Presented at the Monterey Workshop 2004 in Baden, Vienna

The Interface is the Message[a]

N. Shankar (with John Rushby, Sam Owre, Harald Ruess, Leonardo de Moura, Ashish Tiwari)

URL: http://www.csl.sri.com/~shankar/LEP.html

Group URL: http://fm.csl.sri.com

Computer Science Laboratory

SRI International

Menlo Park, CA

[a]Supported by NSF Grant Nos. CCR-ITR-0326540 and CCR-ITR-0325808, DARPA REAL project, and SRI International.

An Opening Quote

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

Overview

- Effective deductive engines are critical to automated verification.
- Most uses are *embedded*: Type checkers, compilers, constraint solvers, model checkers, test case generators, program synthesizers.
- We need powerful *components* with rich semantic interfaces as well as flexible *integration frameworks*.
- Achieving modularity through the integration of independent components offers significant theoretical and practical challenges.
- We report on our attempts at meeting this challenge with PVS, SAL, and ICS.

Outline

- Arguments for and against modularity
- Some useful components
- Integration examples and experience
- Integration Frameworks
- Formal Architectures

The Modularity Challenge

Boyer and Moore's 1985 paper contains a long litany of problems that must be overcome when integrating decision procedures into a theorem prover.

The issues they faced are still quite relevant:

- Need to manage language incompatibilities between client and server.
- The communication overhead with an external component can degrade performance.
- Need term information from the decision procedure to activate lemmas.
- Need proof and dependency information from the decision procedures.
- Pushing in and popping out of contexts can be expensive.

The Modularity Challenge

Butler Lampson has argued that the only successful components are those like databases and compilers that provide complex functionality through a relatively narrow interface.

- Engineering to a flexible API is quite difficult.
- Ideas and algorithms are more portable than implementations.
- It is often easier to engineer and implement non-modular interaction without the overhead.
- For any specific application, the functionality available may be too much and/or too little.
- Enriching the interface often rules out certain optimizations.

Why Modularity?

- Individual tools such as decision procedures, theorem provers, and model checkers have become extremely specialized and sophisticated.
- The scope of formal analyses has been widened to include test case generation, type checking, runtime verification, static analysis, controller synthesis, and proof generation.
  - These analyses do need to be integrated.
- Diverse applications such as hardware, system code, communication protocols, fault tolerance, and embedded systems, require slightly different tool combinations.
  - Building/maintaining specialized tools is expensive.

Components

What is a Component?

A self-contained body of code whose functionality is accessed through a well-defined interface.

A component must typically be reusable in a number of different contexts.

From the point of deduction, we can identify three (possibly overlapping) classes of components:

1. Libraries: BDDs, term manipulation and indexing, computer algebra, numerical analysis, polyhedral operations, formalization libraries.
2. Offline Procedures: SAT solvers, model checkers, proof search procedures.
3. Online Procedures: Ground decision procedures, constraint solvers, rewrite engines.

A Library Component: BDDs

Binary Decision Diagrams (BDDs) are a widely used DAG-structured canonical representation for Boolean expressions.

BDD packages do come with a rich API for building boolean terms, applying quantification, taking fixpoints, clustering, variable ordering, garbage collection, and for displaying information in readable form.

The APIs of different BDD packages are not fully standardized, but the usability of a package is largely determined by the richness of its API.

Is standardization of APIs desirable/tenable?
Will standardization of APIs restrict progress?

Offline Component: SAT Solvers

A SAT solver for propositional logic needs to indicate if a given set of clauses (e.g., $p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q$) is satisfiable or unsatisfiable.

Modern SAT solvers are based on the 1960 work of Davis and Putnam

$$\frac{\Gamma}{\Gamma, p \mid \Gamma, \neg p}\ \mathit{split} \qquad p \text{ and } \neg p \text{ are not in } \Gamma.$$

$$\frac{\kappa, C \vee \bar{l}, l}{\kappa, C, l}\ \mathit{unit}$$

A SAT solver must provide satisfying assignments, but also allow additional clause constraints, provide proofs and interpolants, and . . .

Despite the availability of fast SAT solvers, most groups find it easier to implement their own custom SAT solvers.

Online Component: Ground Decision Procedures (GDP)

GDPs form the core of many proof engines and handle problems such as, for a theory $\mathcal{T}$,

- Word problem (WP): Does $\mathcal{T} \models P$ for Σ-atom P?
- Uniform Word Problem (UWP): Does $\mathcal{T} \models \Gamma \rightarrow P$ for the set of Σ-atoms Γ and Σ-atom P?
- Clausal Validity Problem (CVP): Does $\mathcal{T} \models Q_1 \vee \ldots \vee Q_n$ for Σ-literals (Σ-atoms or their negations)?

APIs for Ground Decision Procedures

It has been argued that decision procedures should not be black-box constructions, but should be

1. Online: Process assertions dynamically to construct contexts.
2. Resettable: Allow saving, backtracking, and switching between contexts.
3. Queriable: Allow queries and simplifications with no change to the context.
4. Integrable: Allow fine-grained integration with other tools, possibly implemented in a different language, with seamless garbage collection and error signalling.
5. Evidence producing: Proof objects and counterexamples.

ICS: Integrated Canonizer and Solver

SRI's ICS is a GDP that provides an ask/tell API as described.

It is implemented in OCaml and provides both a C and Lisp interface.

ICS supports a modular combination of linear arithmetic equality and inequality over rationals and integers, uninterpreted term equality, arrays, bit vectors, datatypes.

ICS can be used offline through a shell API as well as interactively through a read-eval-print loop.

ICS has been integrated within `sal-inf-bmc` and the ECL^iPS^e constraint programming system.

Integration

Integrating GDPs and SAT solving

GDP + SAT is a good test-case for the effectiveness of integration for solving SAT(mod $\mathcal{T}$).

CVP can be used as a subroutine in proving the validity of quantifier-free formula, e.g,
$x + 2 * y = z \rightarrow f(4 * y - z) = f(-x)$.

A naïve approach to SAT(mod $\mathcal{T}$) would be to convert the formula to CNF and check the validity of each clause with the GDP, but this can be prohibitively expensive.

Several recent approaches leverage the efficiency of SAT solvers.

The *eager* approach (used by UCLID) reduces satisfiability (in $\mathcal{T}$) of A to the Boolean satisfiability of $A \wedge L$ for a set of generated *lemmas* L.

Lazy Integration of GDP and SAT

The *lazy* approach (used by ICS, Verifun, CVC) employs a GDP to examine splitting decisions and ignore assignments that are unsatisfiable in the theory, while adding clause lemmas reflecting this unsatisfiability.

For most efficient backtracking, the GDP should return the minimal conflict set of literals. This often requires irredundant proofs. (See *Justifying Equality*.)

ICS has a SAT solver (written in C++) that is uses the ICS API.

ICS benchmarks well against the other systems (CAV'04).

Still, many things could be done better within a unified SAT/GDP development.

Integration Frameworks

SAL: Symbolic Analysis Laboratory

SAL was initiated precisely to serve as an integration testbed for transition system analysis tools.

SAL has an intermediate language (designed with David Dill and Tom Henzinger) for the modular description of transition systems, both finite and infinite.

The analysis tools that have been developed for SAL include

1. An explicit-state model checker (`sal-esmc`)
2. A symbolic model checker for LTL properties (`sal-smc`)
3. A witness-generating model checker for CTL properties (`sal-wmc`)
4. A finite-state bounded model checker (`sal-bmc`)
5. An infinite-state bounded model checker (`sal-inf-bmc`)
6. A test case generator
7. A symbolic simulator (`sal-sim`)

Scriptable Model Checking

Interfaces to the CUDD and ICS (and other decision procedure) packages are available through a Scheme front-end.

Tools are defined using Scheme scripts (with lots of flags to control different parameters).

Test case generation can try different strategies, e.g., BMC for shallow targets, followed by slicing, and SMC for deep targets.

The model checker can be used interactively, much like a proof checker.

The symbolic witnesses and counterexamples generated by `sal-wmc` can also be explored interactively as a game.

What is missing is a *tool bus*, but more on this later.

PVS as an Integration Framework

PVS is a general-purpose specification and verification framelwork integrating a number of deductive tools.

The PVS higher-order logic is quite convenient for expressing mathematical content and for embedding other logics.

The PVS theorem prover is driven by a rule interpreter that applies inference rules to a leaf node, the proof goal, in a proof tree.

There is an API for adding external inference procedures (but should be used sparingly).

A strategy language is used to define compound proof procedures such as those for induction, instantiation, and rewriting.

Inference Components in PVS

A BDD-based Boolean simplifier and model checker are essentially used through *offline* foreign function calls.

Translating from PVS to BDDese and back can be quite complicated.

Garbage collection is not a problem unlike online integration.

The MONA library for WS1S is used to decide monadic second-order logic formulas.

PVS has an interface for adding new GDPs for *online* use. The GDP API must provide a few basic operations.

Ground evaluator for generating efficient, safely destructive (Lisp) code from PVS operations.

Characteristics of an Integration Framework

SAL and PVS illustrate many of the elements of an integration framework:

- A uniform language for communicating across components.
- A simple API for adding inference components.
- A powerful base set of components.
- A scripting language for defining complex inference schemes from more primitive ones.
- A proof manager.

The devil is in the details, though.

PVS as a Component

PVS is used as a back-end deduction engine in several ways:

- PC/DC is an embedding of the Duration Calculus interval temporal logic in PVS.
- **Ag** is a similar encoding of first-order dynamic logic.
- Graf and Saïdi's Invariant Generator and InVest use PVS to discharge abstraction proof obligations.
- PVS-Maple instruments the Maple CA system to generate PVS proof obligations that ensure that the side conditions of each operation are satisfied.
- The LOOP tool for Java verification at Nijmegen is used for the symbolic execution of Java programs.
- The PBS system uses extended type checking in PVS to give a simple but effective implementation of the B method.
- The TAME system is embeds proof methods for timed I/O automata in PVS.

Looking Ahead

The Interface is the Message

Formal Architectures for Integration

A science of software design, then, must be about radical, not about normal, design. Its chief goals are to explicate the properties of decompositions and combinations, in requirements, specifications, assumptions about problem worlds, and software components, and to support the tasks of choosing and making appropriate decompositions and combinations. Michael Jackson

Tractable correctness by construction results can provide significant guidance in the design process. Their lack leaves a posteriori verification of the designed system as the only means to ensure its correctness (with the well-known limitations). G. Gössler and J. Sifakis

Formal Architectures

Composition is the key to good software engineering.

A good composition framework provides an interface for plugging in components so that they can interact safely with each other.

Components can be analyzed independently in terms of their composition interface (*composability*).

System properties emerge from component properties (*compositionality*).

The system evolves smoothly through the addition of new components, and the modification and refinement of existing ones.

A Modular Architecture for Combination Decision Procedures

Most natural conjectures employ a combination of theories: arithmetic, uninterpreted functions, lists, and arrays, where the individual theories often have efficient decision procedures.

There is a long history of work on (non-modular) combination decision procedures going back 25 years to dating Nelson/Oppen and Shostak.

Recently, we (Ganzinger, Rueß, and S.) have developed a mathematical theory of inference modules, their composition, and refinement.

ICS is based on this architecture.

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 state ψ is of the form $\kappa_1 | \ldots | \kappa_n$, where each κ_i is a *configuration*.

An inference system for a theory $\mathcal{T}$ is an inference structure that is

1. **Conservative:** The inference relation preserves satisfiability.
2. **Progressive:** The inference relation is well-founded.
3. **Canonizing:** A state is irreducible only if it is either $\bot$ or is satisfiable.

An inference system is a sound and complete decision procedure for satisfiability.

Union–Find Inference System

Delete	$\dfrac{x = y, V; F}{V; F}$ if $F(x) \equiv F(y)$
Merge	$\dfrac{x = y, V; F}{V; F \circ \{orient(F(x) = F(y))\}}$ if $F(x) \not\equiv F(y)$
Contrad	$\dfrac{x \neq y, V; F}{\bot}$ if $F(x) \equiv F(y)$

$orient(x = y)$ returns $y = x$ if variable x precedes variable y, and $x = y$, otherwise.

Fusion: $F \triangleright F'$ as $\{c = F'(d) | c = d \in F\}$, and

Composition: $F \circ F'$ as $(F \triangleright F') \cup F'$.

Inference Modules

An inference module for a theory $\mathcal{T}$ is an *open* version of an inference system.

Its configurations consist of a shared blackboard γ consisting of inputs shared constraints, and a theory-specific part θ that is like a private notebook.

The shared constraints are in a common theory $\mathcal{T}_0$.

The inference relation must be progressive, *strongly conservative* over the shared variables, and *relatively canonical*.

Two *compatible* inference modules can be composed, with a configuration has the form $\gamma; \theta_1; \theta_2$, and the result is an inference module for the union of the theories, under some semantic restrictions.

Generalized Components

Contrad	$\dfrac{(K;G;V);E}{\bot}$ if $\mathcal{T} \models V, E \rightarrow \bot$
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

$(K;G;V)$ is the blackboard with input G, shared workspace V, and shared free variables K.

E is the private notebook.

Nelson–Oppen and Shostak combination methods are part of a refinement hierarchy implementing generalized components.

General Schemes for Combining Decision Procedures

The modular approach to combination decision procedures unifies/generalizes/simplifies previous work on this topic.

It does not cover the case when we combine two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ together with some bridging axioms, e.g., lists and integers bridged by *length*.

The framework can be extended to (non-progressive) semi-inference systems.

A more powerful composition framework should work with heterogeneous logics and logical judgments, i.e., a *tool bus*.

A SAL Tool Bus

- A read-eval-print loop for invoking and interacting with different tools.
- Judgments of the form $T : P \vdash J$ which says that tool T claims the validity of judgment J with proof P.
  - Judgments can be syntactic (A is a well-formed formula in Σ, A is simpler than B) as well as semantic (A is satisfiable in $\mathcal{T}$).
- An API for adding new tools and libraries with accompanying judgments.
- A scripting language for composing judgment derivations through forward and backward chaining.
- The tool bus delivers extensible functionality, evidence management, reproducibility, garbage collection, independent of operational details.

Varieties of Judgment

- A is a well-formed formula.
- A is a well-typed formula in context C.
- a is a BDD representing the formula A.
- $\mathcal{C}'$ is a decision procedure context representing Γ.
- A is satisfiable in theory $\mathcal{T}$.
- Γ is a satisfying assignment for A.
- Γ is a minimal unsatisfiable set of literals.

The plan is to employ a metatheoretic framework like Twelf to carry out semantic evidence management.

Previous Tool Integration Projects

- HOL-Voss (Joyce and Seeger) integrates symbolic trajectory evaluation with HOL.
- PROSPER: A sockets-based architecture integrating HOL98, Prover, and SMV.
- ETI: A tool interchange architecture for various model checking tools.
- VeriTech: Semantic interchange framework for diverse analysis tools.
- Gordon formalizes BDD operations in HOL through judgments: If BDDs a and b represent formulas A and B, then `bdd/and`(a, b) represents $A \wedge B$.
- Berezin's SyMP is an inference rule-based approach to combining theorem proving and model checking.

Conclusions

Integration of components is a tough, tough challenge, particularly for inference components.

Practical challenges involve designing interfaces that allow flexible use without loss of efficiency.

The theoretical challenges are in designing integration architectures that mediate fine-grained interaction between inference components.

More generally, we need to move from components to composition-centric architectures that are safe/correct-by-construction, as exemplified by inference systems/modules.