Combined Static and Dynamic Analysis

Cyrille Artho[1] and Armin Biere[2]

[1] Computer Systems Institute, ETH Zürich, Switzerland
[2] 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.

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 [14], which typically encompasses a (strictly) wider range of behaviors than the original, using an abstract version of the program. Abstract states represent sets of concrete states. The analysis iterates over these abstract states until a fixpoint is reached or a certain limit, such as a second loop execution, is reached. 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 [14]. 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 [26], static analysis is again close 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.

Dynamic analysis actually executes the system under test. This has the key advantage of having precise information available. Some fully automated dynamic analysis algorithms even only require a single execution trace to deduce possible errors [7,30].

This fact is the foundation of run-time verification [1], ameliorating the major weakness of testing, which is the possible dependence of a system execution on the thread schedule. If a test suite with high coverage exists, run-time verification can thus explore a large part of the possible behaviors while scaling significantly better than software model checking. Dynamic analysis requires an execution environment, such as a Java Virtual Machine (VM). However, typical Java VMs only target execution and do not offer all required features, in particular, full state access. Code instrumentation, used by JPaX [20], can solve this problem to some extent only [19]. JNuke contains a specialized VM allowing for both backtracking and full state access. Custom checking algorithms can be implemented using an API that allows an algorithm to register for any event of interest.

1.1 Overview of JNuke

JNuke [8] is a fully self-contained framework for loading and analyzing Java class files. The class loader includes a type checker which implements bytecode verification to ensure the well-formedness of class files to be loaded [24]. It also transforms the byte code into a reduced instruction set, after inlining intra-method subroutines. Additionally, explicit registers are introduced to replace the operand stack [5]. A peep-hole optimizer takes advantage of the register-based byte code. Originally JNuke was designed for dynamic analysis, encompassing explicit-state software model checking [32] and run-time verification [1].

At the core of JNuke is its VM, providing check-points [15] for explicit-state model checking and reachability analysis through backtracking. A check-point allows exploration of different successor states in the search, storing only the difference between states for efficiency. For generic run-time verification, the engine executes only one schedule defined by a given scheduling algorithm. An observer interface provides access to events occurring during program execution. Event listeners can then query the virtual machine for detailed data and thus implement any run-time verification algorithm.

Static analysis was added to JNuke at a later stage. In the initial version, static analysis in JNuke could not handle recursion and required algorithms to be targetted to a static environment [6]. 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.

For portability and best possible efficiency, JNuke was implemented in C. A lightweight object-oriented layer has been added, allowing for a modern design without sacrificing speed. The roughly 1700 unit tests make up half of the 130,000 lines of code (LOC). Full statement coverage results in improved robustness and portability. JNuke runs on Mac OS X and the 32-bit and 64-bit variants of Linux (x86 and Alpha) and Solaris.

1.2 Motivation for this framework

Even a fast execution environment is greatly slowed down by run-time verification and thus needs support from a static data flow analysis in order to reduce the amount of data to be monitored at run-time. Ideally this functionality is pluggable in the class loader. Because the entire framework is integrated, difficult (and often lossy) conversion of static information for dynamic analysis is not necessary.

It is not always certain whether it is beneficial to implement a static or a dynamic analysis for a specific property. Static analysis can scale easily to a million lines of code per minute or more [3,6] if it does not require complex pointer aliasing information. The block-local atomicity analysis algorithm [6] seemed to be most suitable for static analysis because the property checked is context-insensitive (method-local). However, accuracy depends heavily on the quality of the pointer analysis used [34]. Imprecise lock information may generate spurious warnings. Therefore it is interesting to see whether a dynamic version of the same algorithm produces better results.

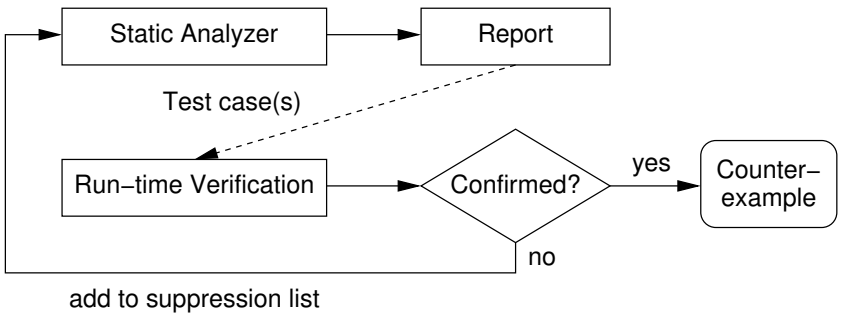

Fig. 1. A new tool flow for fault detection using combined static and dynamic analysis.

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.

1.3 Outline

This paper is the extended version of a previously published paper [4]. Section 2 introduces static analysis in JNuke. Generic analysis algorithms, applicable to both a static

and dynamic context, are described in Section 4. Section 5 shows the viability of this approach based on experiments. Section 6 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 [26] 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.

2.1 Graph-free abstract interpretation

Abstract interpretation of a program involves computation of the least or greatest fix point of a system of semantics of the form:

$$\begin{cases} x_1 = \Phi_1(x_1, \ldots, x_n) \\ \quad \vdots \\ x_n = \Phi_n(x_1, \ldots, x_n) \end{cases}$$

where each index $i \in C = [1,n]$ represents a location of the program, and each function Φ_i is a continuous function from L^n to L. L is the abstract lattice of program properties. Each function Φ_i computes the property holding at i after one program step executed. Applying the equations iteratively therefore computes the solution eventually, but is inefficient [9]. A well-established speed-up technique consists of widening [14], where some equations for x_i are replaced with $x_i = x_i \nabla \Phi_i(x_1, \ldots, x_n)$, operator ∇ being a safe approximation of the least upper bound such that the iteration strategy eventually terminates. For optimal performance, widening operators have to be tuned for a specific iteration strategy [9].

In graph-free abstract interpretation [26], the properties x_i at each program location i are represented by an abstract state, representing a set of concrete states S. Computation of these properties from the abstract state is quite straightforward and can be done uniformly for all locations, by using a property predicate P:

$$\begin{cases} x_1 = P(S_1) \\ \quad \vdots \\ x_n = P(S_n) \end{cases}$$

The central problem is thus computation of the fix point of all abstract states S_i:

$$\begin{cases} S_1 = \text{absexec}(1, S_1) \\ \quad \vdots \\ S_n = \text{absexec}(n, S_n) \end{cases}$$

Since this technique already uses an execution environment, control flow structures are evaluated correctly and efficiently [26]. Whenever a new abstract state S_i' is computed, the effects of the instruction at i are calculated based on the semantics of the abstract domain: $S_i' = \text{absexec}(i, S_i)$. The difference to dynamic execution therefore lies in the nature of an abstract state S_i which can represent several concrete states $\{s_j, \ldots, s_k\}$, some of them possibly unreachable in real execution. Function absexec may also overapproximate its result S_i', which can be compared to widening as described in Section 4.

2.2 Separation of control flow and bytecode semantics

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 [22], executing a full path of computation as long as successor states have not been visited before, without storing the flow graph [26]. Abstract states s as used in this algorithm refer to a set of program states at a single location l. A single abstract state at l thus usually represents a set of concrete states at that location.

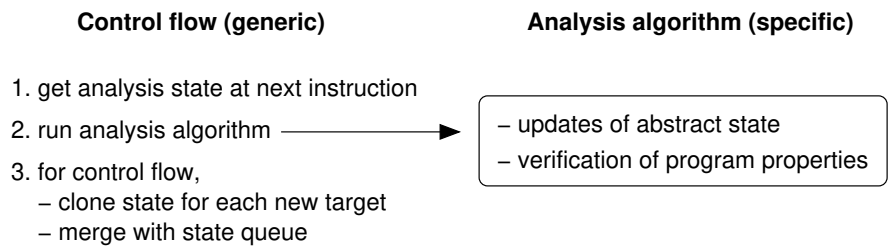

Fig. 2. Separation of control flow and analysis algorithm.

Figure 2 shows the principle of state space exploration: The generic control flow module first chooses an instruction to be executed from a set of unvisited abstract states. It then runs the specific analysis algorithm on that unvisited abstract 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.

It is up to the specific analysis algorithm to model data values. Currently, only the block-local atomicity analysis for stale values [6] 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 [35]). 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.

2.3 Optimized state space management

After the specific algorithm has calculated the outcome of the current abstract state, the control flow algorithm evaluates all possible successor instructions. To achieve this, the current abstract state is cloned for each new possible successor state. The control flow module then adds this state to the queue of states to visit. This corresponds to a basic model-checking algorithm [21] but is not the most efficient way to perform static analysis for software. The observation was that a lot of states were stored and then immediately re-fetched in the next cycle of the main loop. This is because many instructions in Java do not affect control flow. Therefore the above algorithm from Figure 2 was modified to only store a state if (a) it generates multiple successor states or (b) another state with the same program counter had been visited before. This has the effect that the major part of a method is executed "linearly" without storing the current state. Only if a branch instruction occurs, the state is cloned and stored.

The reason why this optimization works well is that many Java bytecode instructions do not affect control flow. Therefore our algorithm does not store the current state if a unique immediate successor instruction is eligible. A state is only stored if it is target of a branch instruction. This reduces memory usage [26] 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. As an example, assume some execution visits states 1 – 5 and then branches back to state 3. No state is stored until state 5 is reached. The current abstract state at 5 is stored since its code consists of a branch instruction. States 3 and 4 are then re-visited because the algorithm has not stored them during its first iteration. During the second iteration, state 3 is stored because it is now known to be the target of a backward jump. Therefore, if the abstract program state at 3 does not change during future loop iterations, that state is not re-visited anymore.

3 Run-time verification in JNuke

JNuke implements a virtual machine that can execute the full set of Java bytecode instructions [24] and therefore any Java program given an implementation of the native code used by it. An application programming interface (API) allows event listeners to connect to any action of interest and query the VM about its internal state, thus implementing any analysis algorithm of choice.

Prior to execution, the class loader transforms the Java bytecode into a more abstract, RISC-like version that only contains 27 instructions. The result is then further transformed into a register-based version of the originally stack-based bytecode [5]. This allows a peep-hole optimizer to reduce some of the complex sequences of instructions used to implement arithmetic operations, which have to copy all operands on the stack first in the original version. Execution of the program generates a series of events,

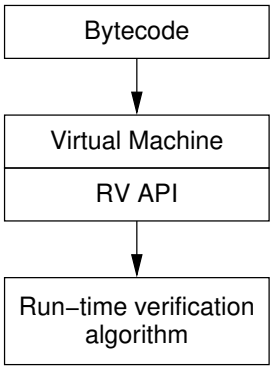

Fig. 3. Run-time verification in JNuke.

denoted by an event *trace*. During execution, the run-time verification API (RV API) offers event listeners, which can capture the event trace. Such listeners are used to implement scheduling policies and run-time verification algorithms. The algorithm is responsible to copy data it needs for later investigation, as the VM is not directly affected by the listeners and thus may choose to free data not used anymore. Figure 3 shows an overview of the JNuke VM and how a run-time verification algorithm can be executed by callback functions in the VM. For simplicity, the figure omits the fact that some communication from the RV algorithm back to the VM actually occurs in the presence of garbage collection. In such a situation, the RV algorithm must instruct the VM to suppress collection of data it is still using, in order to prevent access to memory locations that are already freed or reallocated for other data [16]. Such a protection applies to all algorithms that use references to identify data.

3.1 JNuke VM

Figure 4 shows the key components of the VM [15]. The core parts are subsumed by the *run-time environment*, which controls the execution and effects of single instructions. It also loads classes on demand (using a linker module which is not shown in the figure). After each instruction, an exchangeable *scheduler* decides whether a thread switch should occur. If this is the case, the run-time environment puts the current thread to sleep and enables a new one. This action may involve updates on the lock sets of each thread, which is done via the *lock manager*. Inter-thread communications are queued by the *waitset manager,* while any heap content is updated by the *heap manager*. The heap manager allocates and frees data and is partially accessed directly by the run-time environment, partially by the *garbage collector*. The run-time environment can run with or without garbage collection [16].

This modularization of the VM allowed for a more flexible design. After an initial implementation, each module was augmented with a rollback capability which allows storing and restoring the entire state of the VM. This can be used to explore each possibility in the presence of non-determinism, such as non-determinism arising from thread

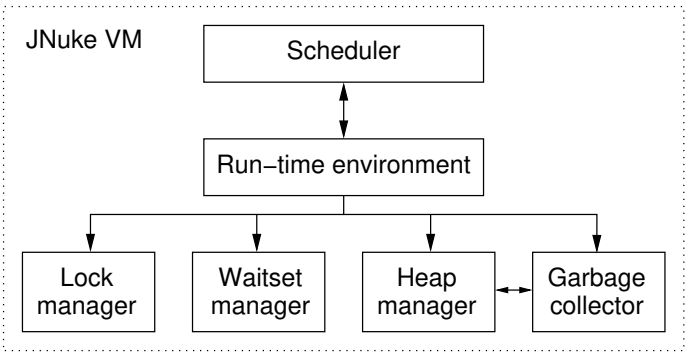

Fig. 4. Overview of the key components of the JNuke VM.

switches [11]. A special scheduler uses this to perform explicit-state model checking for Java programs [8,15]. These interfaces also ensure that native methods needed to perform system calls operate in a well-behaved manner and do not corrupt the Java heap.

3.2 Run-time verification API

The run-time verification API (RV API) allows algorithms to register *listeners* for events of interest, such as a lock acquisition. These listeners are notified through an observer interface [18] whenever such an event occurs. After registering all event handlers, the virtual machine is started as usual. It will call the registered listeners whenever an event of interest occurs. The call includes light-weight event data, containing the exact type of event and a pointer to the run-time environment. The first part of the data is used to distinguish subclasses of events. For instance, read and write accesses may share the same event handler, but the handler may still need to know the exact nature of the access in one decision. The second part of event data serves to query the virtual machine about more information, such as the exact state of each thread. Some events include a little extra information for efficiency, so such queries can be eliminated for basic information that is always needed when an event occurs (for instance, which lock was used in a lock release).

Events are separated into different classes, as shown in Table 1. This allows RV algorithms to install generic event handlers to deal with common aspects of a super class of events, which then delegates fine points to particular subclasses. These event handlers may also have to perform *event re-ordering,* because one instruction may generate several events, which do not necessarily occur in the right order. For instance, entry to a `synchronized` method causes three events: the lock acquistion on method entry, method entry itself, and the first bytecode instruction of the method. Certain algorithms may require these events to occur in a certain order in order to work properly. For simplicity, the current API does not allow for specifying the order in which such

Event	Subclasses (if available)	Purpose/possible checks
Field access	Read/write access	Locking discipline (e.g. Eraser)
Lock event	Lock acq./release	Locking (e.g. deadlock detection)
Method event	Method start/end	Call graph construction
Thread creation	–	Record thread name and type
Bytecode execution	Events for all 27 abstract instructions	Model instruction-specific properties
Program termination	–	Final report, clean up RV data

Table 1. Run-time verification events in JNuke.

simultaneous events are received. However, this minor problem, which does not occur often, can be easily solved in the listener implementation.

The RV API itself builds on low-level listeners provided by the VM. The low-level listeners are embedded in the reponsible module: field access events are treated by the heap manager while the lock manager deals with lock events. Other events are issued by the run-time environment itself, such as bytecode execution events. The RV API was created to provide a single front end to all these different event callbacks. It also allows to activate certain auxiliary listeners that log history information as the program executes. This is very useful for printing more detailed trace information. These two features, a simple front end and history information, greatly reduce the amount of work required for implementing a run-time verification algorithm.

4 Generic Analysis Algorithms

The goal of this extension to JNuke was to be able 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 [14], 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 5 illustrates the problem: The analysis algorithm is duplicated for both analysis scenarios. Much genericity and flexibility is already gained by utilizing a generic observer-based run-time verification interface [2] and a generic iteration module which analyzes control flow [4]. However, the final property-specific part still has to be written twice, adapted to each scenario. This is even though the analysis clearly represents the same rules. The goal is therefore to have a generic analysis. The design that allows to achieve this is the key contribution of this paper.

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

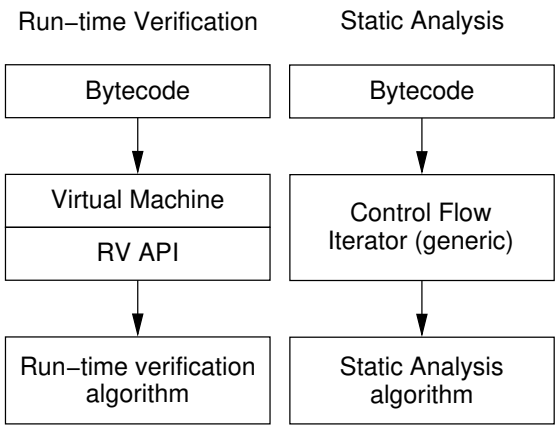

Fig. 5. Classical approaches duplicate the analysis algorithm for the dynamic and static case.

(static or dynamic) from the generic algorithm and maintains a representation of the program state that is suitably detailed.

Figure 6 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.

Existing work in software model checking has shown that using only deterministic choices during state space exploration results in a feasible counter-example trace. This technique therefore corresponds to reducing a set of (abstract) states to a concrete state [27]. However, property verification algorithms as in run-time verification have so far not been applied to the resulting concrete states. This is because in the JPF/Bandera tool chain, the counter-example trace is already known to be concrete at that stage. Instead, the counter-example trace is used for abstraction refinement [27].

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 [28]. Interpretation of events with respect to temporal properties would

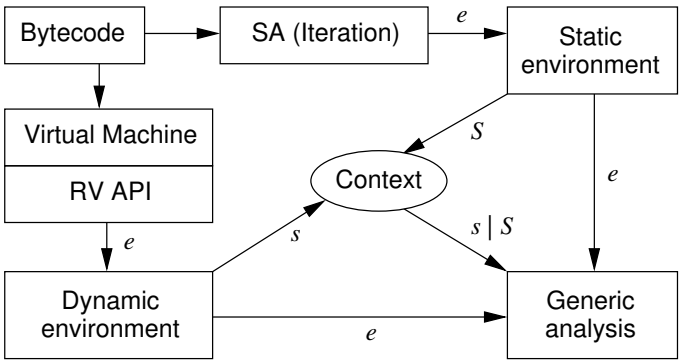

Fig. 6. Running generic analysis algorithms in a static or dynamic environment.

then be encoded in the generic analysis while event generation would be implemented by the static and dynamic environment, respectively.

4.1 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 one 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 non-destructive, 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 [18] 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 proxy objects implemented so far incur little overhead but are rather specialized and may only be re-used if another algorithm has equivalent requirements. For example, if the alias information of two locks is not known, one can either assume they are equal or different. The conservative

approximation of a lock set therefore depends on the algorithm used. A static version of a low-level data race algorithm [30] can safely assume that all locks are different, but this will likely lead to many false positives since the intersection of different lock sets is going to be empty in such cases. The same assumption holds for the high-level data race [7] and block-local atomicity [6] algorithms. Other algorithms may have the reverse requirement, i.e., two locks of which the alias information is unknown have to be treated as equal locks.

Context data	Type	Content for static analysis	Content for dynamic analysis
Current monitor block (ID)	Integer	Integer reflecting approximated non-reentrant lock acquisitions	True count of the total number of lock acquisitions so far
Registers ("stack frame")	Array	Abstract entries containing only information about sets of possible register properties	Shadow values reflecting property of interest (exact status of each register)
Lock set	Set	Integer descriptors for each lock	True lock set
Lock context	Map	Map of integers to locations	Map of locks to locations

Table 2. Context differences for the static and dynamic block-local atomicity analysis.

The generic block-local atomicity algorithm [6] has the property that it is agnostic to certain concrete values (such as the values of integers) but needs relatively 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. Table 2 gives an overview of the differences between the static and dynamic versions of the algorithm.

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. Figure 7 shows an excerpt of this generic algorithm. Its code has been simplified for clarity, using Java-like syntax and ignoring registers with a size of 64 bits. It contains the essence of the idea outlined above: Class `context` stores the lock set, which is updated by the environment and queried by `context.getLockSet()`. Context data is therefore updated with each evaluation step, and queried on demand. A static environment approximates the lock set using proxy objects. Note that the approximation can be made conservative if pointer alias information is imprecise [6]. The dynamic environment simply queries the VM to obtain the real, concrete lock set. The property check itself is completely independent of the environment, as `localvars` refers to "shadow data" which reflects the status of reach register, i.e., whether their value is stale or not [6]. In the static case, the semantics of sets of states are reflected by approximating the set of all possible values in the operations on `localvars` (such as `get` and `set` shown here). Therefore the generic algorithm performs the same operations on concrete states as on sets of abstract states.

```
void atGetField(Bytecode bc) {
    /* Compute effect of GetField instruction w.r.t. stale values. */
    /* Potential data races with the reference to the object instance
     * are discovered by the Eraser lock set algorithm, which monitors
     * individual field accesses. */

    /* Check block-local atomicity property for arguments consumed by
     * this instruction */
    checkRegisters(bc);
    StackElement result = newData(); /* possibly shared, see below */
    localvars.set(bc.getResultRegister(), result); /* store result */
}

StackElement newData() {
    /* Generic case where new data is obtained from a possibly shared
     * field. If data is shared, set correct monitorBlock etc. */

    StackElement data = new StackElement(); /* unshared by default */
    if (context.getLockSet().count() > 0) {
        data.setShared(true);
        data.setMonitorBlock(context.getCurrentMonitorBlock());
    }
    return data;
}

void checkRegisters(Bytecode bc) {
    /* Check each local variable for local atomicity violation. */

    for (int i = 0; i < bc.getNumRegs(); i++) {
      int idx = bc.getRegisterIndex(i);
      if (registerIsLocalVariable(idx)) {
          StackElement data = localvars.get(idx);
          if (data.getShared() &&
              (data.getMonitorBlock() != getCurrentMonitorBlock()))
            /* report error */
        }
    }
}
```

Fig. 7. Excerpt of the block-local atomicity algorithm (simplified).

4.2 Interfacing run-time verification

Many run-time verification algorithms, such as Eraser [30], 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 object holding analysis data. In such a case, creating a static variant is less interesting since the dynamic algorithm, if used with a good test suite, yields excellent results [10].

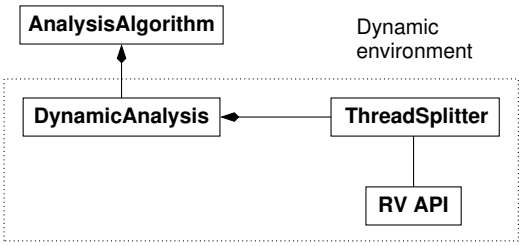

Fig. 8. Interfacing run-time verification with a generic analysis algorithm.

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. The block-local atomicity algorithm serves as an example here, being both thread-local and method-local. For run-time verification, a new instance of this analysis has to be created on each method call and thread. Instances of analysis algorithms then correspond to stack frames on the program stack. Figure 8 contains a UML diagram [29] depicting how the dynamic environment creates instances of an analysis algorithm as needed. The first layer, class *thread splitter,* splits events according to their thread ID, creating a separate instance of class *dynamic analysis* as needed, one for each thread. The second layer, driven by class dynamic analysis, creates a new instance of class *analysis algorithm* for each stack frame, at the beginning of each method call. 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. In this case, lock set information is already available by the RV API and does not have to be managed separately. Other global data can be managed by an extra listener that evaluates events before relaying them to class thread splitter. The thread-specific instances of dynamic analysis deal with communicating return values from an "inner" instance, corresponding to the callee, to the "outer" one, referring to the caller.

4.3 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 [14]. 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, as shown in Figure 9. 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 [14] which can be implemented by the merge operation.

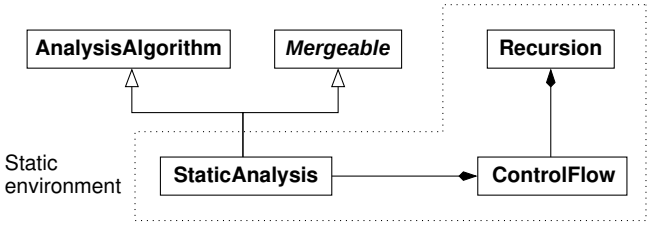

Fig. 9. Interfacing static analysis with a generic analysis algorithm.

In dynamic analysis, only one program location l is active (per thread), corresponding to a single program state s. This current state s is updated and the result assigned to successor state s'; the original state s is then discarded. In static analysis, abstract states S_i at all program locations i are being analyzed. The abstract states are analyzed in an iterative way, and thus the abstract states at each program location are retained until the iteration terminates. Each abstract state S_i 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_i are calculated in each iteration. The choice of i 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 9 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 up in its stack frame using this algorithm. This is because the precise effect of that method call depends on itself.[3] 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 result 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. For a context-insensitive analysis, an empty call context is assumed. Context sensitivity therefore does not directly have an effect on the fact that each method call requires a new instance of control flow and analysis objects. However, once summaries are available, their information will act as a cache. For context-insensitive analysis, the empty call context always matches for a given method, and thus each method call 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 [25] and their nature will affect how the merge operation is implemented. Precision of the analysis will depend on the approximation of pointer aliasing [35]. 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.

5 Experiments

The block-local atomicity algorithm [6] 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 [13]. This analysis only requires reference alias information about locks, making it a suitable candidate for both static and dynamic analysis. Table 3 summarizes the benchmark programs used to compare the static and dynamic version of the stale-value analysis [6].

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 [23]:

- `java/io/BufferedReader.readLine`
- `java/lang/Integer.parseInt`
- `java/util/Date.getTime`

Benchmark	Size [LOC]	Description
Daisy [17]	1900	Multi-threaded (simulated) file system
DiningPhilo [15]	100	Dining Philosophers (3 threads, 5,000 iterations)
JGFCrypt [12]	1700	Large cryptography benchmark
ProdCons [15]	100	Producer/Consumer simulation (12,000 iterations)
Santa [31]	300	Santa Claus problem
SOR [33]	250	Successive Over-Relaxation over a 2D grid: 5 iterations, 5 threads
TSP [33]	700	Travelling Salesman Problem

Table 3. Benchmark programs.

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 4 shows the results of run-time verification and static analysis. Both for run-time verification and static analysis, the number of reports (warnings), the run time, and memory consumption are given. The table omits experiments based on about 25 small programs used for testing, which were all verified correctly.

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 4. Benchmark results.

Run times for dynamic analysis are still quite high, with an overhead of up to a factor of 5.6 compared to normal execution in the JNuke VM. This is 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. [4] In Daisy, they were caused by read-only [ro] and thread-local [tl] values. For the ProdCons benchmark, the stale value comes

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

[4] 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.

```
Object getLock() {
    /* assume some really complex obfuscated code here */
    return this;
}

void correctMethod() {
    Object lock1, lock2;
    int tmp;
    lock1 = getLock();
    lock2 = getLock();
    synchronized (lock1) {
        synchronized (lock2) {
            tmp = getData();
        }
        tmp++;
    }
}
```

Fig. 10. A false positive resulting from redundant locks.

from a `synchronized` buffer [buf] and is thread-local [23]. The two false warnings for the static analysis of the TSP benchmark are caused by thread-local exceptions [exc]. However, an example in which static analysis will provide a false positive because of lock aliasing can be constructed easily. Figure 10 shows a contrived example where two nested locks are used. Assume that `getLock()` is too complex for a precise pointer analysis. Then static analysis will conservatively assume the two locks are different and report the use of a stale value at statement `tmp++`. However, run-time analysis will only use a single monitor block, since the two locks are equal, and not report a false positive. This scenario has been outlined before [6].

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.

6 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, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining Test Case Generation with Runtime Verification. *ASM issue of Theoretical Computer Science*, 2003. Accepted for publication.
3. C. Artho and A. Biere. Applying static analysis to large-scale, multithreaded Java programs. In D. Grant, editor, *Proc. 13th ASWEC*, Canberra, Australia, 2001. IEEE Computer Society.
4. C. Artho and A. Biere. Combined static and dynamic analysis. In *Proc. AIOOL '05*, ENTCS, Paris, France, 2005. Elsevier Science.
5. C. Artho and A. Biere. Subroutine inlining and bytecode abstraction simplify static and dynamic analysis. In *Proc. BYTECODE '05*, ENTCS, Edinburgh, Scotland, 2005. Elsevier Science.
6. 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.
7. C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4), 2003.
8. 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.
9. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141. Springer, 1993.
10. 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.
11. D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.
12. 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.
13. M. Burrows and R. Leino. Finding stale-value errors in concurrent programs. Technical Report SRC-TN-2002-004, Compaq SRC, Palo Alto, USA, 2002.
14. 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.

15. P. Eugster. Java Virtual Machine with rollback procedure allowing systematic and exhaustive testing of multithreaded Java programs. Master's thesis, ETH Zürich, 2003.
16. P. Farkas. Garbage Collection for JNuke, a Java Virtual Machine for Runtime Verification and Model Checking. Master's thesis, ETH Zürich, 2004.
17. S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.
18. 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.
19. A. Goldberg and K. Havelund. Instrumentation of Java bytecode for runtime analysis. In *Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports ETH Zürich*. ETH Zürich, 2003.
20. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Proc. Run-Time Verification Workshop (RV '01)*, volume 55 of *ENTCS*. Elsevier, 2001.
21. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
22. S. Horwitz, A. Demers, and T. Teitebaum. An efficient general iterative algorithm for dataflow analysis. *Acta Inf.*, 24(6):679–694, 1987.
23. D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.
24. T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.
25. 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.
26. M. Mohnen. A graph-free approach to data-flow analysis. In *Proc. 11th CC*, pages 46–61. Springer, 2002.
27. C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *Intl. Journal on Software Tools for Technology Transfer (STTT)*, 5(1), 2003.
28. A. Pnueli. The temporal logic of programs. In *Proc. FOCS '77*, pages 46–57, Rhode Island, 1977. IEEE, IEEE Computer Society Press.
29. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.
30. 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.
31. J. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.
32. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
33. C. von Praun and T. Gross. Object-race detection. In *OOPSLA 2001*, Tampa Bay, USA, 2001. ACM Press.
34. J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*, Washington D.C., USA, June 2004. ACM Press.
35. 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.