

Integrating Tools for Practical Software Analysis

Aaron R. Bradley Henny B. Sipma
Sarah Solter Zohar Manna

Computer Science Department
Stanford University
Stanford, CA 94305-9045
{arbrad,sipma,sbsolter,zm}@theory.stanford.edu

Abstract. The lack of integration between prototype implementations of results of research (“tools”) blocks progress toward direct application of formal methods research in software engineering settings. We survey a host of tools, examining how their integration would increase their power and benefit future research and application. Based on this analysis, we describe a hypothetical and idealized Tool Integration Package (TIP). A TIP has two goals: first, it accelerates the research process by providing a range of tools in an integrated setting; and second, it serves as a ready-to-use tool for application in industry.

1 Introduction

Theoretically and practically, research in formal methods for software is closing in on the applicability barrier, that division between ivory tower research and research that gets noticed. But one problem remains: stand-alone tools. A significant portion of software that is released to support published research works in isolation. This isolation hampers future research efforts. Worse, the desired consumers — companies that produce software — are not interested. Even if an amazing new analysis is demonstrated to be an order of magnitude more accurate at a fraction of the computational cost, it almost certainly will be added to the growing number of implemented but unused research. Companies cannot waste resources on integrating a hundred prototype tools written in almost every language imaginable, only to have them produce a ream of disparate warnings, errors, proofs, and debug statements, 99% of which is pure speculation or superfluities. When we present them with one tool, they will listen.

Integrating tools is the next practical step in the development of our research. We will argue that a moderate effort now at integration will reward both the program analysis and the software engineering communities. To direct our look at tool integration, we will focus on these two groups — program analysis researchers (which includes researchers in, *e.g.*, formal methods and compiler theory) and industrial software engineers. Of course, the domain of formal methods research is wide, encompassing not only software analysis, but also analysis of hardware and continuous, real-time, and hybrid systems, to name a few sub-domains. Readability demands that we limit our scope, however, so we will focus

on software tools. The following two hypothetical situations motivate our argument.

The researcher's perspective Suppose a researcher has developed the theory for a new analysis. It works well on handmade transition systems; it works well on the researcher's favorite pseudo-language. Now he wants to test it on code. After choosing his favorite programming language, he faces several hurdles. First, he needs to parse the code. Programming languages are complex beasts, and parsing them was a research endeavor in itself a few decades ago. Clearly, writing his own parser is not an option. Second, even after solving the parsing problem, perhaps using a wonderful package like CIL [53], he realizes that extracting even rudimentary semantics from the parsed code is yet another research endeavor. Which references point to this piece of memory, or to this object? Which methods can be called here? On which type is this expression working? Is this path even feasible? What about this assertion — can it be trusted? There is no hope of implementing the analysis without relying on many other analyses (*e.g.*, alias analysis, class hierarchy analysis, invariant generation, *etc.*) and foundational tools (*e.g.*, decision procedures).

Assume he finally manages to write and run a few basic analyses, compile and use several different open source packages from fellow researchers, and compile and call a decision procedure package, having written a few wrapper API's in the process, because some of the downloads were in a different language. Of course, this effort is not publishable and not funded, so it has to be done on the side. Hence, there is no time to do it properly and even less time to write documentation. While the researcher may in the end be able to do his analysis and publish a paper with impressive results, the effort that transferred his technology to real code will be largely wasted: his code will not be reusable. Other researchers have to go through the same process. Even the researcher himself will probably have to repeat it for future projects. Clearly, implementing one's idea to work on real code is fraught with tedious tasks and misadventure. There must be a better way.

How can we make the process incremental, such that new analyses can be built on top of existing analyses, rather than each being a separate endeavor? We envisage the following scenario. The same researcher, who was struggling above, now has access to a TIP (specifically, a Tool Integration Package) for his favorite language, say \mathcal{L}_1 . It parses \mathcal{L}_1 , canonicalizing some of the convoluted aspects of \mathcal{L}_1 and producing the intermediate language \mathcal{L} , which is also produced by the TIPs for \mathcal{L}_2 and \mathcal{L}_3 (really the same TIP, just with different front end parsers). Common elements of the languages are factored out in \mathcal{L} . He implements his method, but now has access to other analysis methods. For example, an alias analysis annotates variables of \mathcal{L}_1 with identifiers to memory objects. A class hierarchy analysis narrows down the possible methods that can be called at a particular point. Another tool generates invariants, adding them as annotations to program points. The package also provides foundational tools such as SAT solvers and decision procedures, whose inclusion does not require more than a simple function call. Finally, to enable evaluation, the TIP comes with millions

of lines of open source code, classified by application area, complexity, and perhaps several more criteria, which can be used as the basis for experiments. If the method is an alternative to other published methods, this benchmark code provides a basis for comparison. These other published and implemented methods may already be available in the TIP, allowing the methods to be called in parallel and compared. If the new method is competitive, the researcher cleans up his code, documents it, provides API's for other analyses and submits it for inclusion in the TIP, where it will be available for others to use.

The practitioner's perspective Looking at the other end of the ivory tower, we see software companies having problems of their own. In many software firms, the majority of software engineers are testers rather than developers. Of course, companies would like to change that ratio, increase their software's robustness, and decrease their time-to-market. They do not look, however, to formal methods for answers. The main reason for ignoring the results of this research is that the methods and prototype tools seem irrelevant to their domain. They have not been demonstrated to apply to real software projects, are not known to scale, and perform isolated analyses. Companies simply cannot justify the effort of integrating these isolated tools in their software development process. There is no guarantee that even the next version of the same tool will not require yet another integration effort.

The availability of a TIP may help, especially if some standardization is respected. The TIP is a portal to transfer research results to practitioners. Build files may include selected analyses provided by the TIP, instructing the TIP to present relevant errors, warnings, speculations, and proofs in a single, readable, integrated summary. In the summary, the TIP displays warnings, speculations, and proofs hierarchically according to priority and confidence. Through this portal, researchers provide increasingly sophisticated analyses with improved accuracy, while the practitioners apply a seemingly stable tool via a solid front end. The software developers can focus on customization of input and output of the TIP, rather than having to worry about the correct integration of yet another version of yet another tool. From the software developer's perspective, the TIP is simply an increasingly intelligent compiler.

Admittedly, this TIP oversimplifies the problem and solution. Nonetheless, it is clear that tool integration is a necessary next step for our field. In addition to practical integration of tools, this survey also examines possibilities for integration of fundamental methodologies. We feel that such activity is an unexplored mine of research opportunity.

Paper organization The survey is organized as follows. Section 2 starts by surveying a set of isolated tools for software analysis. The tools represent several classes of theoretical foundations, including type theory, dataflow analysis, model checking, and theorem proving. After summarizing their capabilities, we zoom out to examine their foundations, looking for opportunities for integration of

methodologies. In Sections 3, 4, and 5, we turn to tools that support integration. Language (Section 3) and parsing (Section 4) tools include languages for representing systems, frameworks for parsing code and writing new analyses, and packages for translating between input languages. Foundational tools (Section 5) offer support for deep analyses in the form of decision procedures and mathematical tools. Section 6 discusses some of our experiences with integrating tools. It also proposes ideas for future integration of language, parsing, and foundational tools. Having closed the book on integration issues for researchers, Section 7 turns to software engineers, their needs, existing software to address these needs, and suggestions for future software. Finally, we make a few closing observations and suggestions in Section 8.

1.1 Further Reading

This paper presents tools based on fundamental research. We briefly give some historical foundation for the tools. For a comprehensive presentation of compiler theory, including static analyses, see [51]. Model checking [17, 57] and symbolic model checking [11] and bounded model checking (BMC) [8] were originally directed at hardware verification, but are increasingly used for software verification. See [18] for a textbook presentation. Abstract interpretation [21] was introduced as a general framework for constructing and executing state-based analyses of software and hardware. In [43], we find the first instance of an abstract interpretation, while in [22] is presented the famous abstract domain of polyhedra. Deductive verification has deep roots; see [47] for a textbook presentation and bibliographical references.

Engler’s and Musuvathi’s experience with static analysis and software model checking [31] gives an interesting perspective on deep versus shallow analyses. Rutar, Almazan, and Foster [58] summarize their experience with five Java bug finding tools. Additionally, they describe a meta-tool for combining the output of bug finding tools. It uses statistical metrics to pinpoint suspicious Java classes, based on the output of multiple bug finding tools. In [42] Jackson and Rinard argue that in the future, static analyses will be model-driven, deep analyses.

2 Tools

2.1 A Sampling of Tools

This section provides a description of a sample of tools. The sample is by no means exhaustive; many great tools were missed. We summarize key points in Table 1.

Java Pathfinder Java Pathfinder (JPF) [62] is an explicit state model checker for Java. The first version translated Java to Promela and applied the Spin model checker [39]; the second version implements a model checker specific to Java, including a new JVM. In addition to model checking techniques, including efficient

Tool	Language	B/V	Analysis	Foundation	Interaction	Free
JPF	Java/Bytecode	B/V	Deep	Explicit MC	Moderate	Yes
Bandera	Java	B/V	Deep	Abstraction, MC	Extensive	Yes
CMC	C/C++	B	Deep	Explicit MC	Moderate	No
CBMC	C/C++	B/V	Deep	Bounded MC	Minimal	Yes
BLAST	C	B/V	Deep	Predicate abstraction	Moderate	Yes
SLAM	C	B/V	Deep	Predicate abstraction	Moderate	No
MAGIC	C	B/V	Deep	Predicate abstraction	Moderate	Yes
Spin	Promela	B/V	Deep	Explicit MC	Minimal	Yes
SAL	Pseudo	B/V	Deep	MC	Minimal	Yes
FindBugs	Java	B	Shallow	Static Analysis	Minimal	Yes
PREFix	C/C++	B	Shallow	Static Analysis	Minimal	No
xgcc	C/C++	B	Shallow	Static Analysis	Minimal	No
ESP	C	B/V	Moderate	Static Analysis	Minimal	No
CQual	C	V	Moderate	Type checking	Moderate	Yes
Astrée	C	V	Deep	Abstract interpretation	Minimal	No
ESC	Java, Modula-3	B	Moderate	Theorem proving	Extensive	Yes
STeP	Pseudo	V	Deep	Theorem proving	Extensive	Yes
PVS	Pseudo	V	Deep	Theorem proving	Extensive	Yes

Table 1. Summary of tools. “MC” abbreviates “model checking.” “B” indicates that the tool is *primarily* a bug-finder; “V” indicates that the tool is *primarily* a verifier. **Analysis** is a rough estimate of how “deep” the analysis is, based on, *e.g.*, whether the analysis is more syntactic or semantic. **Foundation** is a succinct categorization of the main research ideas implemented. **Interaction** rates the amount of user interaction required to get results. **Free** indicates whether the tool is available for research use.

state representation, JPF also implements predicate abstraction via the generation of abstract programs, program slicing using Bandera [20], partial-order reduction, and runtime analyses for race and deadlock detection. Partial-order reduction uses static analyses to determine independence between statements in parallel threads. JPF represents an integration of much research.

Applying JPF requires defining an environment (in Java). Additionally, the user can supply invariants and assertions; for predicate abstraction, the user must also provide a set of predicates. The developers cite partial automation of environment construction as a future goal.

As JPF works on bytecode, rather than source, it can be applied to other languages for which bytecode compilers exist, including Eiffel, Ada, O’Caml, Scheme, and Prolog. Additionally, one would assume that integrating the techniques with those of, *e.g.*, Bandera and Flex, would be relatively straightforward. Indeed, JPF already works with Bandera for some of its functionality. Finally, the developers note that, unlike with source analyzers, JPF is able to analyze compiled libraries and programs that use libraries.

Bandera Bandera [20] constructs finite-state models from Java source; these models are then model checked using Spin [39], SMV [59], or SAL [25]. Un-

like other work on translating Java to Promela, Bandera focuses heavily on minimizing the size of the generated finite model, mainly through abstraction. First, given the user-supplied property and annotations, it soundly slices the Java source, extracting an abstraction that is relevant to the property. Second, it applies user-supplied abstractions to reduce the cardinality of types; *e.g.*, it could apply a “sign” abstraction to integers or an *ItemInVector* abstraction to a particular object and vector. Third, it produces, *e.g.*, optimized Promela.

Because it requires extensive user-interaction to generate the abstraction, Bandera supplies an elegant GUI. Specifying properties is made intuitive through property templates. Additionally, it provides access to a library of abstraction modules.

The developers made extensibility one of the goals of the project. Thus, adding new analyses and techniques, as well as exploiting the existing functionality, should be relatively easy.

BLAST BLAST (Berkeley Lazy Abstraction Software Verification Tool) [38] verifies safety properties of C programs using predicate abstraction. It has been successfully applied for verifying properties of drivers.

BLAST represents an integration of tools. For example, it parses C via CIL and uses several decision procedures (*e.g.*, Simplify [54] and CVC [60]). Several aspects of BLAST would obviously benefit from greater integration. For example, the user manual [15] cites aliasing as a “major source of complexity in the implementation” — complexity that could have been factored out given a TIP. Additionally, the generation of seed predicates and the extraction of refinement predicates could easily be exported so that BLAST could potentially benefit from other analyses.

SLAM Like BLAST, SLAM [5] verifies C programs via predicate abstraction. The BLAST and SLAM strategies differ; however, SLAM has also been successfully applied to verifying properties of drivers and other real applications.

MAGIC MAGIC (Modular Analysis of Programs in C) [14] relies on predicate abstraction, like BLAST and SLAM. However, it builds SAT formulas that assert weak simulation between a given labeled transition system (LTS) and a C procedure. Additionally, the developers stress compositionality in their method, claiming that this approach allows the tools to spend relatively more time in constructing an abstract model. Targeted software includes implementations of security and network protocols.

CMC CMC (C Model Checker) [52] uses efficient explicit state model checking to find bugs in network protocols implemented in C and C++. Network protocols are suitable applications for analysis, as they are event-driven. CMC exploits the event-driven architecture in several ways. Memory requirements are reduced by ignoring the stack and registers, which are preserved by event handlers. Atomic

transitions are easily defined as the event handlers. Finally, the protocol's interactions with the network are defined by calls to the event handler. Nevertheless, CMC is fully implemented to work with other applications. The user supplies correctness properties beyond the usual set of domain independent properties (*e.g.*, never accessing illegal memory) and assertions. Additionally, the user defines an environment, consisting of stubs for functions like `malloc`, and identifies the initialization functions and event handlers.

Like BLAST [38] and SLAM [5], CMC requires relatively little effort when applied to applications of the same kind. Once correctness properties and an environment are defined for a particular type of network protocol, for example, checking several implementations should be straightforward. However, applying it to a new domain requires some effort.

CBMC CBMC [16] exploits bounded model checking (BMC) to verify correctness of C programs. It translates C code into SAT by unrolling loops a finite number of times, translating the C to SSA (static single assignment) form, and then constructing two bit vector equations, one for the program and one for the property. The negation of the usual implication between program and property is then compiled into SAT. Dynamic arrays, via `malloc`, are handled with uninterpreted functions.

Targeted programs include embedded software and prototype simulations that will eventually be implemented as hardware. Hence, its success relies on the fact that loops in such software can often be soundly unrolled a finite number of times. In the latter class of programs, CBMC checks formal equivalence between the C simulation and a Verilog description of the hardware.

The developers also provide a GUI that presents a debugger-like interface. Counter examples are presented as traces that can be incrementally executed, just like in a debugger. The combination of an automatic analysis and a programmer-friendly interface makes this tool undoubtedly useful for development of its targeted programs.

Spin Spin [39] is an extensively-used explicit-state model checker. Its input language, Promela, is expressive, enabling recent efforts at using Spin to verify software (*e.g.*, JPF1 [37], Bandera [20], JCAT [26]).

SAL At SRI, an ongoing effort aims at integrating finite state tools with their existing infinite state tools. SAL (Symbolic Analysis Laboratory) [25] consists of an expressive language, one that includes higher types, predicate subtypes, datatypes, integers, reals, and more; an API to core functionality; and four model checkers implemented with the core functionality. The “infinite bounded” model checker extends bounded model checking, compiling level k into the language of ICS [33], a combination decision procedure that includes theories for ground linear arithmetic, uninterpreted functions with equality, arrays, and several other theories.

SAL’s language is intended to be an intermediate language, so it is defined in XML. Functionality for parsing and pretty-printing SAL instances support its use. The developers plan to support translators from other modeling languages. Long term, the developers intend to integrate SAL with PVS [55] via specification translation. They also propose a “SAL Tool Bus,” through which other tools and SAL may interact.

FindBugs FindBugs [40] looks for bugs in Java code based on bug patterns. Its core functionality rests on several standard dataflow analyses, as well as class hierarchy analysis. While its bug-finding techniques are simple pattern matches, the authors believe this shallow method is an effective approach, as many bugs in ordinary code arise from simple errors. For example, it is easy to mistakenly compare two `Strings` with the `==` operator instead of calling the `equals()` method — and easy to catch this mistake — but the error can lead to serious bugs in the program. FindBugs has the advantage that it requires low overhead on the part of the programmer: no annotations are needed, and both a GUI and command-line version are available. Additionally, it is possible to obtain it as a plug-in for Eclipse and as a task for the Apache Ant build tool.

PREFix The development of PREFix [13] was motivated by the usability of Purify [41]; however, the developers wanted a static analyzer rather than a dynamic one. PREFix aims to identify many bugs, minimize false errors, and perform with little or no user interaction. Their reported results include statically identifying uninitialized memory; dereferences of NULL, uninitialized, or invalid pointers; and leaked memory, on software as large as Mozilla.

PREFix automatically constructs *models* of functions via symbolic simulation of paths. Models are essentially sets of guarded commands. A path includes assignments, guards for control flow, and descent into function calls. To mitigate effects of the possibly exponential number of paths, especially for root or near-root functions, the user may set an upper limit on the number of paths to explore. When a function is called, its model is used to evaluate its effect. Manually-constructed models are supplied for system calls; however, PREFix can run even without them, which allows it to handle library calls.

Metacompilation Metacompilation (*e.g.*, [36]) proposes a new perspective on finding bugs. Rather than finding bugs through failed proofs (either failed verification conditions or a counterexample produced by a model checker), a metacompiler lets the user specify rules that the source code should obey. In their simplest form, rules are state machines in which guards are based on state and patterns on AST nodes. The metacompiler then applies the rules in a context- and flow-sensitive analysis. Rules need not be sound; indeed, the useful ones are not. Rather, a good rule finds as many bugs as possible without raising too many spurious errors. The technique is enhanced by using weak decision procedures. Some assignments and guards are recorded in a constraint database

during a depth-first exploration of a path; guards are then evaluated based on this constraint database, sometimes ruling out paths.

Extensions are specified via the Metal language; `xgcc` is the analysis tool. Extensions can range from simple state machines to complex analyses with C code. For example, subsequent work introduced statistical analyses to locate bugs [30]. Apparently, the technique is successful enough to support a company (<http://www.coverity.com>).

ESP ESP (Error Detection via Scalable Program Analysis) [24] uses techniques similar to those of metacompilation, except that its analyses are sound. That is, if no errors are reported, then no errors exist. Further, it introduces a new type of dataflow analysis that takes advantage of the states of the property automaton. In path-sensitive analysis, a potentially exponential number of paths are analyzed, sometimes resulting in exponential space requirements. Conversely, standard dataflow analysis is not sensitive to paths, thus, for example, resulting in spurious error reports. *Property simulation* maintains execution state and property state as its dataflow state; only dataflow states with equal property states are merged. In a sense, only property-relevant path-sensitive information is maintained. This new analysis has polynomial time complexity. Using this technique, ESP has verified properties of large programs.

CQual CQual [34] exploits type checking and type inference to verify deeper properties of C programs. Programmers add type *qualifiers* (e.g., `const`, `tainted`, and `sorted`) to the program text in the same way that C/C++ programmers currently use `const`. Type inference is then equivalent to verification. The advantages of the approach, relative to other annotation systems, is that programmers are already comfortable with using types. Specifying deeper properties via qualified types is thus a natural extension. Additionally, because CQual also infers qualifiers, programmers need not qualify every type. For example, adding qualifiers to library entry points may be enough to check certain properties. Finally, type checking and inference is decidable and efficient, so the method scales to large programs.

CQual supports implementing new qualifier-based analyses. The techniques in CQual complement other verification efforts. For example, inferring qualified types may strengthen a subsequent theorem proving effort of deeper properties. Qualified types could also serve as bases for predicate abstraction.

Astrée Astrée (*Analyseur statique de logiciels temps-réel embarqués*) [48] exploits the abstract interpretation framework to verify the nonexistence of runtime errors in a restricted class of C programs. Runtime errors include out-of-bound array accesses, integer division by zero, floating point errors, unwanted integer arithmetic *mod* behavior, casts to smaller types, and user-provided asserts. The tool targets synchronous C programs without dynamic memory allocation, string manipulation, or complex pointer operations. One goal is to produce as

few spurious errors as possible in a realistic amount of time and memory. Astrée achieves impressive results, mainly through a wise choice of inexpensive abstract domains, smart heuristics, and (most likely) a well-written implementation.

ESC ESC (Extended Static Checking) for Modula-3 [27] and Java [45] applies automated theorem proving to find bugs statically. While ESC can be applied without any annotations — to find possible null-dereferences, for example — it is more powerful when the programmer provides method annotations in the form of preconditions, postconditions, and modify statements. For multithreaded programs, the programmer can supply annotations for “locking-level verification” in the form of declarations of which locks (are intended to) protect which variables and which locks can or should be held at program points. Annotations can even declare abstract variables, such as `valid` and `state` variables for the valid/state paradigm. The developers note, though, that annotations are often not required for client programmers; only the interface and implementation of a library need be annotated to catch many errors.

ESC is fundamentally based on generating (based on the annotations) and automatically proving verification conditions. Proofs come via the Simplify [54] decision procedure. More importantly, ESC extracts reasons for failure from failed proofs, thus identifying and explaining potential bugs.

STeP STeP (Stanford Temporal Prover) [9] is an interactive system for proving temporal properties, specified via LTL, of concurrent and real-time systems. The techniques are based on reducing the proof of temporal properties to proofs of first-order verification conditions, many of which the combination decision procedure automatically proves [47]. The system also provides a GUI for constructing and proving verification diagrams [46], algorithms for automatic invariant generation, and a model checker.

PVS PVS (Prototype Verification System) [55] presents a dynamic environment for specifying and verifying models. It includes a powerful combination of decision procedures and theorem proving capabilities, including induction, along with convenient support for defining “tactics.” While PVS is essentially a monolithic and isolated tool, its expressive specification language admits compilation from other research tools, facilitating a loose integration. PVS is not suitable by itself for analyzing large amounts of code, as it is primarily concerned with deep analyses of complex models.

2.2 Tool Foundations

An analysis can be classified as “shallow” or “deep,” depending on the degree to which it considers the semantics of programs — *i.e.*, the behavior that the program text encodes. This loose characterization already suggests the variety of analysis techniques. In this section, we identify three broad categories of analyses based on three perspectives on programs and programming languages.

First, several tools rely on classical static analyses from compiler theory, such as type inference, alias analysis, class hierarchy analysis, and control flow graph construction. Type inference and dataflow analyses exploit the semantics of programming languages to discover facts about programs. Because these analyses were originally proposed and developed for compilers, they are fast — of low polynomial order — and sound. We call this semantic approach the *classical* perspective. CQual extends this perspective by coupling deeper program meaning with language semantics.

Analyses in the second category look at program semantics rather than programming language semantics. A program’s semantics is its reachable state space. We call this view the *state-space* perspective. Analyses following this trend focus on languages of computations, sets of (fair) sequences of program states. Properties of programs are reduced to questions of language inclusion, intersection, emptiness, *etc.*, by transforming properties into languages — more sets of computations. Tools like ESC, STeP, and PVS handle infinite state spaces through theorem proving, mapping a program’s syntax onto its semantics via verification conditions. Tools like Bandera, Java Pathfinder, SLAM, and BLAST handle infinite state spaces by abstracting the state space, which they accomplish by abstracting the program. Theorem proving plays a role in generating and proving the soundness of the abstraction. Then model checking explores the abstract state space. Analyses that take the *state-space* perspective are “deep,” in that they explore the actual meaning of programs — their behavior within an environment. The syntax itself is mere description. These analyses are computationally expensive.

The third category, represented by PREFIX, Metal and xgcc, and ESP, exploits the *language* perspective. Programs represent languages — sets of sequences of program statements — much the way that regular expressions represent regular languages. Of course, this parallel overlooks complicating factors, such as branch guards (which prune infeasible sequences), recursion, and aliasing (so that matching variables syntactically is unsound). Properties are merely specifications of other languages — other sets of sequences of program statements. This perspective is useful to the extent that behavioral properties can be mapped onto such languages. Consider, for example, the multiple-freeing example of [36]. The behavioral property specifies that no allocated memory should be freed more than once. The *language* property roughly specifies a language in which `free` is not applied more than once to any memory allocation point on any path through the program. Unsound containment checking of the analyzed program and the property reveals possible paths that are contained within the language of the program but not the language property. ESP [24] essentially performs a sound containment check so that it verifies (language) properties.

Mapping behavioral properties to pure language properties is the first way of exploiting the language perspective. Adding decision procedures and alias analyses strengthens the mapping between program syntax and program behavior. Online decision procedures allow the analysis to record (a subset of) assignments and branch decisions. The analysis can then identify some infeasible

ble paths via contradictions. From the *language* perspective, regarding branches as nondeterministic choices produces a language — a set of program paths, or sequences of statements — that overapproximates the set of feasible sequences. Soundly applying decision procedures to prune paths refines this overapproximation. Including an alias analysis strengthens the connection with program state. Properties may refer to memory allocation points rather than just to variable names. Finer alias analyses allow finer property checks (fewer false errors).

How can these categories and perspectives be integrated? Beyond integrating tools, we desire to integrate methodologies. Already, loose couplings are present in tools. `xgcc`, for example, uses decision procedures to exclude program paths. But let us consider this question in greater depth via two examples.

First, the analyses that take the classical perspective efficiently produce facts about programs. Such (relatively) easily obtained facts should strengthen abstraction and theorem proving efforts of state-space tools. Consider ESC. In Java, `Collection` classes store `Objects`, so retrieving `Objects` usually requires a dynamic cast. ESC flags such casts as possibly erroneous. However, suppose one supplements the ESC analysis with facts derived from a “`Collection` analysis.” Such an analysis might first apply an alias analysis and class hierarchy analysis (CHA) to form a sound overapproximation to the control-flow graph (CFG). The alias analysis would also allow the tracking of heap allocation points for `Collection` objects. The `Collection` analysis would associate with each such `Collection` object heap allocation point a set of types. If, during the analysis phase, it is discovered that an object of (static) type `X` may be added to a `Collection` object, `X` (and, implicitly, its subtypes) is added to the type set associated with each possible heap allocation point corresponding to the `Collection` object. Finally, this information could be accessed during the proof of verification conditions. Given that dynamic casts are usually correct, this combined analysis would probably eliminate most spurious error reports. Type qualifiers, too, could produce useful facts for theorem proving, as well as predicates for abstraction.

Second, the language, automata, and logic theories that support the state-space perspective should be exploited by the language perspective. In the language perspective, programs are in a sense finite state transition systems, albeit with many transitions and some caveats (if a state is a program statement, then a transition consists of a location guard and a nondeterministic choice of changing the current statement to one of its successors). Recognizing that correspondence, the logical next step is to apply, for example, full LTL/CTL/*etc.*, model checking. Consider, for example, the following two properties, specifying (roughly) mutual exclusion and that no thread will retain forever a lock:

$$(\forall \textit{shared_vars } v)(\exists \textit{mutex } l) \\ \square(\textit{read}(v) \vee \textit{write}(v) \rightarrow \neg \textit{unlock}(l) \mathcal{S} \textit{lock}(l))$$

$$(\forall \textit{mutex } l)\square(\textit{lock}(l) \rightarrow \diamond \textit{unlock}(l))$$

shared_vars and *mutex* are suitably defined macros. These examples have a few motivating subtleties. First, in `xgcc` and ESP, property automata are implicitly

universally quantified. From one view, quantifier-free automata instances are generated and killed dynamically, each tracking, *e.g.*, a variable instance. From another view, a universally quantified automaton tracks sets of, *e.g.*, variable instances. In both views, only universal quantification is permitted, which full model checking reveals as unnecessarily restrictive.

Next, the second property is a progress property. Evaluation of the formula over a program would reveal that some loops must be proved terminating. This requirement is intuitive: a nonterminating loop between the `lock` and `unlock` statements could be dangerous. Thus, applying full model checking opens the door for practical application of recent work in termination of program loops. Property automata of `xgcc` and `ESP` implicitly specify safety properties. Third, properties may take the form of temporal formulas or, *e.g.*, Büchi automata, whichever is more efficient for evaluation. Finally, we note that implementations are independent of property specification: path-sensitive and path-insensitive approaches both have merits and problems. For example, a path-insensitive LTL dataflow analysis, with conjunction for joining, is similar to the *property simulation* of [24], albeit with greater expressiveness. For this level of analysis, an online context-sensitive alias analysis (so that allocation points may be distinguished by call context rather than just program location) is necessary. Additionally, both sound and unsound implementations would have value.

Other opportunities for integration are clear. Many deep analyses, such as deductive verification, invariant generation, and termination analysis, traditionally work over friendly pseudo languages. To some extent, these analyses immediately become useful in the presence of a sound abstraction tool. For example, given a loop of C, an alias context (an overapproximation of what points to what), and a CFG, the abstracter might produce a sound *number abstraction* describing how the number variables of the loop evolve. Such an abstraction is then suitable for (number-based) termination analyses and (number-based) invariant generation. Here, classical analyses boost the effectiveness of state-space analyses. Going another way, generated invariants may aid in pruning paths in depth-first *language* analyses.

Clearly, integration of methodologies is fertile ground for future research. Because researchers with various backgrounds will need to become acquainted with tools and techniques from other domains, it is imperative that we put effort into integrating tools, like parsers, decision procedures, and math packages. In the next few sections, we consider existing tools for integrating implementations via common input representations, parsing data structures, and APIs.

3 Integration via Common Input Representations

Language-based integration allows tools to work together via input representations. As many tools work on various representation languages, efforts at integration focus on translation of one source representation into another. Such translations are difficult, as properties proved by one tool must be translated to the other tool and translated model — and still hold on the translated model.

For hardware and model analysis, such integration via translation is a somewhat *ad hoc* solution to the lack of a standard language. But for software analysis, language-based integration may offer a means of representing multiple languages (*e.g.*, C, C++, and Java) via a common overly expressive intermediate language. Rather than merely translating source, the common framework would export this common representation for direct analysis.

Tool	Language(s)	Purpose	Free
VeriTech	CDL, SMV, Murphi, Spin, STeP, Petri nets	Translator	Yes
MathML	MathML	Common Representation	Yes

Table 2. Summary of language tools.

VeriTech One model for tool integration is based on translation of model representations and specifications. VeriTech [35] is a framework for writing source-to-source compilers. It defines an expressive core design language (CDL) to which source is compiled and from which new source is generated. VeriTech includes translators for SMV [59], Murphi [28], Spin [39], STeP [9], and Petri nets [56]. In some sense, the effort is like SAL; however, VeriTech does not include any analysis tools, as its sole objective is integration of other tools.

The developers of VeriTech have identified several nontrivial issues with source-to-source translation, mainly revolving around differing semantics. As translation often involves introducing new variables and transitions, the compiler records the correspondence between old and new variables and transitions. Furthermore, the developers formalize the *faithfulness* of translations and properties, defining when a property that holds in one source model holds in the other.

The VeriTech model of integration mainly focuses on hardware and model analysis tools, which vary widely in their languages and semantics. Programming languages like C, C++, and Java have more in common; however, semantic differences arising from, *e.g.*, typing, casting, and aliasing, present problems for compilation into a common intermediate language. The VeriTech effort might offer some solutions.

MathML While unrelated to program analysis, the W3C standardization effort, MathML [49], is worth mentioning. MathML is an XML-based language for representing mathematics. It facilitates both the display of mathematics and the maintaining of semantic information, for transfer between applications. Because parsers already exist, MathML is a candidate for saving and communicating the results of mathematically-intense program analyses, such as invariant generation.

4 Integration via Programming Language Parsers

Programming language parsers facilitate efforts at applying one’s research to real code. Parsers parse all or a significant subset of a language into an intermediate representation, such as an AST, three- or four-address code, or Java bytecode. Some tools apply convenient transformations to simplify code and remove language idiosyncrasies, such as moving side-effects from branch guards into the code blocks. Some parsers also allow code transformations, which allow the effects of analyses to propagate to other analyses.

Most parsers have been written with code optimization in mind. For code verification and bug finding, several features are desirable. General support for *annotating code* and *querying other analyses* would be useful. Currently, analyses that gather information useful to other analyses present the information via explicit queries. A complex analysis system will require an architecture for discovering and querying registered analyses. Further, some information is better presented as code annotations, information associated with, *e.g.*, AST nodes. Such information includes inferred type qualifiers and automatically generated invariants. In general, whenever several analyses may supply information of the same type, such as invariants, it would be easier to process annotated code than to query all analyses. Such a structure makes analyses more flexible to changes and additions to the system’s registered analyses. The parsing system should also enforce standards of information presentation to maintain this flexibility. Finally, to support incremental analysis, the parser should be able to write and read annotated code, perhaps as XML.

Several other features would be helpful. The parser could be extensible so that techniques relying on programmer annotated code can read and manipulate the annotations within the parsing framework. Additionally, a parser and its associated decision procedures could share the same expression data structures, more closely integrating reasoning with parsing. No doubt, many other features are possible. Below we give a sampling of existing parsers.

Tool	Language	Representation	Free
CIL	C	AST	Yes
Flex	Java	Quadruples, Bytecode	Yes
Soot	Java Bytecode	Bytecode	Yes
Barat	Java	AST	Yes

Table 3. Summary of parsing tools.

CIL CIL (C Intermediate Language) [53] parses C, producing an AST representation. In the process, it simplifies many unusual C constructs, easing the burden on the client. Moreover, some of its packaged analyses provide extra levels of simplification (*e.g.*, translation to more CFG-like structure and translation to three-address code), which simplifies writing some analyses.

Writing analyses in CIL is remarkably easy. Written in O’Caml, CIL provides an intuitive API for visiting and transforming code. Transforming the code via manipulations of the AST allows analyses to affect each other: executing a set of code transforming analyses in sequence allows each analysis to benefit from the previously executed ones. Additionally, the included points-to analysis offers an example of information transfer via explicit queries. The data structure is first initialized as an analysis; the information is then available to other analyses for querying. Several desirable features are still missing from CIL, including a facility for annotating AST nodes. Annotations would provide a means of recording and transferring information such as invariants, proofs, inferred extended types, and held locks, without resorting to explicit queries. Nevertheless CIL forms an important first step toward a TIP; it certainly provides the most labor-intensive component of a TIP for C.

Flex Flex [32] is a Java compilation and program analysis framework. The Flex intermediate representation is based on quadruples. Many research efforts, focusing largely on program optimization and real-time Java, have used Flex for their implementation.

Soot Soot [61] is a Java compilation and optimization framework. Soot can represent Java bytecode in four ways: as streamlined bytecode (Baf), as three-address code (Jimple), as SSA three-address code (Shimple), and as three-address code suitable for code inspection (Grimp).

Barat Barat [10] parses Java source code into an AST representation. It supports traversals of the AST using the visitor pattern and includes a complete traversal that regenerates the source code. This ability to regenerate the code allows easy implementation of analyses that output annotated code, which can then be used as input to further analyses. Additionally, Barat provides flexibility beyond the visitor pattern by supporting association of attributes with AST nodes, thus providing the means for complex analyses which require caching information about a particular node. A Java parser such as Barat would form an integral part of a TIP for Java.

5 Integration via APIs

Deep analyses require sophisticated reasoning facilities. Extensive research into decision procedures, constraint solving, and optimization has produced a library of foundational tools. Many of these implementations are ready for inclusion in a TIP. Some effort should be expended on standardizing APIs and porting APIs to languages such as C++, Java, and O’Caml. We highlight a few examples of foundational tools in this section. Other examples include tools for linear and semidefinite programming and quantifier elimination for polynomials [19]. Mathematical packages like Mathematica [64], which we describe, provide some of these tools.

Tool	Domain	Language API	Free
Decision Procedures	See text	O’Caml, C/C++, Lisp, <i>etc.</i>	Yes
Chaff	SAT	C/C++	Yes
PPL	Polyhedra	C/C++	Yes
BANE	Set constraints	SML/NJ	Yes
Mathematica	Mathematics	C/C++	No

Table 4. Summary of foundational tools. Note that BANE is actually a framework for specifying constraint-based analyses for C.

Decision Procedures Ground decision procedures based on Nelson-Oppen or Shostak combination include the ones in STeP [9] and PVS [55] and the standalone procedures SVC [6], CVC [60], CVC Lite [23], and ICS [33]. They offer various logics, but usually include at least propositional, ground linear arithmetic for integers and reals, uninterpreted functions with equality, data structures, arrays, and bit vectors. Several offer APIs. ICS provides APIs for O’Caml, C/C++, and Lisp, while CVC Lite offers a C/C++ API. Decision procedures are already essential components of, for example, hardware verification (*e.g.*, [12]), model verification via verification conditions (*e.g.*, STeP [9], PVS [55]), predicate abstraction (*e.g.*, SLAM [5], BLAST [38]), “infinite bounded” model checking (*e.g.*, SAL [25]), and invariant generation. Undoubtedly, the role of decision procedures will only increase in software analysis.

SAT Solvers SAT solvers have improved significantly over the last decade. Chaff [50] is one of the best solvers currently available. Because it is based on the DPLL algorithm, it is complete. However, unlike earlier DPLL-based algorithms, its combination of learning and efficient data structures for Boolean constraint propagation makes it blazingly fast. SAT solvers are already used in many applications for hardware, including model checking, simulation, and fault discovery. Bounded model checking can be applied to software analysis via predicate abstraction. SAT solvers are also an integral part of modern decision procedure packages. Additionally, liveness analysis and reaching definition analysis can easily be encoded into 2SAT, which can be solved in polynomial time, suggesting that SAT may be suitable for higher level analyses, as well.

In a similar vein, BDDs have recently been used for program analyses, specifically, alias analyses [63]. BDDs and SAT solvers will most likely find increasing use as optimized representations of finite sets and relations. The work in [63] includes a tool for manipulating sets at a high level via a BDD implementation of DataLog.

Polyhedra The polyhedra domain is one of the most popular abstract domains used in invariant generation in the abstract interpretation framework. Efficient manipulation of polyhedra is essential for these methods to work. The Parma Polyhedral Library (PPL) [4] is an example of a specialized library that manip-

ulates polyhedra. PPL offers several advantages over other polyhedra libraries. First, memory is managed internally so that the client need not provide a static amount of memory in advance. Second, it implements an API for manipulating not necessarily closed (NNC) polyhedra. This feature makes reasoning about strict inequalities clean. The PPL has a C and C++ API, although porting it to other languages is not difficult. For example, the authors have developed a rudimentary interface for Mathematica [64].

BANE BANE (Berkeley Analysis Engine) [2] is a framework for writing C constraint-based analyses, such as race detection and points-to analysis. CQual [34], for example, is implemented in BANE. Thus, it differs from the purely foundational tools listed above in that it is a framework, rather than a foundational tool. However, reformulating the solver as an API tool for use with other types of analyses seems like a natural step. Analyses are presented as set constraint generators; BANE solves the generated constraints [1]. Many analyses have been written within the BANE framework. The successor to BANE is Banshee [3].

Mathematica Mathematica [64] provides a powerful array of mathematical tools. Additionally, the community of Mathematica users has amassed a remarkable library of tools written in the Mathematica programming language, which includes a subset of ML-like syntax. We have found it invaluable for implementing prototypes of our research.

Mathematica also provides MathLink, an API for interfacing with C/C++ programs. MathLink provides two types of functionality. First, external programs can be integrated within the Mathematica environment, providing, for example, fast implementations of certain functionality. Second, the Mathematica kernel can be called as a tool, making available its vast library of functionality.

6 Tool Integration: A Researcher's Experience

In this section, we provide an account of some of our recent efforts to apply the results of our research to C code, as an example of practical tool integration.

Our first attempt at integration was to apply a termination analysis to C. We took the abstraction approach briefly outlined in Section 2.2. In more detail, we had developed a number-based termination analysis. As with many deep analyses, it is defined in terms of a convenient abstract language, in this case a set of guarded commands. Variables are reals; guards are conjunctions of polynomial inequations; and commands are simultaneous assignments of polynomial expressions to variables. We had already implemented this language of loops and the accompanying analysis in Mathematica [64]. For a prototype, this choice made sense, as the analysis relies heavily on manipulating polynomials.

The challenge, then, was to extract polynomial guarded sets from C source. A sound analysis requires at least the following features:

- *Program slicer*. The slicer extracts a *number abstraction* from a given loop. The remaining variables are number types, while r-values are polynomial expressions (in which no flooring occurs). Guards are replaced by nondeterministic choice where necessary.
- *Control flow graph (CFG)*. Even though the analysis examines one loop at a time, the abstracter must verify that functions called within the loop do not modify variables in the number abstraction. Modified variables are removed and the effects of the removal propagated, unless the modification can be represented in the number abstraction.
- *Alias analysis*. First, the alias analysis is necessary for constructing a CFG because of function pointers. Second, the alias analysis is necessary for determining that variables are not modified via aliases.
- *Invariant generation*. Invariants are necessary for specifying initial conditions. Additionally, the subjects of unsigned casts must be determined to be nonnegative at the cast point.
- *Abstracter*. The abstracter finally generates the set of guarded commands, possibly with an assertion of the initial conditions.

As our goal was merely to test the scalability of the technique itself, we implemented only the slicer and abstracter as a CIL analysis. It produces abstractions that are then analyzed by Mathematica. An attempt to communicate between the CIL analysis and Mathematica in an elegant fashion failed. Mathematica’s MathLink provides an interface for C/C++, but CIL is written in O’Caml. We thus used O’Caml’s facilities for interfacing with C, writing a short MathLink interface for O’Caml. Unfortunately, when we reached the compilation stage, we discovered that the MathLink and O’Caml Unix libraries share three function names, stopping linking in its tracks. A second attempt at elegant integration was to write a Mathematica server with which our application could communicate. We have yet to make it work as fast as simply calling `math`. Our final integration matches our initial “hacky” one. Perhaps there is a lesson in integration in this.

This implementation works well enough for timing information, but the analysis is unsound. We initially explored the other components of a sound implementation described above, but postponed the effort.

Classical alias analyses are intended for compiler optimization, not verification and bug finding. We realized that a verification-oriented alias analysis requires a fine level of detail. For a larger C analysis project, which will exploit the termination analysis (thus justifying our postponing the soundness effort), we have written an alias analysis with the following characteristics:

- *Context sensitivity*. Heap allocation points are identified by location and function call context so that they are not unnecessarily conflated. Moreover, heap allocation points have a *unique* attribute, identifying if they are created within the context of a loop or not. If some call in the call context occurs inside a loop, then the allocation point is not unique.
- *Field sensitivity*. Structures play a large role in C programs. Conflating fields is not an option. Essentially, the analysis provides three “types” of locations: *Basic*, for `ints`, `doubles`, `int *s`, *etc.*; *Structure* of locations; and *Array* of

a location, which conflates all positions into one. Such loose typing is almost insensitive to C's casting features.

- *Support for interprocedural dataflow analysis.* Our larger analysis framework presents clients with an interprocedural dataflow interface (currently *sans* recursion). As aliasing is a crucial part of the framework, the client implementation of the transfer function is presented with the current C statement *and* the alias context.
- *Integration with “interprocedural” aspect of interprocedural analysis.* The alias analysis naturally provides the possible functions for a function pointer dereference, facilitating the CFG exploration.

While this level of sensitivity has higher overhead than is acceptable for code optimization tasks, it should facilitate verification and bug-finding efforts.

One goal in our current implementation work is to separate the analysis code from the language-specific code. Thus, for example, in the alias analysis, approximately 60% of the code is C-independent: it could be applied to Java just as easily (hopefully). However, other modules in the project do not produce as favorable a statistic. Indeed, we will count ourselves lucky if in the end half our code is independent of C. Of course, as we discussed in Section 3, one potential solution to this problem is to create a common intermediate language to which all (interesting) languages can be compiled.

7 Tool Integration: An Industry Perspective

Results of research in analysis techniques do not find their way into industry, at least not easily. Microsoft is adopting predicate abstraction and model checking to verify drivers; however, it took a research lab to effect the transfer. Even reasonably successful methods developed in industry itself are abandoned by that same industry: ESC/Java, developed at DEC/Compaq research lab, did not survive the acquisition by Hewlett Packard; it has now been taken over by academia. There seems to be a serious gap between research results and industry needs. A single portal with integrated tools may reduce the barrier.

What must such a portal provide? Three ingredients are necessary: (1) A GUI-based platform for invoking analyses and presenting results, (2) a back end that integrates the reports of multiple analyses into a single report, and (3) well-defined API's that allow new methods to integrate seamlessly with existing functionality, making the ongoing research process as invisible to the user as possible.

At present, there exists one platform that provides such single-point access: Eclipse [29]. It is an ambitious project with the sole apparent goal of offering the best tools for any task involving code, in one integrated development environment (IDE). It is an open-source environment (<http://www.eclipse.org>) with many hooks to facilitate incorporation of plug-in tools. Tens to hundreds of such plug-ins have been provided, including some static analysis tools. Unfortunately, Eclipse does not provide much beyond single-point access. Plug-ins tend to be loosely coupled, as there is no incentive to integrate the tools. This situation

again places the burden on the user to integrate the methods and their results. Clearly, merely producing Eclipse plug-ins is not the integration answer. Rather, one plug-in should present the results of multiple analyses.

Two research efforts have examined techniques for presenting useful information to users. Kremenek and Engler [44] propose a generic *z-ranking* approach for presenting the output of a single analysis. Essentially, the method is based on the observation that code generally has few bugs; hence, most program locations should satisfy a property, resulting in few error reports. Their approach sorts error reports based on the frequencies of successful and unsuccessful analysis checks. Their experimental results are impressive. Rutar, Almazan, and Foster [58] integrate the error reports of multiple Java checkers to isolate Java classes that probably have many bugs. Ranking at this level allows the user to focus on the buggiest sections of code.

One can imagine a back end integrator that accepts the output of multiple bug finding tools, ranks each tool's report, and ranks code modules based on probable number of bugs. This back end would be tightly coupled with Eclipse's interface so that the user can easily explore error reports by rank (*e.g.*, double-clicking on an error report displays the associated code) or by module (*e.g.*, color-coding indicates the presence of likely bugs).

Of course, not all program analyses operate independently of the user. Banderla [20] provides a useful GUI for exploring, annotating, and slicing Java code, among other tasks. Undoubtedly, some tools will require specialized GUI plug-ins for full analysis power; however, all tools should provide basic output for offline analysis.

7.1 An Industry Assessment of Static Analysis Tools

We had the opportunity to evaluate some static analysis tools for a technology-based company that is exploring the use of static analysis tools in its development and testing cycle. We focused on two tools for Java, FindBugs, a shallow method that looks for bug patterns, and ESC/Java, a deep method that applies theorem proving to analyze the code. We describe the results of our evaluations in the context of two different modes of use in the software development process, as different uses impose different requirements on the tools and their level of integration.

Developer-based use In developer-based use, individual developers are responsible for selecting and applying tools during their own development and testing of software components. In this case, many of the concerns discussed above apply. We found that FindBugs works well in this environment. Not much effort is required to run the tool, and the bug reports are presented in a user-friendly GUI. It does not, however, integrate with other tools, so if other tools are being used, their analysis results will have to be examined separately.

In contrast, ESC requires a large amount of effort by the developer because of the need for annotations in the code. ESC can be run without annotations, but we found that in this case it produces so much output that it ceases to be

useful. As the software developed by the company under consideration is not safety-critical, this amount of overhead cannot be justified.

Automated use Another way to use analysis tools in the development process is to integrate them in the automatic build process: analyses would be triggered by check-in of software modules. No user interaction should be required in this case and reports generated should be ranked in importance, with the option of filtering warnings that are deemed irrelevant at the given stage of development. Neither of the tools considered met these requirements.

The conclusion of our study was that the tools evaluated did not support the needs of this company.

8 Conclusion

In this paper we discussed the gap between theory and practice in software analysis and the roadblocks faced in trying to bridge this gap. The paper is complementary to the previous paper prepared for the CUE initiative [7], which discussed the theory and practice of design methodologies.

It is clear that software analysis will become increasingly important. Market pressure will drive developers to produce ever more sophisticated systems that they do not (fully) understand. Regulatory agencies will not be able to stop this infusion of poorly understood software into all aspects of life. It is imperative that we get *some* handle on these systems. Enforcing sound design methodologies that produce “analyzable code” may work in some restricted environments such as NASA. In general, however, we must expect to be faced with code that is put together under pressure by people with varying levels of skills. To make an impact we must be prepared to apply our methods to the code base produced in this way.

In some sense the software producing world, represented by the body of written software, may be viewed as a natural phenomenon that can be studied like a biological system. Creating the ability to apply analysis methods to this system will create a view that may stimulate entire new research areas, like the invention of the telescope did for astronomy, or the sequencing of the Human Genome for biology. Proper access to the system of study will accelerate research by providing immediate feedback on the effectiveness of methods. In contrast to astronomy and biology, the system under study is an active participant that is itself extremely hungry for metrics that measure performance, reliability, productivity, and complexity. Additionally, it can modify itself through the mere presence of effective analysis methods, resulting — hopefully — in more reliable systems.

As argued in this paper, much theory and many techniques and even prototype tools already exist. The research problem is to apply these methods in an effective way to the software world. It is important that building this bridge to industry is viewed as a legitimate research endeavor, both by funding agencies and by the research community. We believe that the current state of the art

in theoretical software analysis could already provide huge benefits to practical software development, once it is made accessible. It does not, however stop there. The feedback that we receive via this bridge will accelerate research in foundational methods and tools. The present technology push from the side of formal methods will change into a technology pull from the side of industry. A new decision procedure will be celebrated like advances in linear programming are today. Progress will be self-reinforcing: the research will receive more attention, encouraging more funding and more participation.

The potential benefits are sufficiently large and the needs sufficiently urgent that multiple approaches to this problem should be considered and supported, including tool repositories such as the one initiated by SRI, integrated platforms such as Eclipse, and portals that provide web-based analysis services. The question remains: how do we make it happen?

8.1 Topics for Discussion

Following are some of the potential obstacles that we think need to be addressed:

Implementation Generally, forces in theoretical computer science reward publications, not implementations. How can we encourage more implementation effort?

Considerations:

- We are computer scientists. Is implementation of research work a respectable research pursuit?
- Most researchers have more ideas than time. Is it worthwhile to spend our time on implementation?
- MS and Ph.D degrees require largely individual research contributions.
- Authorship often goes to the developer of the idea; implementors are considered secondary.
- Some implementations, such as CIL, require much effort. Yet, once implemented, they significantly decrease the effort of others. How can the original implementor get appropriate credit for his effort?

Ideas:

- A conference for which submissions consist of paper and implementation, such that the implementation meets certain standards, for example, integration in a preset framework.
- A recognized journal of foundational tool descriptions and documentation, especially for tools that help researchers such as parsers and decision procedures.

Funding structure How can funding agencies be convinced to support implementations of foundational tools and integration efforts?

Considerations

- Funding agencies often require “revolutionary new approaches,” explicitly stating in their calls for proposals that extensions to current work will not be considered. This ban immediately excludes any proposal to implement and integrate results of previous research. At best, it encourages ad-hoc implementations that serve merely to demonstrate the feasibility of the methods. As grants usually have durations of up to three years, it is unlikely that significant new methods are developed *and* implemented within one grant.

Ideas

- Funding agencies could instate special implementation grants that are follow-up grants to the regular grants. These grants could have the requirement that implementations be delivered to some repository. Evaluation of proposals for these grants could take into account the potential of the method and the proposer’s past record of delivering implementations.

Working group for TIP Suppose we agree to instigate a “working group” to focus on creating something like a TIP. What are the challenges?

Considerations

- An important aspect of a TIP is agreement on APIs and other standardization. How do we reach agreement? Who should be involved?
- Should implementation of core integration functionality be an academic pursuit?
- Should implementation of a compiler-like front end be an industry pursuit?
- Is academia suited for this endeavor? How do we ensure continuity? Do we want industry to join in the effort, or just to provide input?
- Who funds the core effort?

Ideas

- Integrate the working group with a tool integration conference.
- More than just providing a TIP, the effort maintains a single website as a portal into the software debugging/verification community. The portal provides the integrator and supporting tools for download, a list of *publications* (refereed and accepted tools), and a submission page for new tools.
- The refereed aspect of tool submission provides a source of prestige, which in turn would increase the number of submissions. Moreover, it would enforce robustness, high quality of output, and adherence to agreed upon standards. Such high quality software would in turn promote its use, and then the prestige of getting a tool accepted.

Enforcement or voluntary adoption Should analysis techniques be forced upon industry by government intervention? Or should we have to attract industry?

Considerations

- At present, agencies like the FAA and the FDA impose mostly process-based standards rather than product-based requirements. With increasing system complexity, will process-based standards be sufficient to guarantee public safety? Could agencies such as the FAA and the FDA, and perhaps NIH (and similar agencies in Europe), take a more active role in promoting the development and implementation of analysis methods?
- Government intervention could put too many requirements on the research community.
- Many applications need not be safe. Application of analysis methods for these applications is mostly governed by economic concerns, such as time-to-market and competitive advantage of higher product reliability. In this case, application of analysis methods must be demonstrated to be justifiable.
- Usability is an issue either way.
- What is the role of product liability? Do companies use the lack of efficient analysis methods as an excuse to evade product liability? Or could the threat of product liability promote the use of analysis methods?

References

1. AIKEN, A. Introduction to set constraint-based program analysis. *Science of Computer Programming* (1999).
2. AIKEN, A., FAHNDRICH, M., FOSTER, J. S., AND SU, Z. A toolkit for constructing type- and constraint-based program analyses. In *Types in Compilation* (1998).
3. *Banshee*, 2004. (<http://banshee.sourceforge.net/>).
4. BAGNARA, R., RICCI, E., ZAFFANELLA, E., AND HILL, P. M. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Static Analysis: Proceedings of the 9th International Symposium* (2002).
5. BALL, T., AND RAJAMANI, S. K. The SLAM Toolkit. In *CAV* (2001).
6. BARRETT, C., DILL, D., AND LEVITT, J. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design* (1996).
7. BENEKEN, G., HAMMERSCHALL, U., BROJ, M., CENGARLE, M. V., JÜRJENS, J., PRETSCHNER, A., RUMPE, B., AND SCHOENMAKERS, M. Componentware – state of the art 2003. Tech. rep., Technische Universität München, November 2003. Background Paper for the *Understanding Components* Workshop of the CUE Initiative.
8. BIERE, A., CIMATTI, A., CLARKE, E. M., AND ZHU, Y. Symbolic model checking without BDDs. Tech. rep., Carnegie-Mellon University, 1998. To appear.
9. BJØRNER, N. S., BROWNE, A., COLÓN, M., FINKBEINER, B., MANNA, Z., SIPMA, H. B., AND URIBE, T. E. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design* 16, 3 (June 2000), 227–270.
10. BOKOWSKI, B., AND SPIEGEL, A. Barat - a front-end for java. Tech. Rep. B-98-09, Freie Universität Berlin, Institute of Computer Science, 1998.
11. BURCH, J. R., CLARKE, E. M., McMILLAN, K. L., DILL, D. L., AND HWANG, L. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.* (June 1990), IEEE Computer Society Press, pp. 428–439.
12. BURCH, J. R., AND DILL, D. L. Automatic verification of pipelined microprocessor control. In *Conference on Computer-Aided Verification* (1994).

13. BUSH, W. R., PINCUS, J. D., AND SIELAFF, D. J. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience* 30, 7 (2000), 775–802.
14. CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. Modular verification of software components in c. In *ICSE* (2003).
15. CHLIPALA, A., HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. *BLAST - Infrastructure for C Program Verification*, 2000.
16. CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *TACAS* (2004).
17. CLARKE, E. M., AND EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs* (1981), vol. 131 of *LNCS*, Springer-Verlag, pp. 52–71.
18. CLARKE, E. S., GRUMBERG, O., AND PELED, D. A. *Model Checking*. MIT Press, 2000.
19. COLLINS, G. E. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *GI Conf. Automata Theory and Formal Languages* (1975).
20. CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PĂȘĂREANU, C. S., ROBBY, AND ZHENG, H. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering* (2000).
21. COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. Princ. of Prog. Lang.* (1977), ACM Press, pp. 238–252.
22. COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among the variables of a program. In *5th ACM Symp. Princ. of Prog. Lang.* (Jan. 1978), pp. 84–97.
23. *CVC Lite*, 2004. (<http://chicory.stanford.edu/savannah/projects/cvcl>).
24. DAS, M., LERNER, S., AND SEIGLE, M. ESP: Path-sensitive program verification in polynomial time. In *PLDI* (2002).
25. DE MOURA, L., OWRE, S., RUESS, H., RUSHBY, J., SHANKAR, N., SOREA, M., AND TIWARI, A. SAL 2. To be presented at CAV 2004, 2004.
26. DEMARTINI, C., IOSIF, R., AND SISTO, R. A deadlock detection tool for concurrent java programs. *Software – Practice and Experience* 29, 7 (1999), 577–603.
27. DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. Extended static checking. Tech. Rep. #159, Compaq SRC, Palo Alto, USA, 1998.
28. DILL, D. L., DREXLER, A., HU, A., AND YANG, C. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design* (1992).
29. *Eclipse*, 2004. (<http://www.eclipse.org>).
30. ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP* (2001).
31. ENGLER, D. R., AND MUSUVATHI, M. Static analysis versus software model checking for bug finding. In *VMCAI* (2004).
32. ET AL, M. R. *FLEX Compiler Infrastructure*, 2004. (<http://www.flex-compiler.lcs.mit.edu/Harpoon/>).
33. FILLIÂTRE, J.-C., OWRE, S., RUESS, H., AND SHANKAR, N. ICS: integrated canonizer and solver. In *CAV* (2001).
34. FOSTER, J. S., FAHNDRICH, M., AND AIKEN, A. A theory of type qualifiers. In *SIGPLAN Conference on Programming Language Design and Implementation* (1999).

35. GRUMBERG, O., AND KATZ, S. Faithful translations among models and specifications in veritech.
36. HALLEM, S., CHELF, B., XIE, Y., AND ENGLER, D. A system and language for building system-specific, static analyses. In *PLDI* (2002).
37. HAVELUND, K., AND SKAKKEBAELIG, J. U. Applying model checking in java verification. In *SPIN* (1999).
38. HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. In *POPL* (2002).
39. HOLZMANN, G. J. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.
40. HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Notices* (2004).
41. IBM. *Purify*, 2000. (<http://www-306.ibm.com/software/rational/>).
42. JACKSON, D., AND RINARD, M. Software analysis: a roadmap. In *Proceedings of the conference on The future of Software engineering* (2000), ACM Press, pp. 133–145.
43. KARR, M. Affine relationships among variables of a program. *Acta Inf.* 6 (1976), 133–151.
44. KREMENEK, T., AND ENGLER, D. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS* (2003).
45. LEINO, K. R. M., NELSON, G., AND SAXE, J. B. ESC/Java user’s manual. Tech. rep., Compaq SRC, Palo Alto, USA, 2000.
46. MANNA, Z., BROWNE, A., SIPMA, H. B., AND URIBE, T. E. Visual abstractions for temporal verification. In *Algebraic Methodology and Software Technology (AMAST’98)* (1998).
47. MANNA, Z., AND PNUELI, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
48. MAUBORGNE, L. ASTRÉE: Verification of absence of run-time error. In *Building the Information Society (18th IFIP World Computer Congress)* (2004), The International Federation for Information Processing.
49. *MathML*, 2004. (<http://www.w3.org/Math>).
50. MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC’01)* (2001).
51. MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1998.
52. MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (Dec. 2002).
53. NECULA, G. C., MCPHEAK, S., RAHUL, S. P., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conf. on Compiler Construction* (2002).
54. NELSON, G. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.
55. OWRE, S., RUSHBY, J. M., , AND SHANKAR, N. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)* (1992).
56. *Petri Nets World*, 2004. (<http://www.daimi.au.dk/PetriNets/>).
57. QUEILLE, J., AND SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In *Intl. Symposium on Programming* (1982), M. Dezani-Ciancaglini and U. Montanari, Eds., vol. 137 of *LNCS*, Springer-Verlag, pp. 337–351.
58. RUTAR, N., ALMAZAN, C. B., AND FOSTER, J. S. Comparison of bug finding tools for java. In *ISSRE* (2004).

59. SMV, 2004. (<http://www-2.cs.cmu.edu/~modelcheck/smv.html>).
60. STUMP, A., BARRETT, C., AND DILL, D. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification* (2002).
61. VALLÉE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND Co, P. Soot - a java optimization framework. In *Proceedings of CASCON 1999* (1999).
62. VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. Java pathfinder - second generation of a java model checker, 2000.
63. WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation* (2004).
64. WOLFRAM RESEARCH, INC. *Mathematica, Version 5.0*. Champaign, IL, 2004.