Formal Software Analysis
Emerging Trends in Software Model Checking*

Matthew B. Dwyer
Department of Computer Science and Engineering
University of Nebraska - Lincoln
dwyer@cse.unl.edu

John Hatcliff, Robby
Department of Computing and Information Sciences
Kansas State University
{hatcliff,robby}@cis.ksu.edu

Corina S. Păsăreanu, Willem Visser
NASA Ames Research Center
{pcorina,wvisser}@email.arc.nasa.gov

Abstract

The study of methodologies and techniques to produce correct software has been active for four decades. During this period, researchers have developed and investigated a wide variety of approaches, but techniques based on mathematical modeling of program behavior have been a particular focus since they offer the promise of both finding errors and assuring important program properties. The past fifteen years have seen a marked and accelerating shift towards algorithmic formal reasoning about program behavior - we refer to these as formal software analysis. *In this paper, we define formal software analyses as having several important properties that distinguish them from other forms of software analysis. We describe three foundational formal software analyses, but focus on the adaptation of model checking to reason about software. We review emerging trends in software model checking and identify future directions that promise to significantly improve its cost-effectiveness.*

1. Introduction

Software is everywhere – in devices in our pockets, in the vehicles we travel in, in our banks, hospitals and homes - and its correct operation is essential for our health and well-being. Software systems are growing. Over the past decade, software applications have grown significantly larger and more complex in terms of the sheer size of their descriptions and the capabilities they are designed to implement. Furthermore, many software systems are built to interact with complex, independently operating environments – consisting of physical systems, hardware, and other software – which further complicates their behavior. Despite the best efforts of well-intentioned software developers and researchers, the techniques available to validate software systems are not keeping pace with challenges of modern software systems. The trends driving software complexity will not abate, and we are fast approaching the point at which organizations will deploy software that they do not fully understand. Engineering ethics demand that we develop techniques that will allow software developers to establish strong confidence in the correctness of software systems.

We believe that recent advances in formal software analysis show great promise in addressing this need. These analyses can be seen as the descendants of formal, mathematically-based, software development methodologies that date back over four decades to the work of Floyd [35] and Hoare [45]. While those comprehensive and largely manual methods have failed to win widespread use by practitioners, automated analyses that focus on narrowly defined notions of correctness are finding use in specific software development niches, e.g., device driver development [6]. Our goal in this paper is to: (1) define the class of formal software analyses, (2) capture the state of research in formal software analyses, and (3) identify the critical challenges that must be overcome to realize formal analyses that can be applied by practitioners as part of their normal development practice to produce dependable software.

We begin by defining formal software analyses, then explain and illustrate them with a series of examples and counter-examples (Section 2). We then provide an overview of several classes of formal analysis techniques (Section 3).

*This work was supported in part by the National Science Foundation through awards 0429149, 0444167, and 0541263.

The next two sections (4-5) present emerging approaches and promising future directions in using model checking techniques for path-sensitive and modular program analyses. We believe that continued progress in these areas demands a more mature approach to evaluation and we conclude by describing a set of challenges whose solution would improve the quality and reduce the cost of experimentation with software model checking techniques (Section 6).

2. Formal Software Analysis

An analysis, $A : \mathcal{S} \times \mathcal{P} \rightarrow \{true, false\}$, is a boolean function defined over a set of software systems, $\mathcal{S}$, and specifications, $\mathcal{P}$. Let $S \vdash P$ denote the proof that software $S \in \mathcal{S}$ satisfies specification $P \in \mathcal{P}$. In an ideal world, $\forall S \in \mathcal{S}, P \in \mathcal{P} : A(S,P) = S \vdash P$, but that is impractical, if not undecidable. Practical analyses make sacrifices. An analysis is *sound* if $\forall S \in \mathcal{S}, P \in \mathcal{P} : A(S,P) \Rightarrow S \vdash P$ and it is *complete* if $\forall S \in \mathcal{S}, P \in \mathcal{P} : S \vdash P \Rightarrow A(S,P)$. When a sound analysis produces a positive result this can be taken as *conclusive* evidence that the specification holds on the software; a negative results may indicate the violation of the specification or it may not. When a complete analysis produces a negative result this is *conclusive* evidence that the software has an error with respect to the specification; this is also referred to as *sound error detection*, i.e., a sound analysis for $A(S, \neg P)$. Henceforth, we will only refer to sound and unsound analyses.

> A *formal software analysis* is a mathematically well-founded automated technique for reasoning about the semantics of software with respect to a precise specification of intended behavior for which the sources of unsoundness are defined.

This definition echoes the themes of all formal methods. *Mathematical foundations* permit reasoning not only about the results produced by an analysis but also about the analysis itself, e.g., its correctness. The focus on *semantics* and *specifications* date back to the earliest formal reasoning [35]. The requirement for *automation* is essential if one hopes to scale to the size and complexity of modern software. The most distinctive feature of the definition is its insistence that an analysis be explicit in describing the ways in which it is unsound. This last point merits a deeper discussion.

With a sound analysis, you *know what you know* when you get a positive result. Many analysis techniques, however, make further sacrifices for greater scalability by restricting the $\mathcal{S}$ and $\mathcal{P}$ for which they are sound. This is fine as long as the restrictions are clearly defined, but often they are not. The risk of unspecified unsoundness is that the developer of $S' \notin \mathcal{S}$ may mistakenly interpret $A(S', P)$ as an indication that S' satisfies specification P when it does not. We believe that formal software analyses, in order to be trustworthy, must clearly define when they are sound and when they are not. This requirement holds, in general, only when the formal analysis is showing that a property holds for a system. In contrast, if the analysis has the primary goal of finding a violation to the property then we require sound error detection. Unfortunately, in this latter case, if no error is detected we typically *don't know what we know*.

In the remainder of this section, we sample from the rich program testing and analysis literature to illustrate what we regard as a formal software analysis – and what we do not.

Principled unsoundness Alloy [52] is a modeling notation and a bounded satisfiability-based reasoning engine that has been used to reason about system and algorithm designs by systematically exploring all model instances up to bounds on the number of model elements [72]. These techniques are unsound since a system may require a large bound to be reveal an error; with a small bound no error is reported. The nature of this unsoundness, i.e., the bound, is explicit in the analysis formulation – precisely the kind of qualification we believe is important in a formal software analysis. Another example from the testing field is the Korat tool [10] that performs black-box testing, which gives sound results for sequential programs within the bounds on the size of the input data.

Sound property assurance via over-approximation Abstract interpretations [21], such as ASTRÉE [22], typically over-approximate the set of possible program executions. SLAM [6] explores the behaviors of a program model whose variables have been abstracted to over-approximate the values for each variable which in turn over-approximates the set of possible program executions. If these techniques succeed in demonstrating that all of those executions satisfy the specification of interest, such as freedom from null pointer dereference, then the specification is guaranteed to hold on the program. These are formal software analyses.

Sound error detection via under-approximation An example of a formal software analysis for error detection is CMC [62] and its variants. This analysis performs a stateful simulation of a subset of the program's executions comparing it to a specification of correct behavior. It uses heuristics to drive the search down prefixes of program executions that seem likely to find an error. When it does detect an error it is a real error – CMC performs sound error detection – but if no error is detected one cannot conclude anything about the program's correctness. An exciting trend is emerging at the border of formal software analyses and software testing [8]. Techniques like CMC can be thought of as a powerful kind of specification-based white-box testing where the interaction of specification and program structure is used to guide the search, i.e., select tests on-the-fly.

Static code checking There are an enormous variety of source code analyses [9], some of which we would regard as formal (see Section 3). Many other source code analyses, such as FindBugs [20] and PMD [79], simply scan the source text looking for patterns that are presumed to correlate strongly with errors. While useful, these techniques are no more formal than human inspections since they do not rely on a semantic specification of correctness.

3. Overview of Foundational Techniques

In this section we overview three techniques that underlay many recent advances and promising future advances in formal analysis. Table 3 summarizes the strengths and weaknesses of these techniques. A key feature of many advances in formal analysis is combining techniques so that the strengths of one technique mitigate the weaknesses of another.

3.1. Model Checking

Model checking [18] takes as input a state transition system model M representing a system S's behavior, and a property P to be checked against the system, and then exhaustively explores all paths through M while checking that P is true at each reachable state. In concurrent systems, this exhaustive exploration of paths considers all possible interleavings of concurrent transitions. The state transition system M may be derived from a high-level design model of system behavior, a system's source code, or a system's executable representation (machine code or byte code). The property P may be a temporal property (represented as temporal logic formula, regular expression, or automata) that encodes some required ordering on particular system events, or a state property such as an invariant or assertion. The state-space exploration algorithm may carry out the search using an explicit representation of the transition system, property, and reachable state space, e.g., [48], or a symbolic representation, e.g., [15]. The reachable state space of M must be finite if the analysis is to terminate.

Figure 1 gives an example of a simple algorithm that performs a depth-first search of M's state-space using an explicit representation. In this representation, M is a *guarded transition system* consisting of a set of variables, which for our purposes are coalesced into a single composite *state vector* s, and a set of guarded transitions which atomically test the current state and update the state by executing a transition, α, i.e., $s' = \alpha(s)$. The initial values of program variables are used to define an initial state, s_0. A stack is used to represent the current path in the depth-first search.

On line 4 of Figure 1, *enabled*(s) returns the set of transitions, α, whose guard is true in the given state. Lines 7-9 tests if P has been violated and if so, presents the current stack as a *counterexample trace* representing a system execution path that causes a violation of P, and exits. If P is not been violated, the algorithm checks if s' has been seen before, and if not saves the current state, expands the stack, and continues the exploration. Even though this analysis does not explicitly represent all program paths, it is path-sensitive since it *reasons about paths independently* (summaries of information from multiple paths are not computed) and it *analyzes all paths* (up to the point where a cyclic is detected via *seen*).

```
INIT
 1  seen := {s0}
 2  push(stack, s0)
 3  DFS(s0)

DFS(s)
 4  workSet := enabled(s)
 5  for each α ∈ workSet do
 6    s' := α(s)
 7    if ¬P(s') then
 8      counterexample := stack
 9      exit
10    if s' ∉ seen then
11      seen := seen ∪ {s'}
12      push(stack, s')
13      DFS(s')
14      pop(stack)
end DFS()
```

Figure 1. Depth-first state-space search

The model checking algorithm does not implement a particular scheduling strategy. Instead, at Lines 4-5, the algorithm processes *all* enabled transitions for the current state and imposes *no order* on iterating through these transitions. In effect, this computes an over-approximating abstraction of any scheduling strategy used in any run-time execution environment for system S. This guarantees that the analysis is sound with respect to any environment in which S will be deployed – any interleavings occurring during S's execution will also be covered in the analysis.

3.2. Abstract Interpretation

Abstract interpretation [21] is a lattice-theoretic framework for systematically designing and computing over and under approximations of a system S's behavior. While many different formal analyses including traditional data-flow analysis, weakest pre-condition/strongest post-condition calculi, symbolic execution, and model checking can be cast in the abstract interpretation framework, it is most commonly associated with rigorously designed static data-flow analyses [54].

In the context of a simple data-flow analyses, forming an over-approximating abstract interpretation begins by considering a system S that contains some program variables or memory cells x_i of interest and some lattice of properties $L = (P, \sqsubseteq_L)$ where elements of set P can be thought of as abstract "tokens" that characterize data values that may flow into x_i during S's execution and where $\sqsubseteq_L$ orders the properties P according to the precision/approximation

Model Checking	Abstract Interpretation	Deductive Methods
$\oplus$ The analysis is *path sensitive* – it thus yields very precise information since states/paths are not summarized or approximated in any way. $\oplus$ Facilitates checking of temporal properties and gives greater precision for checking state properties. $\oplus$ Explores all possible interleavings in a concurrent system, and thus is very well-suited for revealing intricate concurrency bugs that are hard to detect using traditional quality assurance methods.	$\oplus$ More scalable than model-checking since information from paths and states are summarized. $\oplus$ Many interesting properties of programs and data can be characterized as property lattices in abstract interpretation. $\oplus$ The framework is more amenable to compositional reasoning than model checking due to focus on summarizing state information. $\oplus$ The framework gives a clear methodology obtaining a provably correct analysis.	$\oplus$ Supports very flexible property languages that are able to specify complex state properties. $\oplus$ They are very amenable to compositional reasoning because the Hoare-logic framework with pre/post-conditions naturally summarizes properties of states at software unit boundaries.
$\ominus$ Exhaustive exploration is costly in the time required to generate reachable states and in the space required to store states. $\ominus$ Achieving practicability in practice usually requires deriving M as an over- or under-approximation of S's true behavior, and which can result in inconclusive verification reports or inconclusive error reports, respectively. $\ominus$ Hard to make the analysis compositional because it is difficult to summarize system path and temporal property information at unit boundaries. This inhibits using unit-wise checking to gain scalability. $\ominus$ Although the analysis works well for temporal control-oriented properties with little data, it tends to do poorly on data intensive properties.	$\ominus$ Properties considered must be fairly simple state-/value-based properties. $\ominus$ Not amenable for supporting temporal properties or properties written in rich specification languages like JML or Spec#. $\ominus$ The property lattice (e.g., in compilers and most other settings) is hard-coded and hidden from the user; few frameworks exist for realistic languages that allow users to define and tailor their own abstract interpretations for user-defined properties. $\ominus$ The over-approximating nature of the framework causes inconclusive error reports to be introduced. $\ominus$ It is difficult to combine different abstractions within the same framework. $\ominus$ Despit the clear methodology, it requires significant mathematical expertise to set up a realistic abstraction.	$\ominus$ Developers often find it difficult to work with a general purpose property language; future directions include designing extensible frameworks that domain-specific primitive properties to be introduced. $\ominus$ Very powerful and flexible properties languages makes it difficult to completely automate associated analyses. Associated methodologies typically rely on users to specify pre/post-conditions and loop invariants, or incorporate some notion of under-approximating (e.g., bounding the number of iterations considered in a loop). $\ominus$ Because the framework is naturally not path sensitive, it is difficult to extend to realistic concurrent systems. $\ominus$ Automation is limited to data domains for which decision procedures exist (e.g., integer linear arithmetic).

Table 1. Strengths and Weaknesses of Foundational Techniques

of information represented (tokens that are more approximating are higher in the lattice). For the simple example of sign/parity analysis of integers, L could be designed as $(\{\bot, \mathsf{neg}, \mathsf{zero}, \mathsf{pos}, \top\}, \bot \sqsubseteq \{\mathsf{neg}, \mathsf{zero}, \mathsf{pos}\} \sqsubseteq \top)$ where $\bot$ represents "no information yet known" and $\top$ represents "don't know – could be neg,zero, or pos". The ordering on the tokens indicates, e.g., that neg is more precise that $\top$ (conversely, $\top$ over-approximates neg).

Once L is chosen, the framework then requires setting up an *abstraction function* $\alpha : \mathcal{P}(D) \rightarrow P$ that associate sets of concrete values from the data domain D (e.g., integers) associated with L to an abstract token in P, and a corresponding *concretization function* $\gamma : P \rightarrow \mathcal{P}(D)$ that maps a token from P to set of concrete values represented by that token. For the sign analysis, example applications of these functions are:

$$\begin{array}{rclrcl} \alpha(\{1,3,7\}) & = & \mathsf{pos} & \gamma(\top) & = & \{\ldots,-1,0,1,\ldots\} \\ \alpha(\{0,2,5\}) & = & \top & \gamma(\mathsf{neg}) & = & \{\ldots,-1\} \\ \alpha(\{-3,-2\}) & = & \mathsf{neg} & \gamma(\mathsf{zero}) & = & \{0\} \\ \alpha(\{\}) & = & \bot & \gamma(\bot) & = & \{\} \end{array}$$

α and γ are required to satisfy the following property, called a *Galois connection*, which ensures that they work together to maintain the approximation ordering as the view shifts between the concrete and abstract information representations: α and γ are monotonic, and for any set of concrete values $V \in \mathcal{P}(D)$ and for any token $p \in P$,

$$V \subseteq \gamma(\alpha(V)) \quad \text{and} \quad p = \alpha(\gamma(p))$$

One then designs an abstract interpreter for language $\mathcal{L}$ in which S is written that processes (and stores in variables) tokens from P instead of concrete values from D. This requires defining an abstract operation and test for each concrete operation and test in $\mathcal{L}$. These definitions must be constructed in such a way as to guarantee that the abstract interpretation computes a *simulation* – i.e., that it maintains the over-approximation relation established by the Galois connection. For example, for a concrete binary operation *op* and its abstract counterpart op_a, the simulation property requires that, for all $v_1, v_2 \in D$ and all $p_1, p_2 \in P$ such that $\alpha(v_i) \sqsubseteq p_i$, if $op(v_1, v_2) = v_3$ then $op_a(p_1, p_2) = p_3$ where $\alpha(v_3) \sqsubseteq p_3$. In short, if the inputs to op_a correctly summarize the inputs to *op*, then the output of op_a must

correctly summarize the output of *op*. Abstract interpretation begins with each variable bound to $\perp$, and then continues by iterating the abstract execution of S until a fixpoint is reached. The simulation property guarantees that the property tokens bound to each variable x_i (and, e.g., each abstract syntax tree node) correctly summarize (over-approximate) the values that could flow into x_i in a concrete execution.

Uses of abstract interpretation are wide-spread: many static analyses that enable compiler optimizations are abstract interpretations [21], abstract interpretation forms the basis of computing over-approximations that make model checking tractable (see Section 4), and it is used directly in property checking frameworks [22].

Regardless of whether or not abstract interpretation is used directly, we feel that it is extremely important for designers of formal analyses to clearly understand the concepts of abstract interpretation since these concepts pervade almost all formal analyses.

3.3. Deductive Methods

Most deductive methods for software analysis trace their origins to Floyd/Hoare Logic [35, 45] which characterizes the behavior of a program statement C using triples of the form

$$\{P\}\, C\, \{Q\}$$

where both P (the *pre-condition*) and Q (the *post-condition*) are formulas that describe properties of states. The triple is *valid* iff for any state s that satisfies P, executing C on s yields a state s' that satisfies Q. A formula P can also be viewed as characterizing a set of states $[\![P]\!]$ – namely, the set of states for which P holds – and a triple can be viewed as summarizing the input/output behavior of C in terms of the set of output states $[\![Q]\!]$ that result from input states in $[\![P]\!]$. A formula Q is weaker than P if $P \Rightarrow Q$ (P entails Q). Intuitively this means that Q is less restrictive and more approximate than P, and Q represents a less precise summary of states – a fact that is perhaps more easily grasped when considering $P \Rightarrow Q$ holds when $[\![P]\!] \subseteq [\![Q]\!]$. Relating back to abstract interpretation, state formulas can be viewed as abstractions that summarize program state information, and they can be arranged in a natural approximation lattice based on the entailment relation as an ordering.

Inference rules for the logic are structured in a *compositional manner* in which the behavior of C is established in terms of the behavior of its components. For example, in the inference rule for a **while**-loop,

$$\frac{\{P \wedge B\}\, C\, \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \textbf{endwhile}\ \{\neg B \wedge P\}}$$

the test B and body C components are used to build up the behavioral description of loop construct itself. Having this compositionality at the heart of the technique leads naturally to compositional reasoning about larger software units such as methods and components. For this rule, P is called the *loop invariant* – since it is a property that should hold before, during, and after the loop executes.

Using the inference rule for statement sequencing (not shown) allows one to chain together statements and formulas to create sequence of statements interspersed with formulas that *summarize* properties of states that flow through the statements on *all execution paths* – illustrating that the technique is not path-sensitive.

$$\begin{array}{l}\{y > 50 \wedge y < 100\}\\ x := y - 20;\\ \{x > 30 \wedge x < 80 \wedge y > 50 \wedge y < 100\}\\ y := x * 2;\\ \{x > 30 \wedge x < 80 \wedge y > 60 \wedge y < 160\}\end{array}$$

Tools associated with Floyd/Hoare Logic methods have traditionally required a high degree of manual intervention to construct appropriate pre/post-conditions. However, significant amounts of automation can be provided using a *weakest precondition operator wp*(C, Q) that, given a statement C and post-condition Q automatically constructs a pre-condition P such that $\{P\}\, C\, \{Q\}$ is valid and P is the weakest formula that can establish Q as a post-condition for C. Recalling the discussion of "weakest" above, the precondition returned by *wp*(C, Q) is the "best" one in the sense that it imposes the fewest restrictions on inputs to C that can guarantee Q to hold. A *wp-calculus* contains rules for computing *wp* such as the rule for assignments

$$wp(x := E, Q) \quad = \quad Q[E/x]$$

i.e., for Q (viewed as a predicate which states a property of x) to hold after the assignment of E to x, Q should hold for the value E before the assignment. A *wp* calculus can go a long way toward automating deductive reasoning for realistic languages, but significant user intervention (e.g., manual construction of loop invariants) or soundness compromises are still required to obtain a full analysis. Its greatest benefit may be in its synergistic combination with other formal analyses which is an emerging trend we discuss in the next sections.

4. Path-sensitive Analysis

The benefit of path-sensitive analyses is that many incorrect or spurious warnings about possible errors in the system will be automatically eliminated (because precise information is computed instead of summaries of information across many paths) – the cost of manually removing such

```
do {
    nPacketsOld=nPackets;
    ...
    if(request){
      ...
      nPackets++;
    }
}while(nPackets!=nPacketsOld);
```

```
do {
    b=true;
    ...
    if(request){
     ...
     b=b?false:*;
    }
}while(!b);
```

Figure 2. Original and abstracted code using predicate `nPackets == nPacketsOld`

spurious warnings can be very high. The cost increases when the flow of control is not straight-forward to determine from the structure of the code, for example, when looking at concurrent programs. However, doing a path-sensitive analysis also comes at a cost in both time and memory usage - in model checking the root cause for this is often called the *state-explosion* problem. In this section, we will look at the two main causes for the state-explosion problem, namely, the influence of the data being manipulated as well as the structure of the control-flow of the system. First we consider how to mitigate the effect of a large data-space (Section 4.1, Taming Data) and then how to reduce the number of control-flow paths that need to be analyzed (Section 4.2, Taming Control).

4.1 Taming Data

4.1.1 Predicate Abstraction

Abstraction of the program data is the primary mechanism to overcome the state-explosion caused by large data domains during model checking. Data abstraction can use either abstract interpretation (with concrete tokens hard-wired to properties) or deductive methods (with properties represented as predicates) to compute a program's over-approximation that has a more tractable state-space than the original, un-abstracted program.

A particularly effective abstraction technique is predicate abstraction [40] – which maps a (potentially infinite state) program into a finite state system via a set of abstraction predicates over the program variables. The basic idea is to replace concrete program variables by boolean variables (usually represented by a bit vector) that evaluate to given abstraction predicates defined over the original variables.

Given a finite set of abstraction predicates $\{p_1, p_2, ...p_n\}$, the abstraction function α maps concrete program states s (i.e. valuations of program variables) to bit vectors $b_1 b_2 ... b_n$ such that $b_i = 1 \Leftrightarrow s \models p_i$. For example, Figure 2 (left) showing a program fragment from a Windows device driver [6]. The program has two integer variables, `nPackets` and `nPacketsOld`, and hence an (effectively) infinite state-space which can not be analyzed exhaustively by an explicit state model checker.

Consider an abstraction predicate encoding the equality relationship between the two variables `nPackets == nPacketsOld`, represented by boolean variable b. The corresponding abstract program is shown in Figure 2 (right). In the abstract program, all the conditions and operations involving `nPackets` and `nPacketsOld` were replaced with corresponding conditions and operations on the boolean variable `b`, such that the behavior of the abstract program is an over-approximation of the behaviors of the original program. The abstracted program has a finite number of states (it has only boolean variables) and therefore it can be analyzed exhaustively with existing techniques for finite state system analysis. Techniques for the automatic generation of abstract programs with predicate abstraction use weakest precondition calculations and are presented in e.g. [6].

Predicate abstraction can introduce over-approximation. In the example, if the value of `b` is false (`nPackets != nPacketsOld`), when the assignment is reached then abstraction non-deterministically chooses either true or false (represented by $\star$) for the new value of `b`. It does this because there is not enough information encoded in the initial value of `b` to precisely increment the value of `nPackets` (specifically, it is not known whether `nPackets==nPacketsOld-1`).

Predicate abstraction techniques form the basis of many popular software model checkers [6, 13, 44] and they have been effective primarily due to the fact that underlying deductive techniques give them the power to reason precisely about relationships among predicates (as mentioned in Section 3). The twin problems of *generality* in deductive techniques (i.e., the property language being so rich that reasoning requires user intervention) and *unsound error reporting due to over-approximation* are both addressed by techniques for iteratively and *automatically refining abstractions* [17] to reduce over-approximation until a precise program abstraction is achieved, i.e., one that produces conclusive positive or negative results. These techniques work well on sequential programs with scalar data, but more work is needed to make them effective on concurrent programs that make significant use of heap-allocated data.

An emerging trend in software model checking proposes *under-approximation* based abstractions for the purpose of falsification [49, 64, 42]. These methods are complementary to the over-approximation based abstraction techniques, which are geared towards proving properties. The work in [49] presents a model checking technique that traverses the concrete transitions of a program and for each explored concrete state it stores an abstract version of that state. The abstract state, computed by a user defined abstraction, is used to determine whether the model checker's search should continue or backtrack (if the abstract state has been visited before). This effectively explores an under-approximation of the analyzed system. Thus, all reported

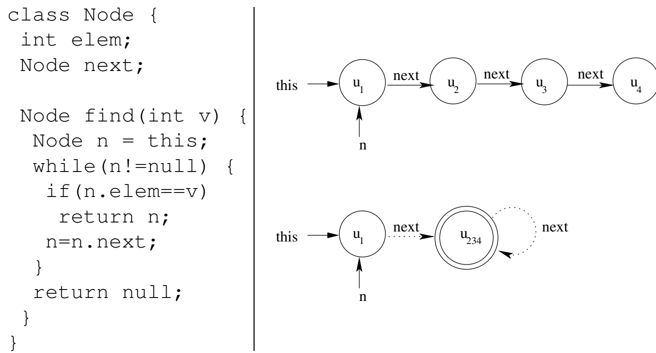

Figure 3. Linked list implementation and graphical representation of the concrete and abstract heaps

```
  int x, y;
1:if (x > y) {
2: x = x + y;
3: y = x - y;
4: x = x - y;
5: if (x > y)
6:  assert(false);
  }
```

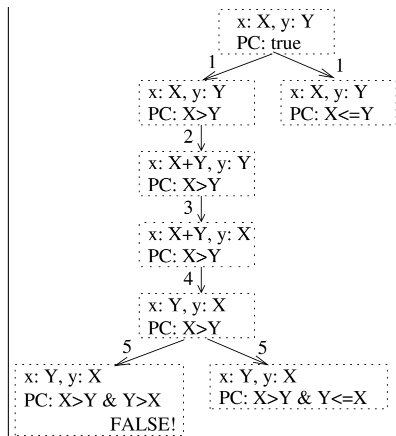

Figure 4. Code that swaps two integers and the corresponding symbolic execution tree

counterexamples correspond to real errors. The technique presented in [64] extends the work of [49] with predicate abstraction and refinement, which results in a fully automatic framework, while Synergy [42] combines "classical" over-approximation predicate abstraction with an analysis of an under-approximation that results in faster convergence of the abstraction refinement algorithm.

4.1.2 Heap Abstraction

The preceding abstractions are limited to handling scalar data. Heap abstractions provide a powerful way to abstractly represent the state of the program heap. Heap cells are represented as nodes in a graph where sets of heap cells that are indistinguishable with respect to the property being analyzed are mapped onto a single *summary* node.

For example consider the code fragment in Figure 3 (left) which contains the `Node` class of a linked list implementation. Method `find` searches for value `v` in the list pointed to by the implicit parameter `this`. Assume that we check that there are no null pointer exceptions for this method. The state space for this method is unbounded, as there is no limit on the length of the input list. Figure 3 (right) shows the *shape graphs* of the program heap representing a concrete list of size 4 (top) and its corresponding abstraction (bottom). Memory cells are nodes in the graphs and edges represent points-to information. The abstraction keeps concrete information about memory cells that are directly pointed to by local variables, e.g., `n`, `this`, and merges all the memory cells that are not pointed to by a program variable into a summary heap node (depicted with a double circle). As the number of local variables is finite, this results in a finite state representation that is amenable to verification. Such heap abstractions have been used in static analysis tools [68], an emerging trend is their application in software model checking [77].

4.1.3 Symbolic Execution

Model checking typically requires a closed system, i.e. a system together with its environment, and a bound on input sizes. Symbolic execution, on the other hand, is a well known technique that enables the analysis of open systems.

The main idea behind symbolic execution [56] is to use *symbolic values*, instead of actual data, as inputs, and to represent the values of program variables as symbolic expressions. As a result, the output values computed by a program are expressed as a function of the input symbolic values.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (`PC`) and a program counter. The path condition is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The program counter defines the next statement to be executed. A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Consider the code fragment in Figure 4, which swaps the values of integer variables `x` and `y`, when `x` is greater than `y` [55]. Figure 4 also shows the corresponding symbolic execution tree. Initially, `PC` is *true* and `x` and `y` have symbolic values `X` and `Y`, respectively. At each branch point, `PC` is updated with assumptions about the inputs, in order to choose between alternative paths. For example, after the execution of the first statement, both `then` and `else` alternatives of the `if` statement are possible, and `PC` is updated accordingly. If the path condition becomes *false*, i.e., there is no set of inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for that path. For example, statement `(6)` is unreachable.

Thus, symbolic execution can be seen as a combination of path sensitive analysis and deductive methods that carry out a form of abstract interpretation (with orderings of properties based on logical implication). Summarizing data properties as path conditions gives an abstract interpretation where properties (which correspond to constraints in this case) can be tailored on the fly, and therefore are more flexible than a static or hardwired collection of properties. The constraints summarize many (possibly an infinite number of) concrete states thus leading to reduced analysis overhead. Symbolic execution has many applications, ranging from test input generation (i.e. one can solve the path conditions to get actual program inputs that can be used to test the program) to proving program correctness.

Symbolic execution traditionally arose in the context of checking sequential programs with a fixed number of integer variables. Recent approaches extend the traditional notion of symbolic execution to handle complex input data structures and concurrency. For example, [55] describes an extension of the JPF model checking tool to perform symbolic execution for Java programs. The approach handles dynamically allocated data, arrays, and concurrency. To handle complex input data structures that contain *uninitialized* fields, the approach in [55] uses *lazy initialization* to assign values to these fields, i.e. it initializes fields when they are first accessed during the method's symbolic execution. An optimized approach is presented in [23] for the Bogor model checking framework.

Performing symbolic execution of looping program may result in an infinite number of symbolic states. Either loop invariants must be supplied to state the desired over-approximation at back-edge points, or one must settle for an under-approximation based on bounding. For example, symbolic execution techniques such as [55] put a limit on the analysis (i.e. limit the search depth, the size of the inputs or the number of loop unrollings), and [23] bounds the length of heap reference chains. Other approaches (such as Symstra [76]) perform symbolic execution and some form of subsumption checking to determine if a symbolic state has been visited before (in which case the generation and exploration of successor symbolic states stops).

4.2 Taming Control

Reducing the number of control flow paths analyzed in a program is crucial in curbing the state-space explosion during model checking. Unlike in the data case, over-approximations of paths plays no role in control reductions. Instead either precise abstractions, i.e. ones where the number of paths are reduced but not at the risk of missing property violations, or under-approximations, where one can miss errors, are used to reduce the paths being analyzed.

4.2.1 Partial Order Reductions

When analyzing concurrent programs the biggest source of state-explosion is due to the large number of possible interleavings between the executions of the (concurrent) threads in the program. Most of these interleavings however involve transitions that are local to a thread, i.e. that has no visible effect outside of the thread it executes in. Partial order reductions [39] (POR) tries to eliminate the interleavings of independent transitions and in doing so produces a precise abstraction of the behavior of the system, since the interleavings that are not analyzed can be shown not to influence the properties being checked. The issue is in how to determine when two transitions are independent.

As described in [18, 39], two transitions are *independent* if the execution of one does not disable the other (and vice versa) (*enabledness* condition) and the execution of both results in the same state regardless of their execution order (*commutativity* condition). *Two transitions n and m are independent at a given state s* if: they commute and executing n does not disable m in the next state.

These conditions define an independence relation between pairs of transitions (statements) that is symmetric and anti-reflexive. In [47], Holzmann and Peled extend this definition of independence with the concept of *global independence*. *Two transitions n and m are globally independent if and only if they are independent in every state where they are simultaneously enabled.* Transitions that only update information in one thread, i.e. that are thread-local, will therefore be globally independent and need not be interleaved.

SPIN uses a simple static analysis to determine such globally independent transitions to enable partial-order reductions. For programs that allow complex aliasing, static analysis is in general not precise enough to detect independent transitions. An emerging trend is to calculate the independence relation on-the-fly during model checking by doing a dynamic escape analysis to determine whether a reference could be visible (i.e. can allow a simultaneous access) in another thread [25]. This technique relies on a traversal of the heap of a program to calculate the visibility of references, which might sound expensive, but in general produces very large state-space reductions and hence reduces overall analysis time. Whereas [25] essentially predicts possible future interactions, in [31] looks at the history of program execution to calculate independence.

Partial order reduction techniques calculate equivalence classes of program paths relative to the property being analyzed and then restrict analysis to a single representative of each class. Equivalence based reductions are powerful and have been used to great effect in other ways in model checking, for example, to identify equivalent heap and thread states [51]. A promising future direction would be to look for domain-specific equivalences that can be efficiently calculated or approximated.

4.2.2 Heuristic Search

An emerging trend is the incorporation of heuristics into path-sensitive analyses. Heuristics are typically designed to calculate a search order that will reach an error state quickly, e.g., [41], or to reach a particular kind of goal state, e.g., one with a short counter-example [27, 73]. Heuristics can also be designed to completely drop transitions from consideration [61].

Heuristics can function in several ways. In traditional heuristic search, one applies a *cost* function to map each enabled transition, α, to a value and then implements line 5 of Figure 1 to iterate over the transitions in cost-order. Cost functions usually calculate a score based on the current state in the exploration or based on the path explored up to the state, for example, Groce and Visser's [41] "demonic" scheduling heuristic scores thread transitions based on the frequency of thread execution along the path. Alternatively one can define heuristics for checking temporal logic properties [28] where the distance (in terms of the length of the control-flow path) to where the variables in a property is updated is used in the heuristic function. The interested reader is referred to [27] for a good introduction to traditional heuristic search used during model checking.

Another heuristic approach is to be selective in storing the program state. This is realized by modifying line 10 of Figure 1 so that the membership test is not performed on the complete state, s, but rather on a projection of the state $\pi(s)$, and projected seen values, $\{\pi(s') \mid \exists s' \in \mathit{seen}\}$. For example, Musuvathi and Engler [61] drop a variable from the state if it has been assigned a large number of distinct values on the path explored. This has the effect of forcing backtracking in the DFS earlier than would happen without this modification.

Heuristics are viewed by many as a promising mechanism for mitigating the combinatorial explosion in the cost of path-sensitive state-space analyses. They have the effect of focusing the search on a portion of the state space.

4.3. Promising Directions

Synergies between analyses: The dominant trend in the area of path-sensitive analysis is to combine multiple approaches to further improve the analysis performance. For example, using abstraction for state matching during the (concrete) execution of a running program combines abstraction, heuristic analysis and dynamic analysis [49, 75]. Further extensions with abstraction refinement [64, 78] and symbolic execution [42] result in increased automation and precision. Other promising techniques combine symbolic execution and dynamic analysis [38, 69] which are made more scalable via compositional techniques [37] (see also Section 5). There is an enormous amount of work needed to scale these techniques to larger programs and to carefully evaluate their cost-effectiveness.

Exploiting new hardware: The new trend towards multi-core processors could lead to huge runtime benefits. Although parallel model checking has been studied extensively (see the Parallel and Distributed Methods in Verification Workshop series), the shared memory architectures of the multi-core processors opens new avenues for algorithmic improvement. The SPIN Model checker has recently been parallelized for a multi-core environment [46].

Beyond scalar data: Significantly more programs could be cost-effectively analyzed if one can scale automated abstraction refinement and symbolic execution techniques beyond scalar data. Specifically, techniques for handling heap data (a promising first step is [59]) and for treating the fine-grain interleavings that arise in threaded programs are needed.

Deep errors in multi-threaded programs: Although model checking has been very successful at finding errors in concurrent programs, it only works well if the errors are "shallow" (i.e. can be found early in the search) or exposing the error is not dependent on (complex) data. One avenue for further exploration is to combine symbolic execution and partial-order reductions to address large concurrent programs. Note that symbolic data will complicate the calculation of independence and thus will make an efficient combination more difficult than one might first appreciate.

Program and domain specific heuristics: Finally, we believe that research into new classes of program and domain-specific heuristics for under-approximation hold great promise. Variations on CMC continue to demonstrate the value of heuristics, but they require significant human insight to identify and configure in the analysis. We need to begin to broaden our understanding of effective heuristics to more than isolated software niches.

Exploiting partial information: Even if a path-sensitive analysis fails to provide a conclusive result, there is still a wealth of information that has been calculated during the analysis. Exploiting this information, for example, to provide feedback to the user on how to proceed with further analysis or give assurances about the behavior of parts of a system, offers a wealth of opportunities.

5. Modular Analysis

Modern software systems usually consist of a large number of components, and make use of software frameworks and libraries which also contribute to their sheer size. Thus, analyzing such systems as a whole becomes quickly intractable. To address this problem, much attention in formal analysis has been given to modular techniques which use a "divide and conquer" approach: properties of a system are decomposed into properties of its components (i.e.

```
  public Interface Iterator{
1: public boolean hasNext();
2: // pre: hasNext();
3: public Object next();
4: public void remove();
  }
```

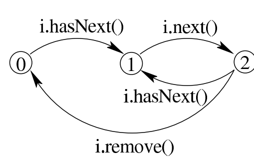

Figure 5. Example illustrating different environment constraints

modules) and each component is then checked separately. Modular techniques enable reasoning about individual components that are developed and maintained independently while ensuring component compatibility during software integration. The individual components are usually smaller than the whole system and therefore easier to analyze.

Moreover, modular techniques address the interaction of a piece of software with its *environment* (i.e., the rest of the system). Thus, they are useful when reasoning about software systems that interact with non-software components (e.g., hardware components for embedded software systems) or native libraries (that cannot be analyzed directly).

When checking components individually, one often needs to incorporate some knowledge about the *behavior* of the environment in which the components are expected to operate correctly. This knowledge can be encoded as preconditions which specify properties of environment state that must be fulfilled *before* executing a component. The design-by-contract paradigm [60] leverages *contracts*, i.e. user annotations in the form of pre-, post- and invariant conditions, to reason about software in a modular way. It is often the case that pre-conditions are not expressive enough to capture the complex interaction between a component and its environment. Assume-guarantee style reasoning [53, 63] uses *assumptions* that describe the continuous interaction between a component and its environment: in a multi-threaded context, assumptions encode the interference between different executing threads, while in the context of checking library code, assumptions (a.k.a. dynamic interfaces) encode the correct sequences of method calls that a client must invoke.

As an example, consider `java.util.Iterator` in Figure 5 (left) which presents a standard interface for generating the elements stored in a Java container (e.g. a set). Figure 5 (right) shows an "assumption" (depicted as a finite state machine) stating that for each instance `i` of a class implementing the `Iterator`, all client applications will call methods in the specified order. Figure 5 (left) also shows a pre-condition, which is part of the "contract" for method `next`, stating that the iteration must have more elements, i.e., `hasNext` returns true. We discuss design-by-contract and assume guarantee reasoning in more detail below.

5.1 Design-by-Contract

Design-by-Contract (DBC) [60] is a software paradigm that enforces contracts (assertions) between computational clients and providers. In addition to specifying pre-/post-conditions and invariants, one also needs to specify frame conditions for operations, i.e., parts of the program state that does not change, usually specified as a set of variables that maybe affected. Modular analysis can be performed to assure that given an input context satisfying the explicitly written conditions the component produces an output that satisfies its behavioral specification. Moreover, DBC makes sure that the clients guarantee the providers' explicit conditions; if so, the clients can assume the providers' specified behaviors.

While the concept of using pre-/post-conditions and invariants as contracts is fairly well-understood, there are several open challenges when applying DBC in object-oriented programs such as expressing contracts involving dynamically created heap objects, exceptions, and behavioral subtyping. The run-time checker described in [14] addresses DBC in the context of Java, by compiling a Java program and its contracts specified using the Java Modeling Language (JML) [58] into bytecode and embedding run-time assertions to check the JML specifications. While this framework supports contract-based checking, it is essentially a (unit) testing technique. The technique in [66] applies model checking, rather than testing, to reason about contracts for Java programs. While its exhaustive nature is better suited for analyzing concurrent programs, it is similar to testing where it only handles closed systems (e.g., units have to be closed by supplying some environments).

Deductive methods have been applied to support DBC verification. For example, LOOP [74] provides JML reasoning using an underlying theorem prover. With sufficient user expertise, LOOP allows strong correctness properties to be established with the highest possible degree of confidence. Advances in LOOP's calculus allow nearly automatic verification of methods with straight-line code performing integer calculations. Reasoning frameworks like ESC/Java [32] are also based on deductive methods but target automatic checking of lightweight properties such as null-dereference and array bounds violations. An example of the emerging trend of synergistic combinations of techniques is the Kiasan framework [23] which uses symbolic execution, model checking, and automated deduction to modularly reason about deep semantic properties of open object-oriented systems.

5.2 Assume-Guarantee Reasoning

Assume guarantee reasoning (AG) checks whether a component M satisfies (guarantees) a temporal property ϕ,

under assumption A about M's environment. Intuitively, A is a property that characterizes all of the environments in which the component M is expected to operate. To complete the proof, one must also show that indeed the actual environment of M (denoted by component M') satisfies A. This proof strategy can also be expressed as the following inference rule:

$$\frac{\begin{array}{l} A \parallel M \models \phi \\ M' \models A \end{array}}{M \parallel M' \models \phi}$$

Here $\models$ stands for some notion of property satisfaction, and "$\parallel$" denotes a composition operator for components (e.g. parallel composition for finite state machines). Note that for this rule to be cost-effective, the assumption A must be *much smaller* than than M', but still reflect M''s behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for M to satisfy ϕ.

Several frameworks have been proposed to support assume guarantee reasoning. For example, the Calvin tool [30] uses assume-guarantee reasoning for the analysis of Java programs, while Mocha [3] supports modular verification of requirements specified in Alternating-time Temporal logic. The practical impact of these previous approaches has been limited because they require non-trivial human input in defining appropriate assumptions.

One approach to automating assume-guarantee reasoning uses the abstraction refinement strategy. In thread-modular abstraction refinement [43], assumptions and guarantees are both refined in an iterative fashion using predicate abstraction techniques. The framework applies to programs that communicate through shared variables. The assumptions in [43] are abstractions of the environment components. The work in [34] also focuses on a shared-memory communication model but does not address notions of abstractions as is done in [43].

An emerging trend in assume-guarantee reasoning is the use of learning algorithms. The first framework to propose this [19] targeted a pair of components M_1 and M_2 a property P. It automatically builds and refines assumptions for one of the components to satisfy P, which it then tries to discharge on the other component. The approach uses learning algorithms to gradually discover appropriate assumptions based on counterexamples obtained from checking the components separately. The framework applies to finite state components with blocking communication and safety properties. Subsequently, several other frameworks that use learning for building assumptions have been developed. For example, a symbolic BDD implementation of a learning based assume guarantee verification framework is presented in [4] while learning is used in [2] to synthesize interfaces for Java classes, and in [70] to check component compatibility after component updates.

5.3. Promising Directions

Modular specifications for heap structures: As mentioned, handling heap objects presents a difficult challenge. In order to address this in a modular fashion, there is a need of intuitive, concise, expressive, and checkable specification forms for reasoning about heap structures. Although ownership types [16] have been introduced to address this, their expressiveness is limited when describing complex, highly-dynamic heap structures. Another promising approach is separation logic [65], an extension of Hoare style logic that allows a concise modular specification of heap structures. Although analysis using separation logic was initially done manually, recent efforts have been made to provide automatic tool supports [7].

Serializability: Modular reasoning about concurrent systems is difficult, since one needs to summarize behaviors that include different threading contexts. A promising approach to address this issue is to detect regions of code that are serializable such that it is safe to reason about such regions sequentially, using DBC for example. Techniques that support this approach use atomicity [33] and the independence in partial order reductions mentioned previously. Recent work in [67] extends JML to handle behavioral specification of concurrent Java programs using the both approaches and it is ripe for exploitation.

Specification inference: One main obstacle with requiring specifications for analysis is the perceived cost of annotation. Yet, without annotation, one cannot apply modular reasoning early during software development. In order to address this cost problem, we believe that specifications should be leveraged not only for analysis, but also for various purposes that directly benefit software developers such as automatically generating test cases from specifications. In addition, one can try to help reduce annotation cost by inferring specifications from existing software artifact. In fact, the automated assume guarantee frameworks discussed in Section 5.2 infer assumptions (interface specifications) automatically, that can be used directly for these purposes. One could also enrich frameworks for dynamic property inference, such as [29], to propose candidate assumptions and interfaces which could then be statically confirmed.

DBC and AG together: We believe that the DBC and AG reasoning styles can be effectively combined. DBC can be used for reasoning about state properties of sequential code, while AG is more appropriate for reasoning about event orderings and concurrency. While earlier frameworks [53, 63] address these in the context of high level models, the real challenge is in providing an effective, automated combination in the context of realistic programming languages and rich properties. As with path-sensitive analysis, we believe that the tight integration of multiple techniques can form a synergistic modular analysis tech-

nique to cost-effectively reason about open systems.

6. Evaluating Path-Sensitive Analyses

The past decade has witnessed significant advances in techniques for precise scalable analysis of software. The literature is filled with papers reporting on proposed techniques. The standards of the formal analysis research community have, implicitly, encouraged authors to provide formal justification for the correctness of each technique and have de-emphasized broad empirical evaluation of the effectiveness of each technique. The rationale has always been that performing such evaluations is very expensive, that requiring such an evaluation would delay the dissemination of new ideas, and that such evaluations will eventually be conducted - but they rarely are conducted. As a consequence, this has shaped a literature where it is extremely difficult to separate the most promising techniques, that researchers should be leveraging, from those that provide only marginal advantage or whose advantages apply only in very restricted settings.

While nearly every paper introducing a new formal analysis technique reports on performance data for *example systems*, there has been very little research reporting the results of carefully performed empirical evaluations. In recent work, we performed a controlled experiment to determine whether the default search order implemented by a path-sensitive formal analysis tool can significantly impact the performance of techniques used to *direct* the analysis to find errors [26] - the results were surprising and should give all researchers on formal analysis pause. We found techniques developed and published (by the authors) over the past five years whose performance were highly-sensitive to variations in the default search order of the analysis tool in which the techniques are implemented. Since search order varies across tools, e.g., JPF, Bogor and SPIN all use different orders, this means that reports of performance advantages for a technique evaluated in the context of JPF, for example, does not assure performance advantage when implemented in Bogor.

In line with the growing trend across many areas of software engineering [71], we believe that the formal software analysis community must begin to *consider the careful evaluation of cost and effectiveness as an essential element of a mature research contribution*.

6.1. Comparative Evaluation

While the formal analysis community has not emphasized the use of controlled-experiments in evaluating emerging techniques, they have adopted two important mechanisms for informal comparative evaluation : *challenge problems* and *competitions*.

Challenge problems have been used for decades across a range of communities. The steam boiler problem is an early and successful example from the formal methods community [1]. A recent CAV/ISSTA challenge problem [12] for path-sensitive formal analysis techniques drew a handful of solutions. Challenge problems provide important qualitative information about relative technique effectiveness and they foster exchange of insights into how techniques operate on single examples. Their weaknesses lies in the fact that they are uncontrolled and solutions are typically not presented in a form that is repeatable - which would allow to confirm and refine the findings on the challenge problem.

Sub-communities that focus on specific foundational techniques, such as boolean satisfiability, have well-established competitions in which tools are applied to a large number of examples [80]. These serve as an important driver of advances in tools and techniques - it is prestigious to win such a competition. They do not, however, emphasize the detailed, and comparative, evaluation of techniques. There may be valuable information that could be gained by a detailed consideration of performance of tools that do not place well in such a competition. Much might be learned by studying the differences between tools and the techniques they implement. For example, two tools might implement some common and some tool-specific techniques. One might be tempted to explain performance advantages solely in terms of differences in tool-specific techniques, but as we found in our study differences in core technique implementations may cause technique performance to vary significantly.

6.2. Promising Directions

Shifting towards a greater emphasis on empirical evaluation presents numerous challenges to the community that must be addressed.

Most researchers formal analysis are trained in the logics, automata, and formal frameworks used for defining analyses and are not trained in designing and conducting empirical studies. This can change. Over the past decade the software testing community has been emphasizing a similar shift. In the early 90s, a series of influential papers explored the cost-effectiveness of control-flow and data-flow based test adequacy criteria using controlled experimental methods [36, 50]. Prior to this work these techniques had only been compared analytically, e.g., their subsumption relationships were shown, and anecdotally. These papers served as models for other testing researchers, established the value of *independent evaluation* (i.e., evaluation by researchers other than the technique inventor), and produced a set of test artifacts that other researchers subsequently leveraged (i.e., the Siemens programs). Within a decade the state of testing research had matured to the

point where papers discussing *how* to perform testing experiments were accepted by the community, e.g., [57, 5, 24]. Formal software analyses are different from testing techniques in important ways that will undoubtedly influence the kinds of experimental methods that ought to be applied and the nature of experiment artifacts that will be needed. The formal software analysis can learn from the lessons of the testing community in embracing the value of experimentation and within a short period of time inculcating it into the fabric of the research community.

Model studies: The community needs a set of model studies to serve as exemplars. We have found that performing studies in path-sensitive formal analysis presents unique challenges from other domains, such as software regression testing. For example, the enormity of program state-spaces and the dearth of knowledge about there structure renders some traditional approaches to sampling used in experiments inapplicable. Experts in formal analysis need to collaborate with experts in empirical studies to define meaningful models for performing such studies.

Technique factor identification: Path-sensitive formal analyses usually involve the combination of many complex techniques for representing and manipulating program state. In our study [26], we identified search order as a factor that must be controlled when evaluating a technique. Our data imply the existence of other factors that can influence the performance of specific techniques. For example, the quality of the independence relation calculated for partial-order reduction controls the number of enabled transitions at a state, as can the introduction of abstraction into analysis, and this exacerbates the effect of search order on analysis cost. A careful and comprehensive accounting of the factors that can influence analysis cost would aid in the design of empirical evaluations.

Artifact development: Many researchers complain that it is difficult to find find a variety of realistic examples on which to apply their techniques. It is easy to suggest that we collect common examples and make them available, but the reality is much more complicated. Different techniques may require very different kinds of examples in order to perform meaningful experiments, e.g., they may target specific classes of errors. Our recent study [26] also indicated that some examples that are in wide-spread use are completely useless for distinguishing the cost-effectiveness of analysis techniques, because they contain errors that are too easy to detect.

Artifact factor identification: Just as the implementation decisions made in realizing an analysis technique may unintentionally influence technique cost, so too may characteristics of programs used in an evaluation. The identification of factors related to programs, specifications, and defects that can influence technique performance would not only aid in the design of empirical evaluations, but allow for creation of a diverse population artifacts to be assembled for the communities use.

It may make sense to decouple the definition of novel analysis techniques from evaluation, but in those cases it is essential that the technique be presented in a way that is amenable to empirical study by other researchers, for example, a complete algorithmic description or an implementation must be made publicly available. Furthermore, the community must value the results of such studies as important regardless of their outcome. This is a significant challenge to our current culture, where we would likely shun a study demonstrating that a previously published result is not cost-effective.

7. Conclusions

Formal software analysis and software model checking are ripe for a new wave of technical development to improve its scalability and applicability to real software systems. The researcher who is fluent in the use of model checking, abstract interpretation, and deductive reasoning techniques is poised to pursue a seemingly endless number of promising avenues for improving the cost and effectiveness of software analysis. We believe, however, that technical progress in this area must be accompanied by careful empirical evaluation and since the shift toward empirical evaluation will likely be gradual, we believe that its importance means that we should begin immediately.

References

[1] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proceedings of POPL'05*, pages 98–109, 2005.

[3] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in Model Checking. In *Proceedings of CAV'98*, pages 521–525, 1998.

[4] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In *Proceedings of CAV'05*, pages 548–562, 2005.

[5] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of ICSE'05*, pages 402–411, 2005.

[6] T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of CAV'01*, pages 260–264, 2001.

[7] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[8] A. Bertolino. Software testing research : Achievements, challenges, dreams. In *[11]*.

[9] D. Binkley. Source code analysis : A road map. In *[11]*.

[10] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of ISSTA'02*, pages 123–133. ACM Press, 2002.

[11] L. Briand and A. Wolf, editors. *Future of Software Engineering 2007*. IEEE-CS Press, 2007.

[12] http://research.microsoft.com/ qadeer/cav-issta.htm.

[13] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *ACM Trans. Computer Systems*, 30(6):388–402, 2004.

[14] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java modeling language. In *Proceedings of the International Conference on Software Engineering Research and Practice*, 2002.

[15] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[16] D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. *Proceedings of OOPSLA'98*, pages 48–64, 1998.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of CAV'00*, pages 154–169, 2000.

[18] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[19] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning Assumptions for Compositional Verification. In *Proceedings of TACAS'03*, pages 331–346, 2003.

[20] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving your software using static analysis to find bugs. In *Companion to Proceedings of OOPSLA'06*, pages 673–674, 2006.

[21] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL'77*, pages 238–252, 1977.

[22] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In *Proceeding of ESOP'05*, pages 21–30, 2005.

[23] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of ASE'06*, pages 157–166, 2006.

[24] H. Do, S. G. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proc. Int'l. Symp. Emp. Softw. Eng.*, pages 60–70, 2004.

[25] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Designs*, 25(2–3):199–240, September–November 2004.

[26] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proceedings of SIGSOFT FSE'06*, 2006.

[27] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology Transfer*, 6(4), 2004.

[28] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF–SPIN. In *Proceedings of SPIN'01*, pages 57–79, 2001.

[29] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of ICSE'00*, 2000.

[30] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-Modular Verification for Shared-Memory Programs. In *Proceedings of ESOP'02*, pages 262–277, 2002.

[31] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of POPL'05*, pages 110–121, 2005.

[32] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI'02*, 2002.

[33] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *Proceedings of PLDI'03*, pages 338–349, 2003.

[34] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of SPIN'03*, pages 213–224, 2003.

[35] R. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Mathematics*, 1967.

[36] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. Software Eng.*, 19(3):202–213, 1993.

[37] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'07*, 2007.

[38] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of PLDI'05*, pages 213–223, 2005.

[39] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer New York, Inc., 1996.

[40] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of CAV'97*, 1997.

[41] A. Groce and W. Visser. Heuristics for model checking java programs. *Int'l. Journal on Software Tools for Tech. Transfer*, 6(4):260–276, 2004.

[42] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A New Algorithm for Property Checking. In *Proceedings of SIGSOFT FSE'06*, 2006.

[43] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. pages 262–274, July 2003.

[44] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of POPL'02*, 2002.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[46] G. Holzmann and D. Bosnacki. Multi-Core Model Checking with SPIN. In *Proceedings of HIPS-TOPMoDRS*, Long Beach, CA, 2007. IEEE.

[47] G. Holzmann and D. Peled. An imporovement in formal verification. In *Proceedings on the Formal Description Techniques Conference*, pages 197–211, 1994.

[48] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–294, May 1997.

[49] G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proceedings of SPIN'04*, 2004.

[50] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of ICSE'94*, pages 191–200, 1994.

[51] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *STTT*, 6(4):302–319, 2004.

[52] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Mass., 2006.

[53] C. B. Jones. Specification and Design of (Parallel) Programs. In *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.

[54] N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. pages 527–636, 1995.

[55] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of TACAS'03*, 2003.

[56] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[57] B. Kitchenham, S. L. Pfleeger, L. Pickard, P. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Software Eng.*, 28(8):721–734, 2002.

[58] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, Oct. 1998.

[59] A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *Proceedings of CAV'05*, pages 519–533, 2005.

[60] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[61] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proceedings of NSDI'04*, 2004.

[62] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of OSDI'02*, 2002.

[63] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, 1984.

[64] C. S. Păsăreanu, R. Pelánek, and W. Visser. Concrete search with abstract matching and refinement. In *Proceedings CAV'05*, pages 52–66, 2005.

[65] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2002.

[66] Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking Strong Specifications Using an Extensible Software Model Checking Framework. In *Proceedings of TACAS'04*, pages 404–420, 2004.

[67] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending sequential specification techniques for modular specification and verification of multi-threaded programs. In *Proceedings of ECOOP'05*, pages 551–576, 2005.

[68] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 2002.

[69] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of ESEC/SIGSOFT FSE'05*, 2005.

[70] N. Sharygina, S. Chaki, E. Clarke, and N. Sinha. Dynamic Component Substitutability Analysis. In *Proceedings of FM'05)*, volume 3582 of *LNCS*, pages 512–528, 2005.

[71] D. Sjœberg, T. Dybå, and M. Jœrgensen. The future of empirical methods in software engineering research. In *[11]*.

[72] M. Taghdiri, R. Seater, and D. Jackson. Lightweight extraction of syntactic specifications. In *Proceedings of SIGSOFT FSE'06*, 2006.

[73] J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in flavers. In *Proceedings of SIGSOFT FSE'04*, pages 201–210, 2004.

[74] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proceedings of TACAS'01*, 2001.

[75] W. Visser, C. Păsăreanu, and R. Pelanek. Test input generation for java containers using state matching. In *Proceedings of ISSTA'06*, 2006.

[76] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of TACAS'05*, 2005.

[77] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *Proceedings of POPL'01*, pages 27–40, 2001.

[78] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: Better together. In *Proceedings of ISSTA'06*, 2006.

[79] `http://pmd.sourceforge.net`.

[80] `http://www.satcompetition.org`.