Software Analysis: A Roadmap

Daniel Jackson & Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology

ABSTRACT

We describe the challenges of software analysis by presenting a series of dichotomies. Each gives a spectrum on which any particular analysis can be placed; together, they give some structure to the space of possible analyses. Our intent is not, however, to provide a survey of existing analyses within this space, but to argue that some regions are more likely to be important in the future than others. Recognizing that our opinions do not represent the consensus of the community, we have tried, for each dichotomy, to make a case for both extremes (or at least identifying the contexts in which one makes more sense) while arguing primarily for one over the other.

We argue that in the future analyses will be *model-driven*, namely centered on abstract models of behaviour; *modular and incremental*, to enable analysis of components, and of systems before completion; and *focused and partial*, rather than uniform, paying closer attention to properties that matter most and to the parts of the software that affect those properties. In support of such analyses, we expect modelling languages to be *global*, with a focus on structural relationships across the system, and *declarative,* and we expect the analyses themselves to make more use of induction than has been fashionable recently. Finally, although we believe that unsound analyses have a bright future, we expect the increasing importance of infrastructural software to bring a renewed credibility to sound, precise and resource-intensive analyses.

INTRODUCTION

In speculating on the future of software analysis, the first question we should ask is whether analysis has a future at all. Perhaps programming languages will progress so far that code will unambiguously express the intentions of the programmer; components will be assembled on a trial-and-error basis; and fault-tolerance mechanisms will make up for any problems that arise at runtime due to errors in components or in their composition.

We think not. The next few decades will see a rapid growth in our software infrastructure, so that eventually we will come to rely on software in almost every interaction with our environment. Transportation, energy distribution, communications, banking and health care will all depend on software. For end-user applications, time to market and feature count may continue to be driving forces but, in the development of our infrastructure, 'getting it right' will matter once again.

Experience tells us that it is near impossible to get a system right by fudging late in the day, so early investment in modelling and analysis will be essential. Moreover, vast amounts of existing code will be reusable only if there are precise and cogent models that describe their guarantees and assumptions. And with less code to write afresh, the proportion of development effort allotted to coding at the expense of design and analysis will fall further.

Code analysis of all kinds will become increasingly common. This trend will be driven by several factors: the continuing need for information about the behaviour of software during all phases of the development; the widespread use of Java, whose type safety and high-level intermediate language make it significantly easier to analyze than languages such as C and C++; and the overall progress in program analysis technology. Moore's Law will also play a role. Static analysis systems that can produce detailed results for huge programs only with resources that seem unreasonable today will become feasible, and the run-time overheads of dynamic analyses will be more palatable.

Until now, research into abstract models and research into program analysis have been largely disjoint enterprises. We see great opportunities for collaboration between the two research communities. On the one hand, we believe that the value of abstract models will be greatly enhanced if a relationship with code can be established, so that a developer who uses a model as a surrogate for code can have some confidence that properties inferred from the model hold also for the code itself. On the other hand, we believe that program analysis can leverage abstract models: by offering a vocabulary in which to report analysis results, by focusing analysis on properties of interest, and by providing induction hypotheses that enable modular analysis.

Our article is divided into three parts. In the first two parts we discuss the analysis of abstract models and code respectively, using a series of dichotomies to structure the space of analyses. In the third part, we expand on the opportunities that lie in the combination of the two—how the value of models can be enhanced by code analysis, and how code analysis can leverage abstract models.

Throughout our article, we construe the term 'analysis' rather narrowly. It means, for us, the extraction of behavioural information from the software, represented as an abstract model or code. Other forms of analysis are important but lie beyond the scope of our comments here. We do not consider, for example, analysis of human factors (whether the software will be usable); analysis of syntactic properties (for example, using dependences to expose undesirable couplings between modules); or the entire raft of compiler analyses that enable optimizations (such as constant propagation and common subexpression elimination).

1 ANALYSIS OF MODELS

Production of lines of code is a common measure of development productivity, and yet code is an expensive liability. Amenability to efficient execution is code's only merit; as a repository of domain knowledge, environmental assumptions, design rationale or even required behaviour, code is a poor second to a carefully constructed model. A model can be more succinct, and can be structured more effectively to separate concerns and articulate key properties. Any opportunity to apply an analysis to a model of the system rather than the system itself should therefore be taken (so long as the model can be shown to be faithful—more on this later). Especially in the early phases of development, when code does not yet exist, analysis of models is a particularly rewarding investment, often exposing problems that can cost much more if not discovered until later.

1.1 Analysis vs. Description

Researchers who work on abstract models can be divided roughly into two camps. There are those, typified by the developers of Z [44, 47], who have focused primarily on the form of the model, and have treated analysis as secondary, favoring succinctness of reasoning over amenability to automation. There are others, typified by the developers of symbolic model checking [4], who have focused on analysis, and paid less attention to the way in which the model is expressed.

A case can be made for either approach. Proponents of analysis over description argue that the main motivation for recording design decisions is to explore their consequences by analysis; designing a language without an analysis in mind is thus like designing a programming language that cannot be compiled. Proponents of description over analysis point out that, in large-scale software development, the key obstacle is not to find the hidden showstopper flaw, but rather to articulate the development steps, with notations that succinctly and precisely express the intentions of the developer (and no more), from requirements through to detailed design.

The truth lies, of course, somewhere between these extremes, and varies according to the problem at hand. John Rushby has argued that Z sacrificed too much analyzability in its design, ruling out the kind of powerful analysis that the PVS theorem prover can provide [40]; PVS does not, however, subscribe to the minimalism of HOL, and its language is rather rich and expressive. Michael Jackson has made a case for the centrality of description in software development, but recognizes the importance of reasoning, especially in the triangle of specifications, environment properties and requirements [25].

The two approaches can also be seen as responses to different problems. In the world of hardware design, which has motivated most of the work on model checking, the problem has been to expose flaws in an existing design. The description is made to enable the analysis, and can be regarded as disposable once the results of the analysis have been obtained. In the world of software—protocol design aside—the problem is usually to construct a plausible description of the domain or required behaviour. Here, the analysis is simply a means to better description.

The aims of analyzability and descriptive power are not incompatible. On the contrary, there seems often to be a synergy between the two. For example, in our Alloy modelling language [23], the treatment of scalars as singleton sets has nice consequences for the user, by making navigation expressions uniform and by sidestepping the partial function problem. At the same time, it makes analysis easier because it reduces the complexity of the underlying calculus, and ensures that every expression can be given a denotation.

Furthermore, a language design that precludes analysis cannot make sense. For this reason, a modelling language must at least have a formal semantics. Having a formal basis does not imply that the language itself should feel formal or have a mathematical flavour; on the contrary, a formal semantics imposes demands of simplicity and uniformity that can make the language far easier to learn and use.

Designing a modelling language without a formal semantics, or at least a strong sense of which features are likely to be semantically troublesome, is therefore a dangerous enterprise. Whether UML [37], a modelling language of unprecedented complexity, can be productively formalized remains to be seen. There are serious efforts underway [2, 36] to retrofit a semantics to the core elements of the language, especially the object modelling component. Until these succeed or the language is radically altered, any form of tool support that goes beyond superficial syntactic analysis and transformation seems unlikely to be feasible.

1.2 Global vs. Local Models

Two kinds of model dominate the abstract description of software. Look in any book describing an object-oriented method, and you'll find object models, which show what objects exist and how they are related, and state transition diagrams, which show states of objects and the transitions between them. At first glance, these two kinds of model are barely related, but to understand their roles, and what analyses might be applied, we need to understand how they differ fundamentally. Surprisingly, the two criteria that readily come to mind do not survive careful scrutiny.

One criterion is that object models are 'static' and transition diagrams are 'dynamic'. Object models are often no more than class diagrams, showing the classes, their fields, and the subclassing relationships between them. A class diagram is a partial representation of the syntactic structure of the code, and in this sense is certainly static in comparison to a transition diagram that

describes runtime states. But a good object model is much more than a class diagram: its nodes represent sets of objects, not classes, and these can classify objects dynamically. The set *Telephone*, for example, might have subsets *Busy* and *Idle*. What are these if not states?

(Incidentally, a good object model is also much *less* than a class diagram, at least when used in the early stages of development. It excludes classes that are artifacts of implementation—such as containers and the kinds of proxy design patterns often use to achieve decoupling—and avoids prematurely allocating state components to objects, eg, by using bidirectional associations between sets rather than by specifying attributes or instance variables).

An object model, therefore, is a state machine; its states are configurations of objects, and its transitions are the changes that result as objects come and go and as relationships are altered. Admittedly, developers often pay less attention to the transitions, despite the notations that most object-oriented methods offer for describing them. And the static/dynamic distinction is not entirely without merit: even if properly elaborated with transitions, the object model describes changes that occur less often. One might view a program's execution as a low and high frequency component superposed: the former described by an object model and the latter by a state transition diagram.

The other common criterion is that object models are 'data intensive' and state transition diagrams are 'control intensive'. This is more helpful, since it highlights the key problem of analyzing object models, namely rich data structures. But the distinction is hard to make precise. All systems have both data and control components, and drawing the boundary between the too is tricky. In a distributed file system, for example, do we regard the caching state of a file as data or control? And if we should describe the set of dirty files as a node in an object model, is that state any more 'data intensive' than the node marked *dirty* in a state transition diagram for a file?

A more productive distinction, it seems to us, sees the two kinds of model as *global* and *local* respectively. The object model describes the global relationships amongst the elements of the system; its states are global configurations of objects, and its transitions are the global changes that result as objects come and go and as relationships are altered. The state transition diagram, on the other hand, describes the states of an individual object and the protocol of interactions amongst objects. In a global model, the state of the system as a whole is directly expressed; in a local model, the global state emerges from the definitions of the local states.

The structuring mechanisms in a global model are specification-oriented: properties of the global state and of global transitions are formed by conjoining subproperties. The structuring mechanisms in a local model are more implementation-oriented: components are combined by communication mechanisms that can often be implemented fairly directly. For these reasons, global models tend to be more useful for the earlier stages of development in which characterizing basic properties and expected global behaviour matter most; local models come into their own when a particular design of the system as a collection of interacting components must be evaluated.

For the object models, their global nature is the key to their utility. Relationships between objects are represented as bidirectional associations and not as pointers, precisely to avoid premature implementation commitments. Especially in conceptual modelling, local notions such as methods and attributes at best make little sense, and at worst undermine the entire enterprise. Object-oriented methods offer constructs—operations in Fusion [7] and joint actions in Catalysis [12], for example—to express state transitions without referring to the internal states of objects and without identifying particular objects as 'targets'.

Global and local models present very different analysis challenges. Local models are usually very operational in flavour, and their complexity arises from concurrent interaction of many small machines. Global models, on the other hand, are usually declarative and partial, and view transitions in terms of a single, sequentialized stream of operations. The state machines that result from local models are *wide* but shallow: there may be many processes arrayed in communication with one another, but each has only a few states—perhaps a thousand at most. The state machines that result from global models, on the other hand, are *deep* but narrow: there is only a single process, but it typically has billions of states.

Almost all work in state machine analysis has focused on local models and the particular complexity that arises from concurrency. Great advances have been made, both in language and in techniques for overcoming the 'state explosion problem', in which the composition of k local machines has a state space that grows exponentially with k. Indeed, the success of model checking, in particular symbolic model checking [4], can largely be credited with saving the reputation of formal methods.

Unfortunately, though, model checking techniques are not readily applicable to global models. The underlying technologies are often optimized for local machines; partial-order reduction, for example, exploits the equivalence of different interleavings of uncoupled events in distinct processes. The languages cannot accommodate global models; to our knowledge, no model checker provides data structures beyond simple records and arrays, or the operators used commonly in global models such as relational image and transitive closure.

1.3 Simulation vs. Checking

Simulation has always been viewed with suspicion by academics. After all, in a space of billions of possibilities, it must surely be better to check a property exhaustively than to examine only a tiny subspace in an ad hoc manner. And indeed, model checking is far more effective than simulation in exposing subtle errors.

Nevertheless, simulation is tremendously useful. When building a model incrementally, it is easy to make mistakes that simulation immediately exposes. Animating a model by generating sample states and transitions makes the experience of model construction far more compelling; to use David Notkin's term, it 'electrifies' the model. By analogy, most programmers have more confidence in their code when they have observed a few sample executions. Perhaps this is a bad thing, but it is nevertheless an almost

universal phenomenon. Our experience is that, in exactly the same way, a model that has been simulated is much less likely to contain egregious flaws.

One of the reasons that simulation has been undervalued is that checking makes better news. Every inventor of a model checking tool is dutybound to exhibit at least one example of a subtle error that was detected in a real system. Recognizing the value of counterexamples is a major contribution of the model checking community. But is analysis really only about finding errors?

The supremacy of error detection over other analysis aims is based on economic assumptions that may not apply to software. A bug in a hardware design can exact a huge cost, so heavy investment in error detection before manufacturing begins is well justified.

But for software, there are few showstopper bugs. Although some flaws can cost a lot to fix if detected late in the development—by a factor of up to one thousand in comparison to early detection, according to a study by Barry Boehm—most subtle errors are far more expensive to detect than to correct. Of the bugs that evade testing, many will never be noticed at all, and those that might have serious economic consequences can be fixed in later releases, or by inducing the hapless customer to download patches.

Why then does early modelling matter? Not so much because it reduces the probability of subtle design flaws, but rather because it prompts an early and serious consideration of fundamental design questions. The lack of a coherent model manifests itself in the code as needless complexity, mismatched interfaces, and a mass of special cases added to overcome the deficiencies of the gross structure. Simulation of the model allows the developer to experiment with different structurings in advance, and investigate their consequences.

We are not so naive as to claim that there are no systems for which correctness matters. On the contrary, we believe that infrastructural software—the kind of software on which we increasingly depend, for energy, telecommunications, medicine, traffic management, banking, and so on—will have to meet higher standards than today's commercial software. Checking will be crucial, not only to ensure that the model is right, but also to establish a correspondence between the code and its abstract model. We discuss some preliminary ideas on this topic in section 3 below.

1.4 Verification vs. Refutation

Checking can rarely be fully automated, since any modelling language that is rich enough to be useful is likely to be undecidable. Some compromise is therefore inevitable. There are, roughly speaking, two options: verification or refutation.

The verification approach has the analysis attempt to find a proof for the given property. If no proof is found, the property may yet hold, but such analyses rarely fail in way that enables the user to determine whether it is the proof strategy or the model itself that is at fault.

The refutation approach has the analysis attempt to refute the given property by finding a counterexample. Such analyses usually employ some form of search in a space that has been artificially bounded. If no counterexample is found, either the bound was inappropriate (that is, counterexamples exist outside the space searched), or the property does in fact hold.

Theorem provers follow the former approach, and model checkers the latter. Model checkers are often described as performing verification, but in practice almost every system involves classes of components and the analysis fixes the number in each class. Some analysis tools (eg, Alcoa [24]) allow the model to be described in a parameterized fashion, and the bound to be imposed as a separate input to the analyzer.

This dichotomy has been shrinking recently. Theorem provers (such as PVS [35] and the engine underlying ESC [11]) are incorporating powerful decision procedures, and there have been numerous model checking schemes involving abstraction and induction to extend the results to unbounded systems. We believe, however, that refutation-based tools will continue to be more attractive to modellers, since they do not demand after the first investment (formalization of the model itself) a second investment (namely in proof) of similar or even greater magnitude. As we have argued before, refutation can be smoothly combined with simulation, and can provide constructive feedback in an interactive fashion [26].

Nevertheless, we accept John Rushby's argument that the choice of analysis is not made in a vacuum [40]. More expensive forms of analysis, such as theorem proving, provide greater degrees of assurance. The choice of analysis must therefore depend on many factors, mostly economic, such as how serious the consequences of missing an error are. Rushby recommends a strategy in which one starts with the cheapest analyses to find the most egregious errors, and then applies successively more costly analyses until the required confidence has been reached or resources have been exhausted.

1.5 Declarative vs. Operational Style

Most languages for global models such as Z [44], VDM [27] and Larch [16] are declarative. In a declarative language, invariants and operations alike are written as logical formulas (constraining individual states and pre/post state pairs respectively). Languages for local models, such as Statecharts [18] and Promela [20], tend to be more operational, defining transitions in a programmatic fashion. Operational descriptions run the risk of implementation bias, but local models rarely suffer from it. The division of the system as a whole into communicating components is of course an intentional design step. The descriptions of the components themselves are not intended to embody implementation decisions, but usually the structure of the local state is simple enough for its updating to require no more than a single assignment or arrow in a diagram (although the microsteps of Statecharts are a concern).

Declarative description brings several benefits to global models. First is *partiality*: the model need not determine the behaviour of the system completely. For reactive systems, this form of nondeterminism is sometimes viewed as a deficiency, but in other models it is often appropriate. In a system with caching, for exam-

ple, we might want to specify the behaviour without choosing a particular replacement policy; the model can say simply that the cache is free to drop entries arbitrarily.

A second, related, benefit is *incrementality*, a consequence of supporting partiality. A declarative model can be understood and evaluated at various stages of completion. One can start by recording only essential properties; discover, by analysis, that these rely on additional properties; and then expand the collection of properties, never making any more choices than necessary.

Third is *separation of concerns*: the model can be organized so that distinct properties of the design (which an implementation would intertwine) are recorded separately. In a text editor, for example, the insertion and removal of characters can be separated from line-breaking concerns, such as hyphenation and justification. Many researchers have advocated the separation of aspects into entirely distinct models [1, 5, 14, 22, 49], and similar ideas have been proposed for code [19, 29].

Declarative features make trouble for analysis. Explicit model checkers, for example, are designed on the assumption that generating the successors of a state is cheap. When there is no recipe for determining the result of a state transition, however, some kind of search is necessary. Symbolic techniques are well suited to declarative models since they already employ an implicit representation of states; the SMV model checker, for example, allows transitions to be specified as formulas.

One nice property of an operation specified declaratively is that backward execution is no harder to implement than forward execution. In using our Alcoa tool [24] to simulate a model, for example, we often ask for executions of an operation that result in a particular condition; the resulting search often proceeds by finding the values of the more tightly constrained post-state variables first, and from these finding appropriate pre-states. Alcoa's ability to simulate the execution of declarative operations belies the traditional position that executability can only be obtained by introducing implementation features. By using an efficient search mechanism, simulation can be obtained without compromising abstractness, and the term 'executable specification' is not an oxymoron.

2 ANALYSIS OF CODE

Current modelling languages have an Achilles heel: the lack of any enforced or checked correspondence with the actual implementation. A designer or engineer who cannot rely on the model to accurately reflect the actual behaviour of the program will be understandably reluctant to use the model to reason about the system's behaviour. In fact, an incorrect model can be worse than no model at all, if it misleads the designer or software engineer into an incorrect understanding of the causes of the program's behaviour. The practical result of the lack of enforced correspondence between the model and the implementation is that the model becomes increasingly less useful during the software development process, to the point where it is essentially discarded as a useful source of information during the later stages of the development and maintainance. This is especially unfortunate in light of the beneficial role that an accurate model could play in these phases of the development process.

The future development of program analyses that guarantee the conformance of the software to the model will increase the utility and therefore the importance of modelling languages during all phases of the program development. A key issue will be the mechanism used to establish the correspondence between the abstractions of the modelling language and the constructs of the programming language. We expect the correspondence to be mostly straightforward, with a single atomic entity in the modelling language often mapping to a single construct in the language. So, for example, the nodes in a standard object model might map to classes in an object-oriented language, and the actions in an event model might map to specific method invocations. There will inevitably be a need for more sophisticated mappings, however, to account for implementation decisions. A single entity in the model might be elaborated into multiple entities in the program; a set in an object model might be mapped to those objects of a class for which a field has a particular value. And there will be a need to indicate those program points at which the invariants specified in the model apply, perhaps by marking a fringe in the procedure call tree.

Program analysis will also be of use in the automatic construction of initial models from the program source. An engineer might use an analysis tool such as Womble [28] to obtain a rough model, and then refine it using a variety of transformations until it matches the abstract model, thus specifying the mapping implicitly. For object models, transformations might include splitting a single node in the model into multiple nodes, eliminating edges that can never actually occur, and annotating potential edges to indicate additional properties of the model such as multiplicity relationships.

Aside from its application to legacy systems, such as approach might also be useful even when the system was initially developed with modern modelling languages. Potential uses in this context include augmenting preexisting models of part of the system to reflect the complete structure of the system, and obtaining coarser or finer models than the initially developed model. It can also be used to obtain updated models that correctly reflect the changes that occur as the software is maintained and modified for other purposes, and to obtain models of components extracted for use in other systems.

We next present several dichotomies in program analysis, and discuss how these dichotomies relate to our view of future trends in program analysis.

2.1 Static vs. Dynamic

Static analyses analyze the program to obtain information that is valid for all possible executions. Dynamic analyses instrument the program to collect information as it runs. The results of a dynamic analysis are typically valid for the run in question, but make no guarantees for other runs. For example, a dynamic analysis for the problem of determining the values of global variables could simply record the values as they are assigned. A static analysis might analyze the program to find all statements that potentially affect the global variables, then analyze the statements to extract infor-

mation about the assigned values.

Dynamic analyses have the advantage that detailed information about a single execution is typically much easier to obtain than comparably detailed information that is valid over all executions. So useful dynamic tools are available for a wider range of problems than static tools. Consider, for example, the problem of detecting data races, which occur when two parallel threads access the same memory location in conflicting ways without synchronization. There are many dynamic tools that will detect data races in a given execution of the program [42, 8], but far fewer static analysis tools that detect potential data races in all possible executions .

Another significant advantage of dynamic tools is the precision of the information that they provide, at least for the execution under consideration. Virtually all static analyses extract properties that are only approximations of the properties that actually hold when the program runs. This imprecision means that a static analysis may provide information that is not accurate enough to be useful. If the static analysis is designed to detect errors (as opposed to simply extracting interesting properties), the approximations may cause the tool to report many false positives. In the worst case, a flood of bogus error reports may render the tool useless for all practical purposes. Because dynamic analyses usually record complete information about the current execution, they do not suffer from these problems. The trade-off, of course, is that the properties extracted from one execution may not hold in all executions.

One important practical issue for the construction of dynamic analysis tools is the level at which the tool manipulates the program. One alternative is to parse the program source, insert the instrumentation, then generate modified source. The unavailability of source code for parts of the system and the difficulty of constructing parsers have led researchers to analyze and instrument the binary code. This approach, of course, makes it difficult for the tool to give the engineer feedback in terms of the original source.

In the future, we expect researchers to develop new dynamic analysis tools with increased power and sophistication. These tools will be used primarily as debugging aids and to help engineers understand the behaviour of large systems. Engineers will communicate with these tools using higher-level paradigms such as database-style queries expressed using models of the program's state or execution. The emergence of standard, high level executable formats such Java bytecode will make it much easier to develop dynamic tools that provide high-level, structured feedback to the engineer. We also expect the ease of analyzing Java byte codes and other high level executable formats to drive a (potentially short) arms race between developers of executable analysis tools and the developers of code obfuscation tools [10]. The goal of code obfuscation tools is to rewrite the shipped version of the code in a way that preserves the semantics of the program but makes it impossible for analysis tools to extract useful information from the shipped version. Without obfuscation, these representations are very easy for competitors to analyze and reverse engineer. Their distribution via the Internet also dramatically increases their availability to unauthorized users and competitors.

In the longer term, we believe that static analysis to discover or verify sophisticated properties of programs will become increasingly viable and important. The major enabling factors for this trend are the wider use of clean languages, increased hardware capabilities, advances in program analysis techniques, and, most importantly, the increasing deployment of critical infrastructural software that must perform as designed.

2.2 Sound vs. Unsound

Sound static analyses produce information that is guaranteed to hold on all program executions; sound dynamic analyses produce information that is guaranteed to hold for the analyzed execution alone. Unsound analyses make no such guarantees. A sound analysis for determining the potential values of global variables might, for example, use pointer analysis to ensure that it correctly models the effect of indirect assignments that take place via pointers to global variables. An unsound analysis might simply scan the program to locate and analyze only assignments that use the global variable directly, by name. Because such an analysis ignores the effect of indirect assignments, it may fail to compute all of the potential values of global variables.

The soundness of an analysis may depend on the characteristics of the programming language, whether or not the program uses certain features, and even the adherence of the programmer to coding standards. Consider, for example, an analysis that uses the declared types of object fields to extract information about the potential referencing relationships between objects. Such an analysis would be sound for type-safe languages such as Java, but unsound for languages such as C or C++. Of course, if the C or C++ program used none of the constructs (such as typecasts) that may violate type safety, the analysis would produce results that are sound for that program.

Why would an engineer be interested in the results of an unsound analysis? The answer is that in many cases, the information from an unsound analysis is correct, and even when incorrect, may provide a useful starting point for further investigation. Unsound analyses are therefore often quite useful for engineers who are faced with the task of understanding and maintaining legacy code.

The most important advantages of unsound analyses, however, are their ease of implementation and efficiency. Consider the two example analyses cited above for extracting the potential values of global variables. Pointer analysis is a complicated interprocedural analysis that requires a sophisticated program analysis infrastructure and a potentially time-consuming analysis of the entire program; locating direct assignments, on the other hand, requires nothing more than a simple linear scan of the program. An unsound analysis may thus be able to analyze programs that are simply beyond the reach of the corresponding sound analysis, and may be implemented with a small fraction of the implementation time and effort required for the sound analysis.

If unsoundness is tolerated, the analysis may avoid even parsing the code; lexical pattern matching to find function calls can be used, for example, to construct a (potentially incomplete) call graph [32]. The advantages of this approach include ease of

implementation, speed of analysis, and easy retargeting to new languages or language dialects. In return, an engineer may well be willing to accept less accurate results.

Finally, unsound analyses can exploit information that is unavailable to sound analyses. An example of this kind of information is information present in comments, or in the exact layout of the code.

For all these reasons, unsound analyses will continue to be important, and we look forward to research advances that increase their power, so that an engineer can obtain crucial information at very low cost. At the same time, we anticipate a renewed interest in sound analyses. Software producers will come under increasing pressure to increase quality and reduce the cost of testing, and will be more willing to invest in the resources required to develop sound analyses. The prevalence of Java will ease the development of sound tools, and its open standards will make the tools that are developed more widely applicable.

2.3 Speed vs. Precision

Static analyses typically exhibit an analyses time versus precision trade off. We illustrate this trade off by discussing two fundamental distinctions that separate many analyses: the distinction between flow-sensitive and flow-insensitive analyses, and the distinction between context-sensitive and context-insensitive analyses.

Flow-sensitive vs. Flow-insensitive

Flow-sensitive analyses take the execution order of the program's statements into account. They normally use some form of iterative dataflow analysis to produce a potentially different analysis result for each program point. Flow-insensitive analyses do not take the execution order of the program's statements into account, and are therefore incapable of extracting any property that depends on this order. They often use some form of type-based or constraint-based analysis to produce a single analysis result that is valid for the entire program.

Researchers have developed flow-insensitive pointer analysis algorithms that have been shown to scale to programs consisting of hundreds of thousands of lines of code [3,46,15]. But because the analyses are flow-insensitive, they cannot, for example, determine if a pointer is initialized before it is used or determine that a pointer has different values in different regions of the program. Both of these properties depend on the order in which the statements of the program execute.

At the other extreme are shape analysis techniques, which are designed to extract detailed information about the referencing relationships between the objects in the program [17,45]. Shape analyses have been designed, for example, to discover that data structure is a tree and not a graph. The analyses are effective even in the presence of destructive updates such as balanced tree rotations and list reversals. Because these detailed properties depend heavily on the statement execution order, the analyses must be flow sensitive and generate an analysis result at each program point. The price for this precision is paid in efficiency. Many of the proposed analyses have not been implemented; those that have been implemented have been tested only on programs consisting of several thousand lines of code. Researchers have also developed flow-sensitive algorithms that do not attempt to extract such detailed information about the object referencing patterns. These analyses have been shown to scale to programs consisting of tens of thousands of lines of code[48].

In our view, engineers will be interested in properties that only flow-sensitive analyses can extract or check. We expect the current efficiency problems to be ameliorated by the introduction of design information as expressed in modelling languages. This information will promote the development of more efficient, modular analyses that will scale to larger programs. In the long run, we expect flow-sensitive analyses, in combination with design information, to play the primary role in the analysis of critical infrastructural software systems. Given their efficiency and relative ease of use, flow-insensitive analyses will be prominent in contexts (such as reverse engineering) where the engineer must rely primarily or exclusively on the code.

Context-sensitive vs. Context-insensitive

Many programming languages provide constructs such as procedures that can be used in different contexts. Roughly speaking, a context-insensitive analysis produces a single result that is used directly in—and is thus approximate enough for—all contexts. A context-sensitive analysis produces a different result for each different analysis context. The two primary approaches are to reanalyze the construct for each new analysis context, or to analyze the construct once (typically in the absence of any information about the contexts in which it will be used) to obtain a single parameterized analysis result that can be specialized for each analysis construct. We call the latter analysis a *compositional* analysis. Some compositional analyses are parameterized to the extent that their results, when specialized for a given context, are as precise as those provided by an analysis that completely reanalyzes the construct for each different context. Others trade off precision in return for a smaller parameterized analysis result or for increased analysis efficiency.

Most flow-sensitive and context-sensitive pointer analysis algorithms use some form of parameterization to avoid excessive procedure reanalysis, but will reanalyze procedures if the aliasing relationships between the parameters are different[13,48]. Others are truly compositional in that they analyze each procedure once (in the absence of recursion)[43,34,9,41]. The context-sensitive algorithms that reanalyze procedures have exponential worst case complexity, although this behaviour has not been observed in practice. The compositional analyses that produce comparably precise results also have exponential worst case complexity. The context-insensitive analyses typically have polynomial worst case complexity, and researchers have developed less precise compositional analyses with polynomial worst case complexity.

In our view, context sensitivity is essential for analyzing modern programs in which abstractions (such as abstract datatypes and procedures) are pervasive. Future research in the field will therefore, in our opinion, focus on context-sensitive analyses. Given the potential analysis time and space implications of reanalyzing abstractions such as procedures, a focus on composition-

al analyses is likely. It may turn out, however, that engineers do not, in practice, use abstractions in significantly different contexts. There is some experimental evidence indicating that this may be the case for procedures[39]. We therefore expect to see future research whose goal is to determine, for different analysis problems, the extent to which context sensitivity matters.

2.4 Multithreaded vs. Singlethreaded

Most classic program analysis techniques were developed for optimizing compilers for sequential languages such as Fortran. The goal was to produce efficient code for a single sequential machine. In the future, however, programs will increasingly use explicitly parallel constructs, both as a program structuring mechanism and for performance reasons. Web servers, for example, often use multiple threads to respond quickly to multiple clients. In a multithreaded program, of course, the instruction streams of the multiple threads may interleave in many different ways. The classical dataflow analysis algorithms were simply not designed for this model of computation. A straightforward adaptation that analyzes all potential interleavings fails because of exponential blowup in the number of analyzed paths. Researchers have already developed efficient extensions of dataflow analysis to multithreaded programs for restricted analysis problems or restricted models of multithreaded computation. The key extensions involve a variety of techniques that represent the potential interactions between threads in ways that allow the analysis to efficiently compute their effect [30,38,41]. An alternative is to use flow-insensitive analyses, which trivially model all the interleavings because they are insensitive to the order in which the statements execute. But in our view, such analyses will, by themselves, provide results that are not precise enough to discover or verify properties that are of interest to many engineers.

Given the importance of threads in modern languages, we believe that the analysis of multithreaded programs will become an active area of research. We also believe that modelling languages will play an important role in this field, and anticipate the development of languages designed to express important properties of multithreaded programs such as synchronization policies and sharing properties of objects and data.

2.5 Distributed vs. Localized

Multiple threads of control are used to express conceptually concurrent activities with potentially fine-grain interactions. The threads typically execute on the same machine, although the machine may have many processors. Another important source of concurrency exists between distributed components executing on separate machines. These components are often developed using middleware packages such as CORBA and DCOM. One of the goals of these packages is to provide an abstraction boundary that decouples the different components.

We believe a narrow, minimal interface is appropriate and even necessary for systems in which components—such as Web servers and http clients—are developed independently and deployed and administered by different organizations. In this context, issues of security and trust argue for strong boundaries and minimal interactions. But there is another kind of distributed system in which components are more strongly coupled. The prevalence of embedded devices will make such systems more common. In such a system, components typically have complicated interactions, and must be designed in close cooperation if the system as a whole is to operate reliably. We therefore anticipate the development of code analyses that are designed for programs expressed as interacting distributed components, and will report results for the behaviour of the system as a whole.

3 MODEL-DRIVEN CODE ANALYSIS

A particularly exciting prospect, we believe, is the exploitation of abstract models in the static analysis of code. There is an intriguing correspondence between object models—advocated as the central design representation by almost all object-oriented methods—and the shape graphs produced by many static analyses. Both are abstractions of the runtime heap structure, and can be viewed as representations of global, structural invariants.

We see great opportunities for synergy in this correspondence. Static analysis can help modelling; the value of models will be bolstered greatly if their correspondence to code can be firmly established. Models can help static analysis too. In this section, we explain how object models might be used to focus and amplify the power of static analyses, and to make the results more useful to the developer.

There is a third crucial element in this approach beyond the model and the analysis: a mapping to connect the two. We believe that, although a tool might suggest possible mappings, it will be essential for the mapping to be provided by the developer. We expect that something along the lines of Aspect's abstraction functions [21] or Leino's dependences [31] might do the trick, although viewed globally rather than within a module in the style of reflexion models [33].

3.1 Modular Analysis by Induction

Our perspective on program analysis algorithms is that they discover invariants that the program preserves. This sheds some light on some of the fundamental difficulties of analysis. To discover an invariant, the analysis must analyze all of the parts of the program that may affect that invariant. In practice, this has meant that most analyses that extract the kind of deep program information (such as object referencing information) that engineers find useful have had to analyze the entire program. This lack of modularity significantly reduces the utility of the analysis. Many projects integrate components from different organizations and companies, and there may be no point at which all components are brought together as coherent whole in an analyzable form. And of course in early stages of development, many components do not even exist.

An alternative is to obtain the invariants from another source, typically the engineer, then use a modular analysis to verify that the program preserves the invariants. Instead of analyzing the entire program as a monolithic unit, the modular analysis would

analyze the program at the granularity of components. It would start the analysis of each component by assuming that the invariants hold before the component executes. It would then analyze the component in the presence of these properties, verifying that the component preserves the invariants.

Program verification systems, which require the engineer to provide invariants (typically expressed in formal logic) that completely characterize the behaviour of the program, are the classic example of this approach. In principle, the systems have a very fine granularity—each loop in the program is a component. Experience with these systems has shown the combination of formal logic, complete specification, and the fine component granularity to be too burdensome for all except the most safety-critical programs. We expect that in the future, partial specification will become an increasingly popular alternative. These properties will be checked in a modular way. The Extended Static Checker (ESC) from Compaq SRC is a good example of such a system [11]. ESC allows programmers to specify invariants for each class, and verifies that each method in the class preserves the invariants.

Aside from these advantages, modularity also allows the analyses to scale to much larger programs. The cost of analyzing multiple components increases linearly with the number of components, rather than quadratically in the program size or worse as is often the case for whole-program analyses.

Modelling languages will become a popular way for engineers to specify the properties required for modular analyses. Instead of specifying the relevant properties in a general-purpose logical language, engineers will instead use special-purpose modelling languages that are optimized for expressing specific aspects of the overall design of the program such as the referencing relationships between objects or the program's sequence of externally visible actions.

Object models, for example, provide an appropriate means for expressing object referencing invariants. These properties could be checked by a sophisticated pointer analysis algorithm. Event sequence models provide an appropriate means for expressing properties of the sequence of externally observable actions that a program performs. These properties could be checked by a sophisticated interprocedural control flow analysis.

3.2 Giving Engineers Control

Modelling languages will also help to address a major weakness in current program analysis approaches. Most analyses are unfocused, and take place in the absence of any indication of which properties are of interest to the consumer of the analysis information. The analysis results produced by the efficient algorithms are often so imprecise that they may be of little use to the engineer. But the precise analyses may not scale to the required program size, leaving the engineer with no viable analysis alternative. Engineers need different degrees of precision in different situations, at different points in the program, and for different data structures. Applying a single analysis uniformly across the entire program is therefore counterproductive. Instead researchers will move in the direction of analyses that accept direction from the engineer via a modelling language regarding the required level of precision. Such an analysis would exploit the engineer's input to increase precision where required, and revert to more efficient but less precise strategies elsewhere.

Communicating the analysis results to the engineer and allowing the engineer to browse or search the analysis results is another area in which modelling languages can help. In general, there is a need to translate the analysis results from the internal data structures generated by the analysis tools into a representation that the engineer can readily understand. Modelling languages provide an effective medium for this communication: their abstractions are designed for human use, and they often have convenient graphical representations. Representing the results of deep static analysis visually is a hard challenge in the absence of a mapping to an abstract model, or a focus on a particular task (such as restructuring, for which Griswold and his colleagues have produced impressive results [6]).

Although we have focused in this section on static analysis, we also believe that modelling languages will play a similarly important role for dynamic analyses. The advantages are similar in both situations: an intuitive language that the designer can use to communicate with the analysis tool and a sharpening of the analysis focus with the concomitant increase in performance and relevance of the extracted analysis information.

CONCLUSIONS

We see software analysis coming full circle. In the last decade, an appreciation for cost-effectiveness has caused researchers to identify more carefully the information engineers require, and to find the most effective means of obtaining it. Precise and sound analyses have fallen out of fashion, since they have tended to scale poorly and to exact a high price for questionable benefits.

In the future, several factors may bring such analyses to the fore again: the demand for reliable software; the availability of vast computational resources; and, perhaps most significantly, the exploitation of abstract models, to focus analysis effort where the payoff is greatest, and to enable modular reasoning on a large scale.

REFERENCES

[1] M. Ainsworth, A.H. Cruickshank, L.J. Groves, and P. J. L. Wallis. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, February 1994.

[2] A. S. Evans and A.N.Clark. Foundations of the unified modeling language. In *2nd Northern Formal Methods Workshop*, Ilkley, Electronic Workshops in Computing. Springer-Verlag, 1998.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, Vol. 98, No. 2, June 1992.

[5] E. Boiten, J. Derrick, H. Bowman, and M.Steen. Consistency and refinement for partial specification in Z. In M.-C.

Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit of Formal Methods, Third International Symposium of Formal Methods Europe*, Volume 1051 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, March 1996.

[6] R. W. Bowdidge, W. G. Griswold. Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization. *ACM Transactions on Software Engineering and Methodology,* April 1998.

[7] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[8] G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998.

[9] R. Chatterjee, B. Ryder and W. Landi. Relevant Context Inference. *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1999.

[10] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient and stealthy opaque constructs. *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1998. ACM, New York.

[11] D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. *Extended static checking*. Technical Report 159, Compaq Systems Research Center, 1998.

[12] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks With UML : The Catalysis Approach*. Addison-Wesley, 1998.

[13] M. Emami, R. Ghiya and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[14] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal on Software Engineering and Knowledge Engineering*. Special issue on *Trends and Research Directions in Software Engineering Environments*, 2(1):31–58, March 1992.

[15] M. Fahndrich, J Foster, Z. Su and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[16] John V. Guttag and James J. Horning with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[17] R. Ghiya and L. Hendren. Is is a tree, a DAG or a cyclic graph? A shape analysis for heap-directed pointers in C. *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 1–15, January 1996.

[18] David Harel. On visual formalisms. *Communications of the ACM*, Vol. 31, No. 5, pp. 514-530, 1988.

[19] William Harrison and Harold Ossher. Subject-Oriented Programming—A Critique of Pure Objects. *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993.

[20] Gerard J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering, Special issue on Formal Methods in Software Practice*, Volume 23, Number 5, May 1997, 279-295.

[21] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 2, April 1995, pp. 109-145.

[22] Daniel Jackson. Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 4 (October), 1995, pp. 365–389.

[23] Daniel Jackson. *Alloy: A Lightweight Object Modelling Notation*. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, Mass, February 2000. Available at: http://sdg.lcs.mit.edu/~dnj/abstracts.html#alloy.

[24] Daniel Jackson. *Alcoa: Alloy Constraint Analyzer.* Tool, examples and documentation available at: http://sdg.lcs.mit.edu/alcoa.

[25] Michael Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[26] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484-495.

[27] Cliff Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.

[28] Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *Proc. International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin, Aspect-Oriented Programming. *Proceedings of European Conference on Object-Oriented Programming* (ECOOP 97), pp. 220–242.

[30] J. Knoop, B. Steen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[31] K. Rustan M. Leino. *Toward reliable modular programs.* Technical Report CS-TR-95-03, California Institute of Technology, January 1995.

[32] G. Murphy and D. Notkin, Lightweight Source Model Extraction. *Proceedings of the ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering*.

[33] Gail C. Murphy, David Notkin and Kevin Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering* (FSE '95), October 1995.

[34] Robert O'Callahan and Daniel Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. *Proc. International Conference on Software Engineering*, Boston, MA, May 1997.

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

[36] *Precise UML Group*. http://www.cs.york.ac.uk/36/.

[37] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[38] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.

[39] E. Ruf. Context-insensitive alias analysis reconsidered. *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[40] John Rushby. *Formal Methods and Digital Systems Validation for Airborne Systems*. NASA Contractor Report 4551, December 1993. *Chapter 2: Issues in Formal Methods.*

[41] Martin Rinard and John Whaley. *Compositional pointer and escape analysis for multithreaded Java programs.* Technical Report MIT-LCS-TR-795, Laboratory for Computer Science, Massachusetts Institute of Technology, November 1999.

[42] S. Savage, M. Burrows, G. Nelson, P. Solbovarro, and T. Anderson. Eraser: A dynamic race detector for multithreaded programs. *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[43] P. Sathyanathan and M. Lam. Context-sensitive interprocedural pointer analysis in the presence of dynamic aliasing. *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[44] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second edition, Prentice Hall, 1992.

[45] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[46] Bjarne Steensgaard. Points-to analysis in almost linear time. *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[47] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall International Series in Computer Science, 1996.

[48] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[49] Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology,* II(4):379-411, October 1993.