Combined Static and Dynamic Analysis

Cyrille Artho

Computer Systems Institute, ETH Zürich, Switzerland

Armin Biere

Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria

Abstract

Static analysis is usually faster than dynamic analysis but less precise. Therefore it is often desirable to retain information from static analysis for run-time verification, or to compare the results of both techniques. However, this requires writing two programs, which may not act identically under the same conditions. It would be desirable to share the same generic algorithm by static and dynamic analysis. In JNuke, a framework for static and dynamic analysis of Java programs, this has been achieved. By keeping the architecture of static analysis similar to a virtual machine, the only key difference between abstract interpretation and execution remains the nature of program states. In dynamic analysis, concrete states are available, while in static analysis, sets of (abstract) states are considered. Our new analysis is generic because it can re-use the same algorithm in static analysis and dynamic analysis. This paper describes the architecture of such a generic analysis. To our knowledge, JNuke is the first tool that has achieved this integration, which enables static and dynamic analysis to interact in novel ways.

Key words: Static analysis, dynamic analysis, Java.

1 Introduction

Java is a popular object-oriented, multi-threaded programming language. Verification of Java programs has become increasingly important. Two major approaches have been established: *Static analysis* and *dynamic analysis*. Static analysis approximates the set of possible program states. It includes abstract interpretation [8], where a fix point over abstract states, which represent sets of concrete states, is calculated. Static analysis scales well for many properties, as they may only require summary information of dependent methods or modules. "Classical" static analysis constructs a graph representation of the program and calculates the fix point of properties using that graph [8]. This is very different from dynamic analysis, which evaluates properties against an event trace originating from a concrete program execution. Using a graph-free analysis [15], static analysis is again close

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

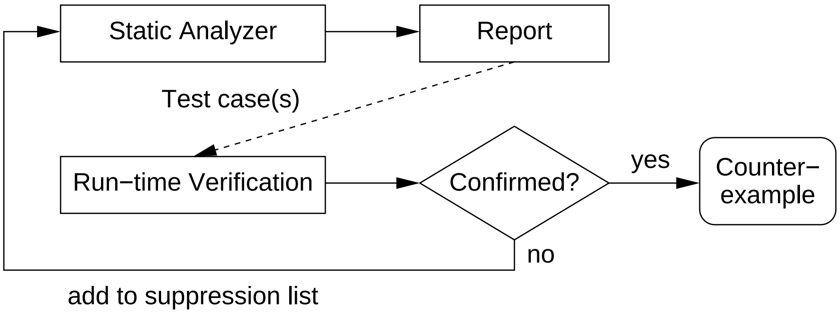

Figure 1. A new tool flow for fault detection using combined static and dynamic analysis.

to dynamic execution. In this paper, a graph-free static analysis is extended to a *generic analysis* which is applicable to dynamic analysis as well.

JNuke [4] is a fully self-contained framework for loading and analyzing Java class files. Originally JNuke was designed for dynamic analysis, encompassing explicit-state software model checking [18] and run-time verification [1].

Static analysis can be very efficient for checking properties such as block-local atomicity [3]. In the initial version, static analysis in JNuke could not handle recursion and required algorithms to be targetted to a static environment [3]. This paper describes the solution for recursion and furthermore allows sharing of algorithms in a static and dynamic environment.

JNuke's generic analysis framework allows the entire analysis logics to be written such that they are agnostic of whether the "environment" is a static or dynamic analysis. Both versions require only a simple wrapper that converts environment-specific data into a form that a generic algorithm can use. Furthermore, the fact that the algorithm itself is identical for static and dynamic analysis allows a novel kind of combined analysis for fault detection, as outlined in Figure 1. A static analyzer looks for faults. Reports are then analyzed by a human, who writes test cases for each kind of fault reported. Run-time verification will then analyze the program using the dynamic version of the same algorithm, possibly confirming the fault as a failure or counterexample. If a failure is not confirmed, even after multiple iterations of creating test cases, given reports can be suppressed in future runs of the static analyzer. Of course this particular application gives up soundness but facilitates fault finding. Current approaches only offer a manual review of reports. The generic algorithm is shared by both tools, which is our contribution and enables this tight integration of static and dynamic analysis.

Section 2 introduces static analysis in JNuke. Generic analysis algorithms, applicable to both a static and dynamic context, are described in Section 3. Section 4 shows the viability of this approach based on experiments. Section 5 concludes.

2 Static Analysis in JNuke

In JNuke, static analysis works very much like dynamic execution, where the *environment* only implements non-deterministic control flow. It thus implements a graph-free data flow analysis [15] where data locality is improved because an entire path of computation is followed as long as valid new successor states are discovered. Each Java method can be executed in this way. The abstract behavior of the program is modelled by the user. The environment runs the analysis algorithm until an abortion criterion is met or the full abstract state space is exhausted.

The iteration over the program state space is separated from the analysis logics. A generic *control flow module* controls symbolic execution of instructions, while the analysis algorithm deals with the representation of (abstract) data and the semantics of the analysis. The control flow module implements a variant of priority queue iteration [12], executing a full path of computation as long as successor states have not been visited before, without storing the flow graph [15].

The generic control flow module first chooses an instruction to be executed from a set of unvisited states. It then runs the specific analysis algorithm on that unvisited state. That algorithm updates its abstract state and verifies the properties of interest. After evaluation of the current instruction, the control flow module adds all valid successor states to the queue of states to visit, avoiding duplicates by keeping a set of seen states. When encountering a branch instruction such as `switch`, all possible successors are added to the state space. Furthermore, each possible exception target is also added to the states that have to be explored.

Many Java bytecode instructions do not affect control flow. Therefore our algorithm does not store the current state if an immediate successor instruction is eligible. A state is only stored if it is target of a branch instruction. This reduces memory usage [15] but may visit a state twice: If an instruction i_b is the target of a backward jump, such as in a `while` loop, it is only recognized as such when the branch instruction is visited, which usually occurs after i_b has been visited before. However, this overhead is small since it only occurs during the first iteration.

It is up to the specific analysis algorithm to model data values. Currently, only the block-local atomicity analysis for stale values [3] is implemented. This analysis tracks the state of each register (whether it is shared and therefore possibly stale) and includes a simple approximation of lock identities (pointer aliasing [20]). It does not require any further information about the state of variables, and thus chooses to execute every branch target. Due to the limited number of possible states for each register, the analysis converges very quickly.

3 Generic Analysis Algorithms

As described in the introduction, the goal was to use *generic analysis algorithms*. These algorithms should work equally in both a *static environment* (using abstract interpretation) and a *dynamic environment* (using run-time verification). The problem is that the environments are quite different: the VM offers a single fully detailed state. Abstract interpretation [8], on the other hand, deals with sets of states, each state containing imprecise information that represents several concrete states. The challenge was to reconcile the differences between these two worlds and factor out the common parts of the algorithm.

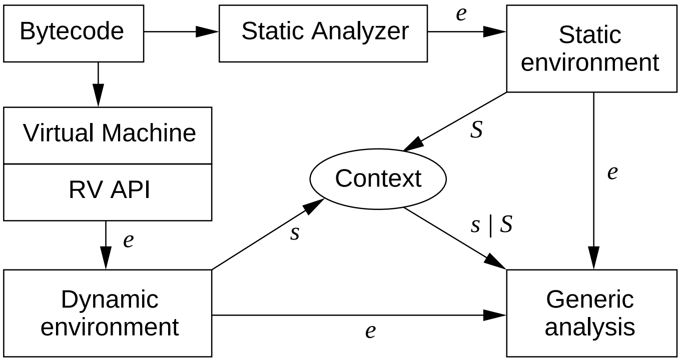

Figure 2. Running generic analysis algorithms in a static or dynamic environment.

A generic analysis represents a single program state or a set of program states at a *single* program location. It also embodies a number of event handlers that model the semantics of byte code operations. Both static analysis and run-time analysis trigger an intermediate layer that evaluates the events. The environment hides its actual nature (static or dynamic) from the generic algorithm and maintains a representation of the program state that is suitably detailed.

Figure 2 shows the principle. Run-time verification is driven by a *trace*, a series of events e emitted by the run-time verification API. An event represents method entry or exit, or execution of an instruction at location l. Conventional run-time analysis analyzes these events directly. The dynamic environment, on the other hand, uses the event information to maintain a *context* c of algorithm-specific data before relaying the event to the generic analysis. This context is used to maintain state information s that cannot be updated uniformly for the static and dynamic case. It is updated similarly by the static environment, which also receives events e, determining that successor states at l are to be computed. The key difference for the static environment is that its updates to c concern *sets of states* S. Sets of states are also stored in components used by the generic algorithm. Operation on states (such as comparisons) are performed through delegation to component members. Therefore the “true nature” of state components, whether they embody single concrete states or sets of abstract states, is transparent to the generic analysis. It can thus be used statically or dynamically.

The abstract domain is chosen based on the features required by the generic analysis to evaluate given properties. Both the domain and the properties are implemented as an observer algorithm in JNuke. Future algorithms may include an interpreter for logics such as LTL. Interpretation of events with respect to temporal properties would then be encoded in the generic analysis while event generation would be implemented by the static and dynamic environment, respectively.

3.1 Context data

Context data c has to be applicable to static and dynamic analysis. The dynamic environment maintains a single (current) context c, while the static environment maintains one context per location, c_l. In a static environment, certain data may not be defined precisely; for instance, in a field access, the static environment often cannot provide a pointer to the instance of which the field was accessed. There are two ways to deal with this problem: The generic analysis must not require such data, or the static layer must insert artificial values. The latter was used for modeling static *lock sets*, where the static layer uses symbolic IDs to distinguish locks, rather than their pointers. On each lock acquisition, the lock set in c_l is updated with a new such lock ID. The generic analysis may only read locks or perform nondestructive, abstract tests, such as testing set intersections for emptiness. Due to polymorphism (in the implementation) of the actual set content, the generic analysis therefore never becomes aware of the true nature of the locks. The environment also maintains contextual information for each lock, such as the line number where it was acquired. Again, polymorphism allows lookup from locks to line numbers without revealing the content of the lock.

In general, the environment must create suitable representations of state information used by the generic analysis. The generic analysis only operates on such data. The environment thus acts as a proxy [11] for the virtual machine, if present, or replaces that data with appropriate facsimiles in static analysis. These facsimiles have to be conceptually isomorphic with respect to concrete values obtained during run-time analysis. Distinct objects have to map to distinct representations. Of course, true isomorphism is only achieved if pointer analysis is absolutely accurate.

The generic block-local atomicity algorithm [3] has the property that it is agnostic to certain concrete values (such as the values of integers) but needs precise information about others (locks). It thus provides a good example of a generic analysis algorithm, as other ones are expected to show similar differences. In the block-local atomicity algorithm, the static environment approximates the lock set, representing it with proxy objects; the dynamic environment simply queries the VM. The property check itself is completely independent of the environment, as it refers to “shadow data” which reflects the status of each register, i.e., whether their value is stale or not. In the static case, the semantics of sets of states are reflected by approximating the set of all possible values in the operations on registers.

3.2 Interfacing run-time verification

Many run-time verification algorithms, such as Eraser [16], are context-sensitive and not thread-local. Such an algorithm receives events from *all* threads and methods. A run-time variant of such an algorithm therefore requires only a single instance of the analysis. In such cases, creating a static variant is less interesting since the dynamic algorithm, if used with a good test suite, yields excellent results [5].

Conversely, analyzing a context-insensitive (method-local), thread-local property is more amenable to static analysis, but actually makes run-time analysis more difficult. This is counter-intuitive because such properties are conceptually simpler. However, in run-time verification, a *new instance* of the analysis has to be created on each method call and thread. Instances of analysis algorithms then correspond to stack frames on the program stack. Each new analysis instance is completely independent of any others, except for a shared, global context (such as lock sets, which are kept throughout method calls) and return values of method calls. The dynamic environment maintains the shared context and relays return values of method calls to the analysis instance corresponding to the caller. Detailed knowledge of run-time verification is not necessary for the remainder of this paper; more about run-time verification in JNuke is described in the extended version of this paper [2].

3.3 Interfacing static analysis

Static analysis calculates the set of all possible program states. Branches (test nodes) are treated non-derministically by considering all possible successors and copying *(cloning)* the current state for each outcome. Junction nodes denote points where control flow of several predecessor nodes merges [8]. In this paper, the operation that creates a new set of possible states at this node will be called *merging.*

The key is that the generic algorithm is not aware that static analysis requires copying and merging operations. To achieve this, the capabilities of the generic analysis must be extended with the *Mergeable* interface. The extended class inherits the algorithm and delegates cloning and merging states to the components of a state. By merging states, sets of states are generated. Computations of state components must therefore support set semantics for static analysis. What is important is that the *analysis logics* are unchanged: the generic algorithm is still unaware that cloning, merging, and set operations happen “behind the scenes” and implements its property checking as if only a single state existed. In some cases, static analysis may converge slowly; convergence is improved by using a widening operator [8] which can be implemented by the merge operation.

In dynamic analysis, only one program location l is active (per thread), corresponding to a single program state s. In static analysis, the sets of states S_l at all program locations l are of interest. Each set of states S_l is represented by an instance of the generic algorithm. The type of operation performed to model the semantics of each instruction remains the same for static and dynamic analysis.

In our framework, the successor states of one set S_l are calculated in each iteration. The choice of l is implemented by a control flow module, as described in Section 2. This covers intra-method analysis, leaving open the problem of method calls. It is desirable that the entire statically reachable call graph is traversed so each reachable method in a program is analyzed. A *recursion* class solves this challenge. It *expands* a method call by starting a new instance of the control flow class. Figure 3 shows an overview of these connections. The recursion class starts with the `main` method and creates a new instance of the control flow class for each called method. The control flow class performs intra-method analysis and delegates method calls back to the recursion class, which also handles multi-threading by exploring the behavior of threads when encountering a thread start point, e.g. a `run` method. This way, the algorithm explores the behavior of all threads.

This leaves open the problem of self-recursion or mutual recursion. It is not possible to model the effects of a recursive method that calls another method higher

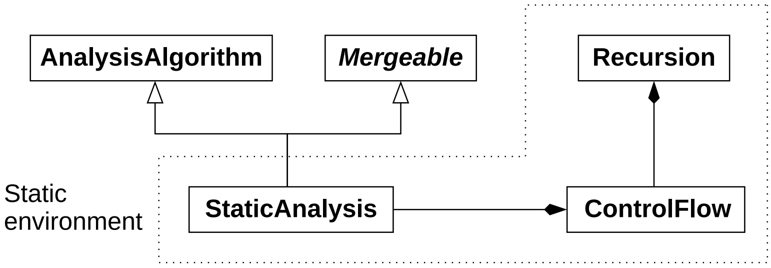

Figure 3. Interfacing static analysis with a generic analysis algorithm.

up in its stack frame using this algorithm. This is because the precise effect of that method call depends on itself. [1] Therefore the static analysis class has to implement a *summary* method, which models method calls without requiring knowledge about the effects of a method. Such a summary method can conservatively assume the worst possible outcome or provide more precise information.

The effect of each evaluated method call is stored as a summary. Context-sensitivity is modeled by evaluating each method call once for each possible call context. If an analysis is context-insensitive, an empty call context is assumed, having the effect that each method is only evaluated once.

In principle, every analysis algorithm can be split up into a generic algorithm and its environment. Most data flow problems can be seen as set-theoretic or closure problems [14] and their nature will affect how the merge operation is implemented. Precision of the analysis will depend on the approximation of pointer aliasing [20]. If accurate information about data values is needed or when environment-specific optimizations are called for, the generic part of an algorithm may become rather small compared to the size of its (static or dynamic) environment. However, with the block-local atomicity algorithm, it has been our experience that the generic algorithm does indeed embody the entire logics and thus is not just a negligeable part of the whole. Notably, adapting a static algorithm for dynamic analysis is greatly facilitated with our approach.

4 Experiments

The block-local atomicity algorithm [3] has been implemented as a generic algorithm that can be used to compare the static and dynamic approach. It analyzes method-local data flow, checking for copies of shared data *(stale values)* that are used outside the critical section in which shared data was read [7]. This analysis only requires reference alias information about locks, making it a suitable candidate for both static and dynamic analysis. Table 1 summarizes the benchmark programs used to compare the static and dynamic version of the stale-value analysis [3].

The static analysis module includes a *suppression list* to avoid a few common cases of false positives. The list contains three methods which return thread-local information, corresponding to the hand-over protocol for return data [13].

[1] A bounded expansion of recursion is possible, approximating the unbounded behavior.

Benchmark	Size [LOC]	Description
Daisy [10]	1900	Multi-threaded (simulated) file system
DiningPhilo [9]	100	Dining Philosophers (3 threads, 5,000 iterations)
JGFCrypt [6]	1700	Large cryptography benchmark
ProdCons [9]	100	Producer/Consumer simulation (12,000 iterations)
Santa [17]	300	Santa Claus problem
SOR [19]	250	Successive Over-Relaxation over a 2D grid: 5 iterations, 5 threads
TSP [19]	700	Travelling Salesman Problem

Table 1
Benchmark programs.

Benchmark	Run-time verification			Static analysis		
	Reports	Time [s]	Mem. [MB]	Reports	Time [s]	Mem. [MB]
Daisy	0	11.03	23.9	3 [ro, tl, tl]	0.17	1.9
DiningPhilo	0	9.45	20.4	0	0.02	0.3
JGFCrypt	0	1127.92	36.6	0	0.14	1.9
ProdCons	1 [buf]	4.35	7.0	1 [buf]	0.01	0.2
Santa	0	0.25	1.4	0	0.04	0.8
SOR	0	32.95	2.5	0	0.11	1.5
TSP, size 10	0	2.76	3.3	2 [exc]	0.09	1.1

Table 2
Benchmark results.

The experiments emphasize the aim of applying a tool to test suites of real-world programs without user-defined abstractions or annotations. All experiments were run on a Pentium 4 with a clock frequency of 2.8 GHz and 1 MB of level II cache. Table 2 shows the results of run-time verification and static analysis. In both cases, the number of reports, run time, and memory consumption are given. About 25 small programs used for testing, which were correctly verified, are omitted.

Run times for dynamic analysis are still quite high, even though Java foundation methods have been omitted from being monitored. A very effective optimization would therefore exclude any methods that can be statically proven to be safe.

Given warnings are all false positives.[2] In Daisy, they were caused by read-only [ro] and thread-local [tl] values. For the ProdCons benchmark, the stale value

[2] A more precise pointer analysis could suppress such warnings. Run-time verification would never report false positives concerning thread-local data, such as in the five cases in Daisy and TSP, due to fully accurate pointer information.

comes from a `synchronized` buffer [buf] and is thread-local [13]. The two false alarms for the TSP benchmark are caused by thread-local exceptions [exc].

The overall experience shows that the approach works as envisioned. Experiments clearly indicate that static analysis is a lot faster, while being less precise. The staggering difference in execution times for the two analysis types is easily explained: for SOR, for instance, the dynamic version generates many thousands of objects, on which a series of mathematical operations is performed. In the static version, each method is only executed once, which by itself reduces complexity by many orders of magnitude. In summary, given experiments show that the framework is fully applicable to real-world programs, analyzing them both statically or dynamically depending on whether one requires a fast analysis or high precision.

5 Conclusion and Future Work

Static and dynamic analysis algorithms can be abstracted to a generic version, which can be run in a static or dynamic environment. By using a graph-free analysis, static analysis remains close enough to program execution such that the algorithmic part can be re-used for dynamic analysis. The environment encapsulates the differences between these two scenarios, making evaluation of the generic algorithm completely transparent to its environment. This way, the entire analysis logics and data structures can be re-used, allowing for comparing the two technologies with respect to precision and efficiency. Experiments with JNuke have shown that the static variant of a stale-value detection algorithm is significantly faster but less precise than the dynamic version. This underlines the benefit of using static information in order to reduce the overhead of run-time analysis. The fact that both types of analysis share the algorithm also allows for combining them in a tool that applies run-time verification to test cases resulting from static analysis reports.

Future work includes evaluation of our combined analysis for fault detection, and porting more algorithms to the generic framework. Furthermore, run-time verification in JNuke needs more commonly used classes and libraries, while static analysis in JNuke is still limited by the lack of a precise pointer analysis.

References

[1] *1st, 2nd, 3rd and 4th Workshops on Runtime Verification (RV ’01 – RV ’04)*, volume 55(2), 70(4), 89(2), 113 of *ENTCS*. Elsevier Science, 2001 – 2004.

[2] C. Artho and A. Biere. Combined static and dynamic analysis. Technical Report 466, ETH Zürich, Zürich, Switzerland, 2005.

[3] C. Artho, A. Biere, and K. Havelund. Using block-local atomicity to detect stale-value concurrency errors. In Farn Wang, editor, *Proc. ATVA ’04*. Springer, 2004.

[4] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In R. Alur and D. Peled, editors, *Proc. CAV ’04*, Boston, USA, 2004. Springer.

[5] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian rover software. *Formal Methods in System Design*, 25(2), 2004.

[6] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking Java Grande applications. In *Proc. ACM Java Grande Conference*, 1999.

[7] M. Burrows and R. Leino. Finding stale-value errors in concurrent programs. Technical Report SRC-TN-2002-004, Compaq SRC, Palo Alto, USA, 2002.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symp. Principles of Programming Languages*. ACM Press, 1977.

[9] P. Eugster. Java Virtual Machine with rollback procedure allowing systematic and exhaustive testing of multithreaded Java programs. Master's thesis, ETH Zürich, 2003.

[10] S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[12] S. Horwitz, A. Demers, and T. Teitebaum. An efficient general iterative algorithm for dataflow analysis. *Acta Inf.*, 24(6):679–694, 1987.

[13] D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.

[14] T. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proc. 17th ACM SIGPLAN-SIGACT*, pages 184–196, San Francisco, USA, 1990. ACM Press.

[15] M. Mohnen. A graph-free approach to data-flow analysis. In *Proc. 11th CC*, pages 46–61. Springer, 2002.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems*, 15(4), 1997.

[17] J. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

[18] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[19] C. von Praun and T. Gross. Object-race detection. In *OOPSLA 2001*, Tampa Bay, USA, 2001. ACM Press.

[20] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. OOPSLA '99*, pages 187–206, Denver, USA, 1999. ACM Press.