Position paper for VSTTE, Zurich, Switzerland, October 2005.

Integrating Verification Components*

Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. A number of impressive verification tools and techniques have been developed over the last few years. These tools have proved successful in verifying limited classes of properties or small-scale systems. These verification methods include test case generation, static analysis, type checking, model checking, decision procedures, and interactive theorem provers. Effective large-scale verification requires the careful integration of these verification tools so that deeper properties of large systems emerge from the cooperation of a suite of tools. We outline some of the challenges in achieving a coherent integration of verification components both at fine-grain and coarse-grain levels.

Computer-aided verification through the use of model checkers and theorem provers has become a critical technology in the design of reliable systems, and the efforts of researchers over the past 50 years has yielded an impressive array of verification tools. However, no single tool or technique is going to solve the verification problem. Instead, an entire spectrum of formal methods and tools are needed ranging from test case generators, runtime verifiers, static analyzers, and type checkers, to invariant generators, decision procedures, bounded model checkers, explicit and symbolic model checkers, and program verifiers. These tools and techniques are used to calculate properties of designs and implementations to varying degrees of assurance. They are also interdependent so that a useful verification system typically combines several of these techniques.

There are many reasons why tool integration has become critical in computer-aided verification. The individual tools have become quite sophisticated and specialized and their development and maintenance requires a substantial investment of time and effort. Few research groups have the resources to afford the development of custom tools. The range of applications of verification technology has been broadened to include a wide array of analyses such as test case generation, extended static checking, runtime verification, invariant generation,

* Funded by NSF Grant Nos. CCR-ITR-0326540 and CCR-ITR-0325808, DARPA REAL project, and SRI International.

controller synthesis, and proof-carrying code, to name a few recent developments. Several of these applications make opportunistic use of available tools to achieve partial but effective analyses that uncover a large class of bugs. Specialized tools also need to be integrated to deal with domain-specific nature of the verifications tasks, where the domains may range from hardware and systems code to embedded real-time and hybrid systems and cryptographic protocols.

Predicate abstraction [SG97] is a good example of the integration of various verification tools. Predicates over the concrete state space are used to construct a finite-state approximation of the transitions and properties through the use of theorem proving. Model checking is then used to explore the abstract state space. If an abstract counterexample is found, satisfiability checkers to construct a corresponding concrete counterexample. If there is no corresponding concrete counterexample, techniques like interpolation [JM05] can be applied to proof of unsatisfiability to refine the abstraction predicates so as to exclude spurious abstract counterexamples.

We therefore argue that a program verifier as envisioned in the Verification Grand Challenge should consist of interconnected specialized analysis and verification components for various logics and logic fragments. The challenge here is to design a verification architecture that supports coherent integration between inference components for fine-grained and coarse-grained interaction. Cooperating decision procedures in the style of the Nelson–Oppen method [NO79] is an example of fine-grained interaction. The combination of propositional satisfiability and ground decision procedures can be carried out through fine-grained interaction as in the lazy approach [dMRS02] where a satisfiability solver is modified to produce assertions and queries for a decision procedure. Such a combination can also be realized through a coarse-grained interaction as in the eager approach of using a decision procedure to generate lemmas that assist a satisfiability solver interaction. PVS [ORS92] supports both a fine-grained interaction with a decision procedure and rewriter as well as the coarse-grained integration of various inference procedures including a model checker.

This position paper reports on the theoretical and practical challenges of building component tools as well as integrating components into a larger system. The practical challenges are mainly in managing the trade-off between efficiency and modularity, whereas the theoretical challenges are in achieving cohesive fine-grained and coarse-grained interaction between specialized components. We present two related challenges for a component technology for verification. The first challenge addresses the interfaces that these components must support for ease of integration. The second challenge focuses on the architectural frameworks for coarse-grained and fine-grained integration of verification components.

Challenge 1:

> **Design of interfaces for dedicated verification components that allow flexible use without loss of efficiency.**

Verification tools can either be employed *directly* or *embedded* within other analysis tools such as type checkers, compilers, test case generators, and program synthesizers. Whereas the interface of a verification engine for direct use is straightforward, embedded deduction require component interfaces that are

1. *Online*: Allow incremental processing of assertions and queries.
2. *Resettable*: Support the saving of contexts as well as backtracking to a prior context with fewer assertions, and switching between contexts.
3. *Queriable.* Simplify expressions with respect to a context.
4. *Evidential.* Generate proof objects, unsatisfiable cores, and models.
5. *Tunable.* Provide prompt or any-time response that delivers useful partial results in the face of resource constraints.
6. *Integrable.* Support fine-grained integration with other tools, possibly implemented in a different language, with seamless memory management and error signalling.

For example, the *lazy integration* of a SAT solver with a theory-specific constraint solver requires the SAT solver to provide interfaces for elaborating the splitting heuristic and for resuming the search with additional clauses. Likewise, the constraint solver must contain operations for incremental and resettable processing, maintaining multiple contexts, and for returning compact conflict sets. These requirements are not orthogonal. For example, proof production assists in identifying unsatisfiable cores which in turn yields efficient backtracking in the search procedure for satisfiability. Similarly, an efficient querying capability can be exploited within the propagation steps of the satisfiability procedure.

This kind of flexibility, however, can impair efficiency in applications where these features are irrelevant. With a narrow API, less of the implementation is exposed leaving many more choices for the internal design of the component. It is often easier to engineer and implement non-modular interaction without the overhead. Indeed, Lampson [Lam] argues that only a small number of components, typically those like data bases and compilers, actually see much reuse, and Boyer and Moore argue that it is often easier to implement decision procedures that are customized for a specific purpose than to adapt off-the-shelf components [BM86].

These challenges confront both the implementors and the integrators of components, and they are by no means unique to software. However, the problems are compounded by the fact that software offers manifold modes of interaction. Though modularity poses serious challenges, we have already noted that there are compelling reasons for pursuing it in the context of verification software.

Challenge 2:

> **Design of an integration architecture that supports semantic interaction between inference components.**

Effective integration requires careful engineering of the components as well as the integration frameworks. For this purpose, we have to distinguish between coarse-gained and fine-grained interaction between components.

- *Coarse-grained interaction* can be between homogeneous components which share the same pattern of usage such as tactics, or between heterogeneous components such as model checkers and decision procedures. Components themselves can be developed as libraries, or for online or offline use. Online components process inputs incrementally and therefore employ algorithms that are different from those in an offline component. In such an integration where components do not interfere with each other, the framework can impose discipline on the interaction.
- *Fine-grained interaction* requires shared representations and shared state between components and is typical of combination decision procedures over a union of theories.

Many theorem proving systems are based on specific integration architectures such as tactics-based integration [GMM+77] or Nelson-Oppen [NO79] combination. Because of the specific demands of these integration architecture, the components are usually designed specifically for their use within these systems. Modern verification components, however, are extremely sophisticated and their implementation and maintenance require a substantial investment of time and energy. This effort would be squandered if we cannot find effective ways of reusing the components.

One particular challenge is to map between different logics of existing verification components [Mes89]. This does not only include the mapping of formulas and theorems across theories, but also certificates including proofs or models. We argue that the theoretical design of the composition framework is key to achieving flexible and efficient integrated tool suites. This need is not peculiar to verification tools, since composition is the primary challenge in any complex design. In the case of integrated verification tools, formal composition frameworks are needed that provide architectures and interfaces for communicating models, properties, counterexamples, and proofs. We first discuss the challenge of achieving coarse-grained integration, and then examine the case of fine-grained integration.

A Tool Bus for Loosely Coupled Integration. A formal framework for the loose coupling of heterogeneous components must provide

1. A read-eval-print loop for interacting with different components.
2. A scripting language for building analysis tools combining the existing components.
3. An interface for adding new components.
4. A mechanism for building evidence justifying the results of the analyses obtained by chaining together the evidence generated by the individual components.

5. An incremental, and possibly distributed, development manager for recording and rerunning development scripts in the face of changes.
6. A logical query mechanism for the database of developments and judgments.

We call this framework an *evidential tool bus*. Unlike most previous attempts for building tool integration frameworks [DCN$^+$00], the proposed verification tool bus focuses on the conceptual level rather than the operational details of tool invocation.

The basic primitive in the verification tool bus is an assertion of the form $T \vdash P : J$ which denotes the claim that tool T provides a proof P for judgment J. The proof P here need not be a mathematical proof but merely the supporting evidence for a claim. In the integration, tools can communicate in terms of labels for structures, where the content of these labels is internal to a specific tool in a manner similar to variable abstraction in combination decision procedures. For example, the BDD package exports labels for BDDs without exposing their actual structure. The specific judgment forms can be syntactic as well as semantic. Typical judgments include

1. A is a well-formed formula.
2. A is a well-typed formula in context τ.
3. a is a BDD representing the formula A.
4. $\mathcal{C}$ is a decision procedure context representing the input Γ.
5. A is satisfiable in theory $\mathcal{T}$.
6. Γ is a satisfying assignment for A.
7. Γ is a minimal unsatisfiable set of literals.

Each component builds such judgments by forward chaining from existing judgments or backward chaining through the generation of proof obligations. For example, a type checker can establish a judgment of type correctness relative to a set of proof obligations that may be discharged either by a decision procedure or an interactive theorem prover, that may generate additional type checking queries. Static analysis can be used to infer simple program properties. Verification condition generators can also establish program properties relative to the generated verification conditions. Slicing and abstraction can be used to generate reduced programs that preserve certain classes of properties. Abstract reachability can be used to combine theorem proving and model checking for establishing nontrivial program properties. Test case generation can be used as an inexpensive method for finding bugs in both programs and their specifications. These basic analysis techniques can be incorporated into scripts that support automated ways of decomposing verification tasks and assembling analysis results. The tool bus thus serves as a uniform framework for interacting with existing components, adding new components, defining scripts, translating between different logics, coordinating garbage collection, and managing the evidence generated.

Coarse-grained integration is necessary for building powerful verification tools out of specialized analysis components. The main challenges here are in managing the translations between the logics and formats employed by the different systems and managing the evidence produced by each component.

Formal Architectures for Fine-Grained Integration

The integration frameworks described in the previous section dealt with the loose coupling of large inference components such as those used in proof construction or model checking. In such an integration where components do not interfere with each other, the framework can impose discipline on the interaction. Tight coupling therefore poses theoretical challenges that are not present in the loosely coupled case. In a tightly coupled setting, the components interact through a shared state. The interaction has to be mediated through a well-defined architecture to avoid unintended interference. Such an architecture for composing components must allow component properties to be established independent of the other components, and system properties to be derived from those of the components. For the case of combination decision procedures, we have developed a formal architecture that provides a theoretical framework for composing decision procedures for specific theories to obtain a combination decision procedure for the union of these theories [RS01, GRS04]. This framework is based on the concepts of inference systems and inference modules and a theory of compositionality and refinement for inference systems.

Inference systems offer a scheme for defining sound and complete decision procedures for a specific theory. Inference components capture open decision procedures that can interact with similar components for other theories. An inference component is an inference structure where each configuration κ consists of a shared part (a *blackboard*) γ and a local, theory-specific part (a *notebook*) θ. The shared part γ contains the input constraints G in the union theory $\mathcal{T}_\infty$ as well as the shared constraints V in the intersection theory $\mathcal{T}_0$. The semantic constraints on the inference relation of an inference module are slightly stronger than those of an inference system since the former must interact with other inference components by means of inputs and outputs through the shared blackboard.

The composition $M_1 \otimes M_2$ of two inference modules M_1 and M_2 is defined to yield an inference module with configurations of the form $\gamma; \theta_1, \theta_2$, where γ, θ_i is a configuration in module M_i for $i \in \{1, 2\}$. The inference relation for $M_1 \otimes M_2$ is the union of those for the component modules and is applied to the relevant part of the logical state. Two inference modules are compatible if they can be shown to be jointly progressive on the shared part. The composition of two compatible inference modules can be shown to be an inference module for the union of the respective theories, provided these theories satisfy certain conditions. A generalized component can be defined to capture the abstract behavior of typical inference modules.

Inference modules can be shown to yield a modular presentation of known combination results such as those of Nelson and Oppen [NO79], Shostak [Sho84, Gan02, SR02], and Ghilardi [Ghi03]. These systems are, in a formal sense, refinements of generalized components. In practical terms, inference modules provide a software architecture for combination decision procedures. The architecture of the ICS decision procedures [dMOR+04] is based on inference modules.

The theory of open inference systems and inference modules is a small step in the direction of a software architecture framework for tightly-coupled ground decision procedures. Much more work is needed to handle the integration of richer theories with overlapping signatures and quantification. A recent line of work for generating combination decision procedures from general-purpose proof engines has shown promising results [ABRS05].

Conclusions. Lampson [Lam] in his skepticism about component-based software development, and Boyer and Moore [BM86] in their critique of black box decision procedures, correctly identify the many obstacles to the smooth integration of pre-existing components. Integration does pose significant challenges in theory as well as practice. The technology involved in the construction of inference components has become extremely sophisticated so that we have little choice but to reuse existing software in the form of libraries as well as online and offline components. Components then have to be explicitly engineered for such embedded use through design and interface choices that provide flexibility without significantly compromising efficiency. In the last ten years, several such components have been made available in the form of BDD packages, model checking tools, and decision procedures, and these packages have been integrated within larger systems. Though there is no consensus on the standardized interfaces for such packages, there is a growing empirical understanding of the tradeoffs between flexibility and efficiency. Integration frameworks for loosely coupled components have been built around a shared description language. There is a now an active body of research focused on architectures for tightly coupled integration. The theoretical challenges that are being addressed by ongoing research include novel algorithms for online use, and formal architectures for composing inference components that yield systems that are correct by construction. Finally, we have presented a proposal for a verification tool bus architecture that combines various analysis tools within a framework for constructing reproducible evidence.

References

[ABRS05] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. Big proof engines as little proof engines: New results on rewrite-based satisfiability procedures (extended abstract). In Alessandro Armando and Alessandro Cimatti, editors, *Proceedings of PDPAR'05*, 2005.

[BM86] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[DCN+00] 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.

[dMOR+04] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In David Basin and Michaël Rusinowitch, editors, *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.

[dMRS02] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002. Springer-Verlag.

[Gan02] Harald Ganzinger. Shostak light. In A. Voronkov, editor, *Proceedings of CADE-19*, pages 332–346, Berlin, Germany, 2002. Springer-Verlag.

[Ghi03] Silvio Ghilardi. Reasoners' cooperation and quantifier elimination. Technical report, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 2003.

[GMM+77] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. Technical Report CSR-16-77, Department of Computer Science, University of Edinburgh, 1977.

[GRS04] H. Ganzinger, H. Rueß, and N. Shankar. Modularity and refinement in inference systems. Technical Report CSL-SRI-04-02, SRI International, Computer Science Laboratory, 333 Ravenswood Ave, Menlo Park, CA, 94025, January 2004. Revised, August 2004.

[JM05] Ranjit Jhala and K.L. McMillan. Interpolant-based transition relation approximation. In *Proceedings of the 17th International Conference on Computer Aided Verification: CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 39–51, 2005.

[Lam] Butler W. Lampson. How software components grew up and conquered the world.

[Mes89] J. Meseguer. General logics. In *Logic Colloquium '87*, pages 275–329, Amsterdam, 1989. North Holland.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.

[SG97] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SR02] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, July 2002. Springer-Verlag.