Integrating Verification Components: The Interface is the Message*

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. The efforts of researchers over the past 20 years has yielded an impressive array of verification tools. However, no single tool or method is going to solve the verification problem. An entire spectrum of formal methods and tools are needed ranging from test case generators, 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.

Much of our work at SRI over the past 15 years has been focused on tool integration both in terms of building verification tools that have been integrated in other systems, and in integrating tools that others have built into our own systems. This 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. As a step toward addressing these challenges, we present some principles for designing components and integration frameworks focusing on the interfaces between the two.

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

The black box nature of the decision procedure is frequently destroyed by the need to integrate it...
When sufficiently powerful theorem provers are finally produced they will undoubtedly contain many integrated decision procedures...
The development of useful decision procedures for program verification must take into consideration the problems of connecting those procedures to more powerful theorem provers. Boyer and Moore [BM86]

1 Introduction

Computer-aided verification through the use of model checkers and theorem provers has become a critical technology in the design of reliable systems. Verification tools can either be employed *directly* or *embedded* within other analysis tools such as type checkers, compilers, test case generators, and program synthesizers. In direct or embedded use, there is a choice between integrating components where each component provides a related set of capabilities or in implementing the functionality as needed for the application at hand. At SRI International, we have been engaged in building analysis tools out of components such as constraint solvers, decision procedures, model checkers, and type checkers with systems such as PVS, ICS, and SAL. We report on our experience in handling the challenges for modularity both at the practical and theoretical level. We argue based on this experience that it is inevitable that sophisticated verification tools will be based on powerful components, but that the design of useful components and flexible integration frameworks can pose difficult engineering challenges. These challenges can now be confronted in a systematic way given the availability of a substantial collection of components and a fair bit of experience with integration frameworks and novel applications that involve tool combinations.

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 type checking, runtime verification, invariant generation, 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. These domains require specialized automation. As an example, the verification of aircraft collision detection and resolution algorithms employ a considerable body of real analysis and trigonometry knowledge, and lack of automation can be a real barrier to effective verification.

The verification tools that we have built include PVS, ICS, and SAL [For03]. Of these, ICS [dMOR$^+$04a] is a ground decision procedure component, whereas PVS [ORSvH95] and SAL [dMOR$^+$04c] are frameworks that integrate a number of such components. PVS also serves as a back-end component in verification systems such as InVest [BLO98], TAME [Arc00], LOOP [vdBJ01], and PVS-Maple [ADG$^+$01]. We describe these integrations to motivate a discussion of the technical challenges in designing components as well as composition frameworks. The practical challenge has been in designing component interfaces that make all of the functionality available without the burden of a significant overhead. The design of component interfaces is an extremely delicate task. There are few general principles, but we offer a number of paradigmatic examples to illustrate the perils and pitfalls. 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. Composition is the primary challenge in any complex design. In the case of integrated verification tools, we motivate the need for formal composition frameworks that provide architectures and interfaces for communicating models, properties, counterexamples, and proofs.

Integration is not a new challenge for deduction. Many theorem proving systems from the 1970s were already based on such integration. The architecture of the Boyer–Moore theorem prover [BM79] was built around components for simplification, rewriting, induction, elimination, cross-fertilization, linear arithmetic, and generalization. The Stanford Pascal Verifier [LGvH$^+$79] used a combination decision procedure based on the Nelson–Oppen method [NO79]. The STP [SSMS82] and EHDM [vHCL$^+$88,RvHO91] verification systems at SRI employed a number of theorem proving techniques centered around the Shostak combination method [Sho84]. The LCF family of systems [GMW79] were engineered to be extensible with theorem proving tactics defined in a metalanguage ML [GMM$^+$77]. However, in each of these cases, the components were designed specifically for their use within these systems. Modern verification components are extremely sophisticated and their implementation and maintenance requires a substantial investment of time and energy. This effort would be squandered if we cannot find effective ways of reusing the 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. The latter requires shared representations and shared state between components and is typical of combination decision procedures over a union of theories. 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.

We first present the arguments for and against modularity in the architecture of verification tools in Section 2. A few useful components are described in Section 3 along with examples of their integration and use. Section 4 presents the way PVS, ICS, and SAL operate as integration frameworks. We then describe formal architectures for integrating formal verification components at a fine-grained level of interaction in Section 5, and a *tool bus* for a coarse-grained integration of larger components in Section 6. The conclusions are presented in Section 7.

2 The Modularity Challenge

A 1986 paper by Boyer and Moore [BM86] outlines many of the key obstacles to the integration of black box decision procedures within a theorem prover. Their paper was based on work undertaken in the late 1970s to integrate a decision procedure for linear arithmetic into their induction theorem prover. Their case against black box decision procedures can be summarized as follows.

1. The component decision procedure may employ a language for communicating formulas that is incompatible with that of the larger system. In their case, the decision procedure assumes that all variables range over numbers, whereas in the theorem prover, variables range over objects other than numbers.
2. The communication overhead with an external component can degrade performance. For example, if a large volume of information has to be translated to and from the component, then it might be better to build custom decision procedures that can operate within the same language and context as the theorem prover.
3. Term information from the decision procedure is needed to activate lemmas. Suitable instances of simple arithmetic lemmas such as $i \leq max(i, j)$ must be provided to the decision procedures. The instantiation of the lemmas of course depends on the set of terms that are known to the decision procedure. Unless the decision procedure is designed to anticipate such use, it will not be possible to provide this information without modifying the component.
4. Proof and dependency information must be generated from the decision procedures. For many uses of decision procedures it is not sufficient to merely provide decisions on the validity or refutability of a given conjecture. The procedure must also yield information on the exact proof or of the formulas involved in the proof. The need for such specialized information again means that a standalone component might not entirely serve the purpose at hand.
5. The decision procedure typically does not support any mechanism for dynamically adding and retracting assertions from a context. This level of flexibility is needed when using a decision procedure to simplify an expression in the context of other assertions.

For these and other reasons, Boyer and Moore argued that it was often easier to implement decision procedures that are customized for a specific purpose than to adapt off-the-shelf components.

In a more general setting but for very similar reasons, Lampson [Lam] argues that only a small number of components, typically those like data bases and compilers, actually see much reuse. These components provide complex functionality through a narrow interface. These interfaces are fairly standardized and do not need to be adapted based on the usage. Also, it would take an unreasonable amount of effort to implement the same functionality for each specific application. The arguments against reusable components can be summarized as follows:

1. Engineering to a flexible application programmer interface (API) is quite difficult. With a narrow API, less of the implementation is exposed leaving many more choices for the internal design of the component.
2. Ideas and algorithms are more portable than implementations. A specific implementation of a component is optimized for a usage profile and unless the application fits this profile, it might be easier to build a version of the component reusing the high-level algorithms.
3. It is often easier to engineer and implement non-modular interaction without the overhead . There is often a degree of indirection when using a component and the overhead of keeping track of the state of the component and of translating information back and forth can be formidable.
4. For any specific application, the functionality available may be too much and/or too little. If a component is constructed to be too flexible, then a large part of its functionality might be irrelevant for the application at hand. This excess functionality can still take a toll on performance. On the other hand, if some functionality that is essential for the application is not available in the component, then there is no workaround, and even a largely useful component has to be discarded in favor of a custom implementation.
5. Enriching the interface often rules out certain optimizations. For example, a decision procedure that allows assertions to be retracted will have to either avoid destructive updates or employ invertible operations. Many optimizations are applicable under a *closed world* assumption that existing concepts will not be redefined or elaborated, but this drastically restricts the interface.

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. In the following sections, we describe several approaches for overcoming the obstacles to integration at several levels, focusing on the interfaces between components and the integration frameworks.

3 Deductive Software Components

A software component is a self-contained body of code whose functionality is accessed through a well-defined interface. A piece of software is packaged into a component so that it can be used in a number of different contexts. We first present a few examples of software components that are typically integrated into modern verification systems. These examples illustrate the challenges that arise in the design of the components and their interfaces.

There are three basic classes of software components: *libraries*, *offline procedures*, and *online procedures*. These classes are not necessarily disjoint and a single component can belong to all three categories. A library is essentially a set of related procedures sharing a common set of data structures and representations. Packages which implement the binary decision diagram (BDD) representation of Boolean functions, such as CUDD [Som98], are typical examples of libraries. The MONA decision procedures [EKM98] for the theory WS1S also provides a library of automata-theoretic operations building on BDDs. Term manipulation libraries for defining syntactic and semantic operations are also available as for example in the ASF+SDF environment [vdBvDH+01]. Libraries are the most widely used form of software components and yet their developers receive little acknowledgment for their efforts.

Offline procedures provide a specific prepackaged functionality in the form a black box. Such procedures maintain no state between successive invocations. Typical examples of offline procedures are parsers, black-box decision procedures like SAT solvers and model checkers, and resolution theorem provers. These procedures are relatively easy to implement since few of the design decisions are externally visible. However, they are also the least adaptable as software components. The interfaces offered by offline procedures are too rigid for applications that require greater access to and control over the internal representations and behavior of the software.

Online procedures can be used as coroutines that maintain their own state which can be queried and updated through a family of useful operations. Examples of online procedures are incremental decision procedures and constraint solvers that offer an ask/tell interface for adding constraints to a context as well as querying the context for implied constraints. These procedures are often the most complex in terms of the individual algorithms and the overall architecture. The ability to interact with the state exposes many of the design choices and the procedure has to work correctly with respect to all possible input sequences. Online procedures are best suited for embedded applications since they offer flexible interfaces that can be customized as needed. Online procedures often come with scripting languages that can be used to extend the functionality by building on the available primitives.

3.1 BDD Libraries

Binary decision trees are representations of Boolean functions in the form of conditional expressions where the condition is a Boolean variable [BRB90]. Binary decision diagrams represent the decision trees as graphs where common subtrees are merged. Ordered binary decision diagrams impose an ordering on the decision variables that determines the decision level of the variable. The size of the BDD for a given Boolean function is quite sensitive to the choice of variable ordering. Reduced ordered binary decision diagrams (ROBDDs) also remove irrelevant choices. BDDs were originally introduced as representations for combinational logic but have since been employed in a wide range of applications. The ROBDD representation admits efficient implementations of a number of useful operations on Boolean functions such as negation, the application of binary Boolean connectives, the restriction of variables to Boolean constants, the simplification of a BDD with known constraints on the variables, and the composition of one BDD with another through substitution.

BDDs can compactly represent sets of bit-vectors. Since the states in a finite-state machine can be encoded using bit-vectors, BDDs can be used for representing finite sets of states as well as the transition relation. A typical BDD library provides a number of functions for creating BDD, garbage collection, taking the image of a BDD with respect to a transition relation, and computing fixpoints. There has been some standardization of the APIs for BDD packages so that they can, to some extent, be used interchangeably.

3.2 Offline SAT Solvers

A SAT solver determines if a given Boolean constraint is satisfiable. Typically, the Boolean constraint is presented as a collection of clauses which are disjunctions formed from Boolean variables and their negations. BDDs can also be used to determine satisfiability, but a BDD representation of a Boolean constraint displays all satisfying assignments, whereas a SAT solver merely looks for a single such assignment. Modern SAT solvers are based on the Davis–Logemann–Loveland simplification [DLL62] of the Davis–Putnam procedure [DP60]. This algorithm can be presented as an inference system as shown in Figure 1.

A configuration consists of a set of clauses Γ. The **Split** rule branches on a configuration by adding a unit clause corresponding to a Boolean variable p, while adding its negation to the second branch. The **Unit** rule propagates the consequence of a unit clause through the other clauses in a configuration. If a configuration contains a Boolean variable and its negation as unit clauses, then a contradiction is reported along that branch by the **Contrad** rule. When there are no more variables available for branching or unit clauses left to propagate, the clause set is irreducible and corresponds to a satisfying assignment for the original set of clauses.

Split	$\dfrac{\Gamma}{\Gamma, p \mid \Gamma, \neg p}$ p and $\neg p$ are not in Γ
Unit	$\dfrac{\kappa, C \vee \bar{l}, l}{\kappa, C, l}$
Contrad	$\dfrac{\kappa, \bar{l}, l}{\perp}$

Fig. 1. A Propositional Satisfiability Solver

The efficient implementation of these rules requires good heuristics for picking a branching literal, an efficient mechanism for unit propagation, and mechanisms for backtracking that avoid redundant branches. Additionally, new clauses have to be generated from detected conflicts to direct the search along more fruitful branches. In the past, SAT solving has been a speed sport where performance was the only criterion for success. As SAT solvers are being deployed in real applications, some of the interface issues have come to the fore. For example, it is not enough for a SAT solver to merely detect satisfiability. It should provide a satisfying assignment when one exists, or even all satisfying assignments, and a proof of unsatisfiability when there is no such assignment. It should be possible for an application to query the SAT solver for further satisfying assignments possibly with additional clause constraints. A SAT solver, like any other program, should be customizable with different choices of heuristics and decision parameters. SAT solvers built for embedded use already provide such interfaces.

3.3 Online Ground Decision Procedures

An online procedure is one that processes its input incrementally. Since all of the input is not initially available, this rules out solutions that involve several passes over the given input. Ground decision procedures (GDP) determine if a conjunctions of literals is satisfiable in a given theory. For example, the conjunction of the literals $f(x) = f(f(f(x))), f(f(f(f(f(x))))) \neq f(x)$ is unsatisfiable in the theory of equality over uninterpreted terms. The conjunction of the real arithmetic constraints $x < y, y < z, z < x$ is also unsatisfiable. Many applications require online implementations of GDPs that process assertions of new literals and queries with respect to a context. In the case of an assertion, the procedure either reports a contradiction, returns a new context with the asserted information, or it returns the same context when the given literal is already valid in the old context.

A simple online Gaussian elimination algorithm can process linear arithmetic equality and disequality constraints. There is a diverse collection of techniques for processing linear arithmetic inequality constraints in an incremental manner ranging from Fourier's variable elimination method to the simplex algorithm for linear programming. Online congruence closure procedures process equalities between terms into a union-find structure for equivalence classes. There are a number of theories that admit online ground decision procedures such as arrays, bit-vectors, recursive datatypes such as lists and trees, and linear integer equalities and inequalities.

Shostak [Sho84] gives a schematic presentation of online decision procedures for solvable and canonizable theories which admit a solving algorithm and a computable canonical form for each term. The basic idea underlying Shostak's scheme is that the input constraints are processed into a solution set that maps variables to terms. For each input constraint, the current solution set is first substituted into the formula and the canonizer is applied to it. The resulting formula is solved using the solver, and the solution thus obtained is composed into the existing solution set.

Many constraints involve formulas that contain symbols from a combination of theories. In solving such constraints, the challenge is to combine the decision procedures for the individual theories into one for the combination. Nelson and Oppen [NO79] give a combination method that partitions each mixed input constraint into a set of pure constraints, i.e., those whose symbols are all from a single theory. This is done by introducing fresh variables to label pure subterms. The individual decision procedures then exchange equality information regarding the shared variables until one of the theories identifies a contradiction or no further variable equalities can be derived. Decision procedures for Shostak theories can also be similarly combined by exploiting the solvers and canonizers for the individual theories [RS01,SR02].

Online decision procedures need to be efficient enough to also be usable in an offline manner since some application require a mix of online and offline use. Efficient incremental algorithms for updating the context are essential for achieving good online performance. Another challenge for decision procedures is in generating efficient *proofs* of validity and *explanations* consisting of the subset of input constraints leading to a contradiction. The construction of proofs can be engineered to track the behavior of the algorithm so that the facts that are recorded in the context are annotated with their justifications given in terms of existing annotations. Many applications require efficient explanations in terms of the smallest number of input constraints employed in deriving a contradiction, but computing minimal explanations can be NP-hard even when the decision algorithm is polynomial. For this reason, it is also unlikely that minimal justifications can be constructed as an extension to the basic decision algorithm. In most cases, it is possible to construct an irredundant explanation consisting of a conflicting set of input constraints for which every proper subset is satisfiable [dMRS04].

A ground decision procedure can be extended to determine the satisfiability of a propositional combination of constraints [BDS02,BLS02a,dMRS03]. The naïve approach to satisfiability modulo theories involves converting the formula to disjunctive normal form as a disjunction of conjunctions of literals, and using the ground decision procedure to determine the satisfiability of one of the disjuncts. This results in a formula that is too large. Modern SAT solvers are highly efficient and can be employed in determining satisfiability modulo theories. This is done by taking a Boolean abstraction $\overline{\phi}$ of the input formula ϕ where the atomic formulas A_i are replaced by corresponding Boolean variables p_i. In the *eager* approach [BLS02a], the Boolean abstraction is augmented with clauses in the Boolean variables that are implied when the variable is replaced by the corresponding atomic constraint. Thus if $A_1 \wedge \neg A_2 \wedge A_3$ is unsatisfiable according to a ground decision procedure, then $\neg p_1 \vee p_2 \vee \neg p_3$ is conjoined with $\overline{\phi}$. These *lemma* clauses are generated statically from the atoms in the given formula. Let the augmented formula be $\hat{\phi}$. A SAT solver is then invoked on the augmented formula $\hat{\phi}$. If the lemma generation process is complete, then $\hat{\phi}$ is satisfiable iff ϕ is satisfiable. In the lazy approach to satisfiability modulo theories [dMRS03], the SAT solver is first invoked on the Boolean abstraction $\overline{\phi}$. Within the SAT solver, the splitting step is modified to check that if the current context contains the literals $l_1, \ldots, l_n$ that correspond to the literals $L_1, \ldots, L_n$ in the original formula ϕ, then the split rule can only be invoked on an atom p corresponding to a concrete atom A when $L_1, \ldots, L_n, A$ and $L_1, \ldots, L_n, \neg A$ are both satisfiable according to the ground decision procedures. Also, when the SAT solver returns a satisfying assignment of literals $l_1, \ldots, l_n$, the ground decision procedure checks $L_1, \ldots, L_n$ for satisfiability. If the assignment is satisfiable then the original formula ϕ is also satisfiable, but if the assignment is unsatisfiable, then a lemma is generated from the conflicting literals and added to the clause set for the SAT solver.

Such an integration between a ground decision and a SAT solver can make enormous demands on the APIs of both components. For example, the SAT solver must provide operations for elaborating the splitting heuristic and for resuming the search with additional clauses. The ground decision procedures must contain operations for maintaining multiple contexts and for returning compact conflict sets. However, this kind of flexibility can impair efficiency in applications where these features are irrelevant.

4 Integration Frameworks: SAL and PVS

Powerful and flexible deductive components can be embedded within applications such as bounded model checking, test case generation, optimizing compilers, translation validators, predicate abstraction tools, and theorem provers. These integrations typically involve the use of one or two decision components as back-end engines. An integration framework allows multiple components to work cooperatively. PVS and SAL are verification frameworks that are designed to serve as such integration frameworks. PVS integrates a variety of focused deductive components within a proof checking environment. SAL on the other hand integrates a small number of verification components including model checkers and decision procedures for the analysis of transition systems. Together, SAL and PVS illustrate some of the characteristics of an integration frameworks.

4.1 SAL

SAL (Symbolic Analysis Laboratory) is an integration framework for combining program analysis, model checking, and theorem proving technology for the verification of transition systems. SAL facilitates this interaction through an intermediate language for describing transition systems and their properties. The SAL language is used to define transition system modules. Basic modules are defined in terms of the definitions, and initializations and transitions given by guarded commands. Modules are composed by means of synchronous and asynchronous composition. SAL contains a number of analysis tools including

1. A well-formedness checker (`sal-wfc`) that checks if SAL descriptions are syntactically correct and checks for partial type correctness. The SAL expression language is a sublanguage of PVS with a limited degree of predicate subtyping. Type checking expressions in a module with respect to subtypes generates invariant proof obligations. We are currently developing a more complete type checker for SAL.
2. A deadlock checker (`sal-deadlock-checker`) which uses the symbolic model checker to determine if transitions deadlock. Such deadlocks can arise for subtle reasons and the other analysis tools assume that the given SAL description is deadlock-free.
3. An explicit-state model checker that can be used to prove properties of SAL modules stated in linear-time temporal logic (LTL).
4. A symbolic model checker for LTL properties (`sal-smc`) built using the CUDD library.
5. A witness-generating model checker (`sal-wmc`) for properties expressed in the branching-time temporal logic CTL
6. A finite-state bounded model checker (`sal-bmc`) that is parametric with respect to a number of SAT solvers: ICS, ZChaff [MMZ+01], BerkMin [GN02], and Siege [Rya04].
7. An infinite-state bounded model checker (`sal-inf-bmc`) which is parametric in a number of ground decision procedures: ICS [dMOR+04b], SVC [BDL98], CVC [SBD02], UCLID [BLS02b], MathSAT [ABC+02], and CVC-Lite [BBD]. Infinite-state bounded model checking relies on procedures for satisfiability modulo theories.
8. A random simulator (`sal-path-explorer`) which can be used to generate random traces for testing LTL properties.
9. A finite trace generator (`sal-path-finder`) that uses a SAT solver to generate a computation of a given depth.
10. An interactive simulator (`sal-sim`) which provides access to the model-checking API for SAL.

11. A test case generator (`sal-atg`) which builds on a number of SAL model checking tools to identify input sequences that yield specified coverage targets.

SAL is an open integration framework in that it allows both scriptability and interaction. Many of the tools above are written as scripts in the Scheme language using various library functions for the syntactic and semantic manipulation of SAL descriptions. For example, scripts for test case generation can be defined to combine bounded model checking, slicing, and symbolic model checking to reach shallow and deep targets. The `sal-sim` environment can be used interactively to develop such scripts. The witnesses and counterexamples can also be explored interactively.

4.2 PVS

PVS is written in Common Lisp. It has interfaces to a BDD and BDD-based model checking package written in C, and provides interfaces with three external ground decision procedures; two written in Lisp and one in OCaml. It also integrates a number of PVS-specific inference procedures for operations such as rewriting, lemma introduction, quantifier instantiation, definition expansion, Boolean simplification, and case-splitting. The integration is built around a proof manager that serves as an interpreter for proof strategies defined in terms of the basic inference operations.

Integrating the BDD package was not difficult, mostly because it is written in C (C++ adds some complexity), and its API is well documented. In addition, it provides functions for allocating and freeing memory, which are easy to control from Lisp. The integration essentially involves defining foreign functions, translating PVS terms using the term constructors in the BDD API, and creating a proof rule (`bddsimp`) to invoke the package and interpret the results back to PVS.

Most decision procedures are black boxes that take as input a set of formulas, and return an indication of whether they are satisfiable or not. PVS requires a richer API. Because it is interactive, and subgoals may be postponed, the decision procedure must be able to maintain several distinct contexts, and switch between them. New facts are introduced incrementally, and the decision procedures must cope with this. Finally, some formulas are asserted as valid, but many are simply tests, for example to check the condition in a conditional rewrite. These should not have an effect on the original context.

The first decision procedure added to PVS was originally written by Shostak [SSMS82] in the late seventies and heavily modified since then. It was the core of the theorem prover for EHDM [vHCL+88,RvHO91], and since it was written in Lisp, the interface was quite straightforward.

A new ground decision procedure package was implemented in Common Lisp in 1996. The only new integration issue was that we wanted to support both decision procedures, with the user controlling which to use for a given proof. Since the underlying states were not compatible, the PVS prover was modified so that calls to the decision procedure carried along two states — one for Shostak and one for the new decision procedure; and because the API was more complex, some extra calls to the decision procedure were necessary. This was partially hidden using macros, but it clearly was inefficient.

ICS was developed in OCaml, which created a new set of problems. First of all, to support a third decision procedure meant modifying the macro and adding yet another state to the decision procedure invocations. Rather than do this, we created a new PVS decision procedure API as described below. With this, the prover just invokes whichever decision procedure is current, without any knowledge of the implementation details.

The other problems are primarily due to the fact that both Lisp and OCaml have interpreters and garbage collectors. This means that terms created by OCaml could be moved by the OCaml garbage collector unless they were *registered*. But once registered, the terms must be unregistered when Lisp is finished with them, or memory will be quickly used up. Fortunately, Allegro Common Lisp provides a *finalization* facility that allows hooks to be invoked when an entity is garbage collected. Thus all pointers to data in the OCaml world are wrapped in lists, and a finalization is associated with this list. When the list is garbage collected by Lisp, the finalization function is invoked, and it deregisters the pointer, allowing the data to be garbage collected by OCaml. Ensuring that there are no memory leaks or lost references between Common Lisp and OCaml is quite delicate. Debugging can be difficult because bugs are nondeterministic and are not easy to reproduce or correct.

Another issue with OCaml is the type system. ICS has types of terms and atoms, and if a term is provided where an atom is expected the usual effect is a segmentation fault, killing the whole process, Lisp included. Debugging this usually involves adding lots of print statements to determine which call is the culprit; remembering to flush the print buffer with each call to make sure it is seen before the process dies. The problem is that though Lisp is considered untyped, it actually checks for types at runtime. OCaml is typed, but it strips the types out at runtime, and assumes all calls are safe. The API for ICS has recently been modified to provide a sort of runtime typechecking that makes debugging much easier.

The API for the ground decision procedure in PVS is defined by a set of methods. The primary ones are

`empty-state` - creates a new state for the decision procedure.
`process` - given a PVS boolean expression and a state, returns either
  `FALSE` : the expression is inconsistent.
  `TRUE` : the expression is already known to be true.

new-state : a new state, in which the expression has been asserted.
`state-changed?` - checks whether two states are equivalent.

The `process` method generally invokes a function that translates a PVS expression to terms in the language of the underlying decision procedure.

PVS also has an `addrule` operation to introduce new inference rules for proof construction. The MONA decision procedures for the WS1S fragment of the PVS logic have been introduced this way. The PVS strategy language is used to add compound proof strategies that are defined using the primitive inference rules and previously defined strategies.

4.3 Characteristics of an Integration Framework

Both SAL and PVS illustrate some of the typical characteristics shared by integration frameworks. Both systems are built around a description language for communicating information through terms, formulas, modules, and theories. A shared language is a key feature in integration since it allows components to interact without tool-specific translations and explicit invocation. SAL and PVS also offer a powerful base set of components to bootstrap the integration process. Such components can be built for high performance using efficient representations that can be kept internal to the component. PVS has operations for adding inference components. In PVS, inference components share the same signature as proof constructors mapping goals to subgoals. SAL components are more diverse in terms of their signatures. PVS offers a scripting language for defining new proof strategies in terms of the base components and previously defined proof strategies. SAL uses Scheme as its scripting language. The PVS strategy language ensures that all defined strategies are sound by construction, whereas programming scripts in Scheme for SAL yields no guarantees. PVS has a proof manager that handles the book-keeping involved in proof development. The proof manager keeps track of the unfolding proof structure, the completed branches, and the pending subgoals. A few such managers have been developed for SAL for navigating through a simulation or for exploring witnesses and counterexamples. Systems such as HOL [GM93], Isabelle [Pau94], and STeP [MT96] are also similar examples of integration frameworks.

5 Formal Architectures for Tightly Coupled 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. In a tightly coupled setting, the interaction has to be mediated through a well-defined architecture. Tight coupling therefore poses theoretical challenges that are not present in the loosely coupled case. A formal 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 over specific theories [GRS]. This framework is based on the concepts of inference systems and inference modules and a theory of compositionality and refinement for inference systems.

An inference structure is a pair $\langle \Psi, \vdash \rangle$ of a set of *logical states* Ψ and an *inference relation* $\vdash$ between states. Each logical state is of the form $\kappa_1 | \ldots | \kappa_n$ which represents a bag of zero or more configurations κ_i. The $|$ operator is associative and commutative with the special unsatisfiable configuration $\bot$ as its zero element. The inference relation relates a pair of logical states as $\psi \vdash \psi'$, where ϕ is the premise state and ϕ' is the conclusion state. An inference structure is an inference system for a theory if the inference relation is

1. *Conservative* in that the premise and conclusion must be equisatisfiable with respect to the class of models of interest, i.e., the theory.
2. *Progressive*, i.e., the inference relation should be well-founded.
3. *Canonizing* so that a state is *irreducible*, i.e., has no $\vdash$ successors, only if it is either $\bot$ or is satisfiable in the theory.

Decision procedures given by inference systems can be immediately seen to be sound and complete since an initial state eventually leads, by progressive, through a series of inference steps to an irreducible, equisatisfiable state, by conservativity, that is either $\bot$ or satisfiable, by canonicity. The satisfiability procedure given in Figure 1 is an example of an inference system.

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 different from those of an inference system. For an inference module, the inference relation must be

1. *Local*, so that the premise is a single configuration.
2. *Progressive*, as with inference systems.
3. *Strongly conservative*, in preserving models and not just satisfiability.
4. *Relatively canonizing*, so that for an irreducible state, the V component must be extensible to a closure $Cl(V)$ which is $\mathcal{T}_0$-satisfiable, and any $\mathcal{T}_0$-model of $Cl(V)$ must be extensible to a $\mathcal{T}$-model of $V;\theta$.

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 theories, provided the theories involved satisfy certain conditions.

A generalized component can be defined to capture the abstract behavior of typical inference modules as shown in Figure 2. The configuration has the form $(K; G; V); E$, where K is the set of shared variables, G is the unprocessed input constraints, V the shared constraints, and E the local constraints. The **Contrad** rule detects a contradiction using the theory-specific decision procedure. The **Input** rule moves a T_0-constraint from the input G to V. The **Abstract** rule replaces an input subterm t that is in the theory T by a fresh variable x while recording $x = t$ in E. The **Branch** rule branches by adding an unresolved shared constraint P from a finite set of basis atoms, to V on one branch, while adding $\neg P$ to V on the other branch.

Contrad	$\dfrac{(K; G; V); E}{\bot}$ if $\mathcal{T} \models V, E \to \bot$
Input	$\dfrac{(K; Q, G; V); E}{(K; G; Q, V); E}$ for $\Sigma_0[K]$-literal Q
Abstract	$\dfrac{(K; G\{t\}; V); E}{(K; G\{x\}; V); x = t, E}$ for pure $\Sigma[K]$-term a, fresh x
Branch	$\dfrac{(K; G; V); E}{(K; G; P, V); E \mid (K; G; \neg P, V); E}$ if $P \notin V, \neg P \notin V$ for $\Sigma_0[K]$-basis atom P

Fig. 2. A Generalized Component

Inference modules can be shown to yield a modular presentation of known combination results such as those of Nelson and Oppen [NO79], Shostak [Sho84], 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 is based on inference modules. The theory of 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 and deeper decision problems.

6 A Tool Bus for Loosely Coupled Integration

SAL is intended as a framework for integrating a number of heterogeneous verification components including a variety of model checkers and decision procedures. However, SAL currently lacks a formal framework for the loose coupling of heterogeneous components. Such a framework 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.

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

The basic primitive in the SAL tool bus is an assertion of the form $T : P \vdash J$ 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 can build such judgments by forward chaining from existing judgments or backward chaining through the generation of proof obligations. The tool bus thus serves as a uniform framework for interacting with existing components, adding new components, defining scripts, coordinating garbage collection, and managing the evidence generated.

7 Conclusions

Lampson in his skepticism about component-based software development, and Boyer and Moore 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 trade-offs 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 preliminary plans for a SAL tool bus architecture that combines various analysis tools within a framework for constructing reproducible evidence.

References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proc. of CADE'02*, 2002.

[ADG+01] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001. Springer-Verlag.

[Arc00] Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):139–181, 2000.

[BBD] Clark Barrett, Sergey Berezin, and David Dill. CVC Lite.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Computer-Aided Verification, CAV '2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249, Copenhagen, Denmark, July 2002. Springer-Verlag.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Hu and Vardi [HV98], pages 505–510.

[BLS02a] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification, CAV '2002*, volume 2404 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, July 2002. Springer-Verlag.

[BLS02b] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of CAV'02*, volume 2404 of *LNCS*, 2002.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[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.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[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)*, number 1785 in Lecture Notes in Computer Science, pages 78–92, Berlin, Germany, March 2000. Springer-Verlag.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962. Reprinted in Siekmann and Wrightson [SW83], pages 267–270, 1983.

[dMOR+04a] 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.

[dMOR+04b] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *IJCAR 2004*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, 2004. Springer-Verlag.

[dMOR+04c] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV '2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[dMRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Warren A. Hunt, Jr. and Fabio Somenzi, editors, *Computer-Aided Verification, CAV '2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, CO, July 2003. Springer-Verlag.

[dMRS04] Leonardo de Moura, Harald Rueß, and Natarajan Shankar. Justifying equality. In *Proceedings of PDPAR '04*, 2004.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, 1960.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Möller. Mona 1.x: New techniques for WS1S and WS2S. In Hu and Vardi [HV98], pages 516–520.

[For03] Formal Methods Program. Formal methods roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003. Available at `http://fm.csl.sri.com/doc/roadmap03`.

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

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[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.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.

[GRS] H. Ganzinger, H. Rueß, and N. Shankar. Modularity and refinement in inference systems. Under preparation.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

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

[LGvH+79] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. CSD Report STAN-CS-79-731, Stanford University, Stanford, CA, March 1979.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[MT96] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, New Brunswick, NJ, July/August 1996. Springer-Verlag.

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

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[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.

[RvHO91] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[Rya04] Lawrence Ryan. The siege satisfiability solver, 2004.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: a cooperating validity checker. In *Proc. of CAV'02*, volume 2404 of *LNCS*, 2002.

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

[Som98] F. Somenzi. CUDD: CU Decision Diagram package, 1998.

[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.

[SSMS82] R. E. Shostak, R. Schwartz, and P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.

[SW83] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers on Computational Logic, Volumes 1 & 2*. Springer-Verlag, 1983.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, Genova, Italy, April 2001. Springer-Verlag.

[vdBvDH+01] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370. Springer-Verlag, 2001.

[vHCL+88] F. W. von Henke, J. S. Crow, R. Lee, J. M. Rushby, and R. A. Whitehurst. The EHDM verification environment: An overview. In *Proceedings 11th National Computer Security Conference*, pages 147–155, Baltimore, MD, October 1988. NBS/NCSC.