machine M is a Kripke model for temporal logic formula I"). Components are viewed as oracles that can verify judgments and construct witnesses for them. For example, a symbolic model checker may be able to verify the judgment $M \models I$ when the state machine M and temporal logic formula I are of suitable kinds (e.g., finite state, and expressed in LTL, respectively). A predicate abstractor will take a state machine M, a property I and a set of predicates P, and return a state machine $\hat{M}$ and property $\hat{I}$ such that

$$\hat{M} \models \hat{I} \Rightarrow M \models I.$$

Here, $\hat{M}$ and $\hat{I}$ are witnesses to a judgment that expresses what it means to be a property-preserving abstraction.

Components publish their capabilities to the tool bus in terms of the judgments they can (sometimes) verify and the witnesses they can (sometimes) construct. Given some input formulas and desired judgments, the tool bus performs simple forward and backward chaining (rather like a distributed logic programming system) to find a sequence of component invocations that achieve the desired goal. Possibly, the process could be guided by hints or scripts supplied by tool developers or users.

The logic FTL needs to be at least as rich as that supported by any component, and we propose the PVS logic for this purpose [26, 27]. PVS is a higher-order logic that goes beyond that supported by HOL and Isabelle/HOL in having dependent types and predicate subtypes (and structural subtypes in the latest version), and rich collections of base types and type constructors (including inductive and recursive data types). Predicate subtypes, in particular, are immensely useful as they allow concise yet perspicuous specifications, and associate judgments with formulas in a way that allows the theorem prover easily to find those that are relevant: by looking at the types of the formulas concerned.

State machines can be specified in PVS as transition relations, but tools such as model checkers generally require a more structured presentation such as a collection of guarded commands. We envisage that FTL will augment PVS with several such structured presentations—for state machines, counterexamples, abstractions, and so on. Many component tools will operate on these structured presentations, but they can always be expanded out into their base form (i.e., their underlying semantics) by components dedicated to this purpose. Thus, a state machine could be expanded into its base form as a transition relation so that a theorem prover can verify some invariant, which is then made available to an abstractor that operates on the structured state machine representation.

A large fraction of PVS can be executed very efficiently using its "ground evaluator" (which operates by translation to Lisp, using static analysis to allow destructive updates where possible) [28]. This allows proof procedures and scripts to be developed in PVS via computational reflection—for example, César Muñoz has developed an explicit state model checker Besc in PVS that is proven correct (in PVS) and operates with gratifying performance.[1] Thus, we propose to use PVS itself as the language in which components declare their capabilities, and as the scripting language for the tool bus. Scripts may be used by tool developers to program some components (as with the Besc, although we envisage the main uses to be "wrappers" and "glue logic" that adapt some externally-written component to the needs of the tool bus), and by users to construct ad-hoc analyses for specific purposes.

Although the tool bus functions as a theorem prover by performing sound and automated inferencing, we do not anticipate it becoming a powerful prover in its own right: that task can be delegated to existing theorem provers (such as PVS) that can be attached to the bus as components.

4 Evidence Management

The formal tool bus as described so far can invoke and coordinate computations by numerous components in the service of some analysis goal. For efficiency, the results of component computations should be cached and reused where possible. This requires the tool bus to employ some version management service so that changes to files storing specifications and other entities can be detected and cached analyses based on them invalidated.

The tool bus must then have a capability similar to the "proofchain analyzer" of PVS, which reports on the complete chain of deduction that supports a claimed theorem (e.g., "it uses these definitions and these lemmas, of which these ones have not been proved, and these ones have not been rechecked since their underlying files were modified").

But whereas in PVS, "proved" means "proved by PVS," the tool bus integrates many components and we need to know which ones were used in a particular analysis, and in what way: in short, we need to know the *evidence* for believing a given claim. A record of which components were used, and how, delivers rather weak evidence that rests mainly on the reputation of the components concerned. Thus, we envisage that components should be able to construct

[1] See `http://research.nianet.org/~munoz/sources.html` for Besc and PVSio, which allows evaluation to be used in proof.

"proof objects" justifying their claims. The complexity (and hence potential for error) in most deductive software is in its highly optimized method of search: once the right proof step has been found, it is usually fairly easy to check that it is performed correctly. Proof objects provide a "trail" of the proof steps applied that can be checked by a simple (and potentially verified) program.

On command, the tool bus will instruct its components to deliver their proof objects and will be able to assemble an overall proof object that provides checkable evidence for the claimed analysis. Thus, we speak of an "evidential" tool bus: one that manages evidence. In an imperfect world, we may not be able to obtain checkable proof objects, or even proofs, for some component deductions. Thus, evidence for some steps may be recorded as "because I say so," or "by testing," or "by field experience." The evidential task of the tool bus is to assemble and present the evidence available: assessing the value of that evidence is a task left to human judgment.

5 Conclusion and Prospects

We have considered requirements for a formal tool bus that integrates components that perform the elements of formal analyses. We concluded that the tool bus is best seen as a theorem prover in which the components act as oracles. Components publish their capabilities in the form of proof rules, and the tool bus invokes them as appropriate by forward and backward chaining, possibly guided by scripts. The operation of the tool bus is similar to a distributed logic programming environment: automatically, this allows components and elements of the specification under analysis to be distributed (potentially across different organizations, each responsible for some of the component tools, and some of the specification).

The logic of the tool bus should be a rich one, so that it is capable of describing the types of all the data that it manipulates and the proof rules of its components. It should also be executable, so that it can provide its own scripting language. We proposed the predicatively subtyped higher order logic of PVS for this purpose. The tool bus is integrated with version management, and manages not only the invocation of attached components and the communication of data among them, but also the collection of evidence for the outcomes of the analyses performed.

We plan to construct a prototype of the evidential tool bus to connect our PVS, SAL, and ICS tools. To be truly useful, however, a formal tool bus needs to be a collective endeavor supported by all potential component developers and by all who would use it to perform analyses. We hope that others will give this proposal their consideration and will revise or replace it with one that best serves the needs of the verification community.

Observe that the proposed evidential tool bus operates at a deliberately coarse level of granularity. Different considerations apply inside a tightly integrated component such as an SMT solver, where hundreds of millions, rather than thousands of deductions may be performed in the course of a single analysis [29]. At the other extreme, it is interesting to speculate whether an evidential tool bus can serve, or can be extended to serve, in support of *system certification* (e.g., in support of a safety case) rather than mere analysis.

Acknowledgments

This proposal for an evidential tool bus was developed through discussions with my colleagues Leonardo de Moura, Sam Owre, N. Shankar, and Ashish Tiwari.

References

1. Rajan, S., Shankar, N., Srivas, M.: An integration of model-checking with automated proof checking. In Wolper, P., ed.: Computer-Aided Verification, CAV '95. Volume 939 of Lecture Notes in Computer Science., Liege, Belgium, Springer-Verlag (1995) 84–97
2. Owre, S., Rueß, H.: Integrating WS1S with PVS. In Emerson, E.A., Sistla, A.P., eds.: Computer-Aided Verification, CAV '2000. Volume 1855 of Lecture Notes in Computer Science., Chicago, IL, Springer-Verlag (2000) 548–551
3. Dennis, L.A., Collins, G., Norrish, M., Boulton, R., Slind, K., Robinson, G., Gordon, M., Melham, T.: The PROSPER toolkit. In Graf, S., Schwartzbach, M., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000). Volume 1785 of Lecture Notes in Computer Science., Berlin, Germany, Springer-Verlag (2000) 78–92
4. Tverdyshev, S.: Combination of Isabelle/HOL with automatic tools. In Gramlich, B., ed.: Frontiers of Combining Systems: 5th International Workshop, FroCoS 2005. Volume 3717 of Lecture Notes in Computer Science., Vienna, Austria, Springer-Verlag (2005) 302–309
5. Adams, A., Dunstan, M., Gottliebsen, H., Kelsey, T., Martin, U., Owre, S.: Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Boulton, R.J., Jackson, P.B., eds.: Theorem Proving in Higher Order Logics, TPHOLs 2001. Volume 2152 of Lecture Notes in Computer Science., Edinburgh, Scotland, Springer-Verlag (2001) 27–42
6. Breunesse, C.B., Cataño, N., Huisman, M., Jacobs, B.P.F.: Formal methods for smart cards: An experience report. Science of Computer Programming **55** (2005) 53–80
7. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N.: The ICS decision procedures for embedded deduction. In Basin, D., Rusinowitch, M., eds.: 2nd International Joint Conference on Automated Reasoning (IJCAR). Volume 3097 of Lecture Notes in Computer Science., Cork, Ireland, Springer-Verlag (2004) 218–222
8. de Moura, L., Rueß, H.: Lemmas on demand for satisfiability solvers. In: Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02), Cincinnati, OH (2002) Available at `http://www.csl.sri.com/users/demoura/sat02_journal.pdf`.
9. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability modulo theories competition. [30] 20–23
10. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In Voronkov, A., ed.: 18th International Conference on Automated Deduction (CADE). Volume 2392 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 438–455

11. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In Alur, R., Peled, D., eds.: Computer-Aided Verification, CAV ’2004. Volume 3114 of Lecture Notes in Computer Science., Boston, MA, Springer-Verlag (2004) 496–500
12. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In Hunt, Jr., W.A., Somenzi, F., eds.: Computer-Aided Verification, CAV ’2003. Volume 2725 of Lecture Notes in Computer Science., Boulder, CO, Springer-Verlag (2003) 14–26
13. Shankar, N., Sorea, M.: Counterexample-driven model checking. Technical Report SRI-CSL-03-04, Computer Science Laboratory, SRI International, Menlo Park, CA (2004) Available at http://www.csl.sri.com/users/sorea/reports/wmc.ps.gz.
14. Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: 2nd International Conference on Software Engineering and Formal Methods, Beijing, China, IEEE Computer Society (2004) 261–270
15. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. [31] 495–499
16. Dwyer, M.B., Hatcliff, J., Hoosier, M., Robby: Building your own software model checker using the Bogor extensible model checking framework. [30] 148–152
17. The Mathworks: Stateflow and Stateflow Coder, User’s Guide. Release 13sp1 edn. (2003) Available at http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf.
18. Hamon, G., Rushby, J.: An operational semantics for Stateflow. In Wermelinger, M., Margaria-Steffen, T., eds.: Fundamental Approaches to Software Engineering: 7th International Conference (FASE). Volume 2984 of Lecture Notes in Computer Science., Barcelona, Spain, Springer-Verlag (2004) 229–243
19. Hamon, G.: A denotational semantics for Stateflow. In: EMSOFT 2005: Proceedings of the Fifth ACM Workshop on Embedded Software, Jersey City, NJ, Association for Computing Machinery (2005) 164–172
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with Blast. In: Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN). Volume 2648 of Lecture Notes in Computer Science., Springer-Verlag (2003) 235–239
21. Saïdi, H., Graf, S.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Computer-Aided Verification, CAV ’97. Volume 1254 of Lecture Notes in Computer Science., Haifa, Israel, Springer-Verlag (1997) 72–83
22. Saïdi, H., Shankar, N.: Abstract and model check while you prove. [31] 443–454
23. McMillan, K.L.: An interpolating theorem prover. In Jensen, K., Podelski, A., eds.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’04). Volume 2988 of Lecture Notes in Computer Science., Barcelona, Spain, Springer-Verlag (2004) 16–30
24. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus—an architecture for extensible distributed systems. In: Fourteenth ACM Symposium on Operating System Principles, Asheville, NC (1993) 58–68 (ACM Operating Systems Review, Vol. 27, No. 5).
25. McMillan, K.L.: Verification of an implementation of Tomasulo’s algorithm by compositional model checking. In Hu, A.J., Vardi, M.Y., eds.: Computer-Aided Verification, CAV ’98. Volume 1427 of Lecture Notes in Computer Science., Vancouver, Canada, Springer-Verlag (1998) 110–121
26. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering **21** (1995) 107–125

27. Owre, S., Shankar, N.: The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA (1997)
28. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA (2001) Available from http://www.csl.sri.com/users/rushby/abstracts/attachments.
29. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N.: Integrating verification components. In: Verified Software: Theories, Tools, Experiments (IFIP Working Conference), Zurich, Switzerland (2005) To appear.
30. Etessami, K., Rajamani, S.K., eds.: Computer-Aided Verification, CAV '2005. Volume 3576 of Lecture Notes in Computer Science., Edinburgh, Scotland, Springer-Verlag (2005)
31. Halbwachs, N., Peled, D., eds.: Computer-Aided Verification, CAV '99. Volume 1633 of Lecture Notes in Computer Science., Trento, Italy, Springer-Verlag (1999)