Design of Software Model Checker

Mrs. Rupali Kale[1]

Abstract— In a few cases, modeling languages have been designed to resemble programming languages, although the focus has been on protocol designs. Some of these linguistic choices have made, and still make it feasible to more conveniently experiment with new algorithms and frameworks for analyzing system models. In this paper we will attempt to give convincing arguments for why we believe it is time for the formal methods community to shift some of its attention towards the analysis of programs written in modern programming languages. In keeping with this philosophy we have developed verification and testing environment for Java, called Java PathFinder (JPF), which integrates model checking, program analysis and testing. Part of this work has consisted of building a new Java Virtual Machine that interprets Java byte code. JPF uses state compression to handle big states, and partial order and symmetry reduction, slicing, abstraction, and runtime analysis techniques to reduce the state space.

Index Terms—Model checking, Abstraction, Runtime Analysis, static analysis, Java Path Finder, Linear time temporal logic (LTL).

1 INTRODUCTION

In a few cases, modeling languages have been designed to resemble programming languages, although the focus has been on protocol designs. Some of these linguistic choices have made, and still make it feasible to more conveniently experiment with new algorithms and frameworks for analyzing system models. For example, a logic based language is well suited for rewriting, and a rule based guarded command notation is convenient for a model checker. Continued research in special languages is important since this research investigates semantically clean language concepts and will impact future language designs and analysis algorithms. Next important step for the formal methods subgroup of the software engineering community could be to focus some of its attention on real programs written in modern programming languages. Studying programming languages somehow will result in some new challenges that will drive the research in new directions as described in the first part of the paper. In the second part of the paper, we describe our own effort to follow this vision by presenting the development of verification, analysis and testing environment for Java, called Java PathFinder (JPF). This environment combines model checking techniques with techniques for dealing with large or infinite state spaces. These techniques include static analysis for supporting partial order reduction of the set of transitions to be explored by the model checker, predicate abstraction for abstracting the state space, and runtime analysis such as race condition detection and lock order analysis to pinpoint potentially problematic code fragments. Part of this work has consisted of building a new Java Virtual Machine (JVMJPF) that interprets Java byte code. JVMJPF is called from the model checking engine to interpret byte code generated by a Java compiler. It is an attractive idea to develop a verification environment for Java for three reasons. First, Java is a modern language featuring important concepts such as object-orientation and multi-threading within one language. Languages such as C and C++, for example, do not support multi-threading as part of their core. Second, Java is simple, for example compared to C++. Third, Java is compiled into byte code, and hence, the analysis can be done at the byte code level. This implies that such a tool can be applied to any language that can be translated into byte code'. Byte code furthermore seems to be a convenient breakdown of Java into easily manageable byte code instructions; and this seems to have eased the construction of our analysis tool. JPF is the second generation of a Java model checker developed at NASA Ames. The first generation of JPF (JPFI) was a translator from Java to the Promela language of the Spin model checker.

2 CODE ANALYSIS

It is often argued that verification technologies should be applied to designs rather than to programs since catching errors early at the design level will reduce maintenance costs later on. We do agree that catching errors early is crucial. State of the art formal methods also most naturally lend themselves to designs, simply due to the fact that designs have less complexity, which make formal analysis more feasible and practical. Hence, design verification is a very important research topic, with the most recent popular subject being analysis of statecharts, such as for example found in UML [1]. However, we want to argue that the formal methods community should put some of its attention on programs for a number of reasons that we will describe below.

First of all, programs often contain fatal errors in spite of the existence of careful designs. Many deadlocks and critical section violations for example are introduced at a level of detail which designs typically do not deal with, if formal designs are made at all. This was for example demonstrated in the analysis of NASA's Remote Agent spacecraft control system written in the LISP programming language, and analyzed using the Spin model checker [2]. Here several classical multi-threading errors were found that were not really design errors, but rather programming mistakes such as forgetting to enclose code in critical sections. One of the missing critical section errors found using Spin was later introduced in a sibling module, and caused a real deadlock during flight in space, 60,000 miles from earth [18]. Another way of describing the relationship between design and code is to distinguish between two kinds of errors. On the one hand there are errors caused by flaws in underlying complex algorithms. Examples of complex algorithms for parallel systems are communication protocols. The other kind of errors are more simple minded concurrency programming errors, such as forgetting to put code in a critical section or causing deadlocks. This kind of errors will typically not be caught in a design, and they are a real hazard, in particular in safety critical systems. Complex algorithms should probably be analyzed at the design level, although there is no reason such designs cannot be expressed in a modern programming language.

Second, one can argue that since modern programming languages are the result of decades of research, they are the result of good language design principles. Hence, they may be good design modeling languages. This is to some extent already an applied idea within UML where statechart transitions (between control states) can be annotated with code fragments in your favorite programming language. In fact, the distinction between design and program gets blurred since final code may get generated from the UML designs. An additional observation is that some program development methods suggest a prototyping approach where the system is incrementally constructed using a real programming language, rather than being derived from a pre constructed design. This was for example the case with the Remote Agent mentioned above. Furthermore, any research result on programming languages can benefit design verification since designs typically are less complex.

A third and very different kind of argument for studying verification of real programs is that such research will force the community to deal with very hard problems, and this may drive the research into new areas. We believe for example that it could be advantageous for formal methods to be combined with other research fields that traditionally have been more focused on programs, such as program analysis and testing. Such techniques are typically less complete, but they often scale better. Objective of formal methods is not only to prove programs correct, but also to debug programs and locate errors. With such a more limited ambition, one may be able to apply techniques which are less complete and based on heuristics, such as certain testing techniques.

Fourth, studying formal methods for programming languages may furthermore have some derived advantages for the formal methods community due to the fact that there is a tendency to standardize programming languages. This may make it feasible to compare and integrate different tools working on the same language - or on "clean subsets" of these languages. As mentioned above, it would be very useful to study the relationship between formal methods and other areas such as program analysis and testing techniques. Working at the level of programs will make it possible to better interact with these communities. A final derived advantage will be the many orders of magnitude increased access to real examples and users who may want to experiment with the techniques produced. This may have a very important impact on driving the research towards scalable solutions.

In general, it is our hope that formal methods will play a role for everyday software developers. By focusing on real programming languages community will be able to interact more intensively on solving common problems. Furthermore, the technology transfer problem so often mentioned may vanish, and instead be replaced by a technology demand.

3 MODEL CHECKING JAVA PROGRAMS

3.1 Complexity of Language Constructs

Input languages for model checkers are often kept relatively simple to allow efficient processing during model checking. Of course there are exceptions to this, for example Promela, the input notation of Spin [3], more resembles a programming language than a modeling language. General programming languages, however, contain many new features almost never seen in model checking input languages, for example, classes, dynamic memory allocation, exceptions, floating point numbers, method calls, etc. How will these be treated? Three solutions are currently being pursued by different groups trying to model check Java: one can translate the new features to existing ones, one can create a model checker that can handle these new features, or, one can use a combination of translation and a new/extended model checker.

3.1.1 Translation

The first version of JPF, as well as the JCAT system [4], was based on a translation from Java to Promela. Although both these systems were successful in model checking some interesting Java programs [5], such source-to-source translations suffer from two serious drawbacks:

Language Coverage — each language feature of the source language must have a corresponding feature in the destination language. This is not true of Java and Promela, since Promela for example, does not support floating point numbers.

Source Required — In order to translate one source to another, the original source is required, which is often not the case for Java, since only the byte codes are available — for example in the case of the libraries and code loaded over the WWW. For Java, the requirement that the source exists can be overcome by translating directly from byte codes. This is the approach used by the BANDERA tool [6], where byte codes, after some manipulation, are translated to either Promela or the SMV model checker's input notation. The Stanford Java model checker also uses this approach, by translating byte codes to the SAL intermediate language for model checking. Their SAL model checker is however specifically developed for the purpose of checking programs with dynamic data-structures and hence could be argued to fall into the custom-made model checker category below.

3.1.2 Custom-Made Model Checker

In order to overcome the language coverage problem it is obvious that either the current model checkers need to be extended or a new custom-made model checker must be developed. Some work is being done on extending the Spin model checker to handle dynamic memory allocation [7], but again in terms of Java this only covers a part of the language and much more is required before full Java language coverage will be achieved this way. With JPF we took the other route, we developed our own custom-made model checker that can execute all the byte code instructions, and hence allow the whole of Java to be model checked. The model checker consists of our own Java Virtual Machine (JVMJPF) that executes the byte codes and a search component that guides the execution. Note that the model checker is therefore an explicit state model checker, similar to Spin, rather than a symbolic one based on Binary Decision Diagrams such as SMV. Also, we decided that a depth-first traversal with backtracking would be most appropriate for checking temporal liveness properties (breadth-first liveness checking is inefficient due to the problems in detecting cycles). A nice side-effect of developing our own model checker was the ease with which we are able to extend the model checker with interesting new search algorithms—this would, in general, not have been easy to achieve with existing model checkers (especially not with Spin). A major design decision for JPF was to make it as modular and understandable to others as possible, but we sacrificed speed in the process—Spin is at least an order of magnitude faster than JPF. We believe this is a price worth paying in the long run. JPF is written in Java and uses the JavaClass package2 to manipulate class files.

3.2.1 Language and Properties Supported

The JVMJPF supports all Java byte codes, hence any program written in pure Java can be analyzed. Unfortunately, not all Java programs consist of pure Java code - one often finds that certain methods are defined as being native to the operating system. When a Java program calls methods that have no corresponding byte codes, then JPF cannot determine what the state of these code fragments will be and hence cannot handle programs that, for example, access the file system (user-defined class-loaders, file I/O operations, etc.), or communicate over a network, contains GUI code, etc. Fortunately, many native methods do not have side-effects and hence simple wrapper-methods can be written that translate the inputs and outputs to the native method, which then allow the original method to be called and all state changes to happen after returning from the call. However, if the native method itself causes an error, JPF will not be able to detect it, unless its output also causes an error in the Java code. Furthermore, JPF can only handle closed systems, i.e. a system and the environment it will execute in. This however is also the case in testing, where a test-harness is required to close a system, and is not considered a drawback of the approach. The current model checker can check for deadlocks, invariants and user defined assertions in the code, as well as Linear Time Temporal Logic (LTL) properties. In fact JPF supports all properties expressible in the BANDERA tool, the interested reader is referred to for more detail.

3.2 Complex States

In order to ensure termination during explicit state model checking one must know when a state is revisited. It is common for a hash table to be used to store states, which means an efficient hash function is required as well as fast state comparison. The Verisoft system [8] was developed to model check software, but the design premise was that the state of a software system is too complex to be encoded efficiently, hence Verisoft does not store any of the states it visits (Verisoft limits the depth of the search to get around the termination problem mentioned above). Since the Verisoft system executes the actual code (C/C++), and has little control over the execution, except for some user-defined "hooks" into communication statements, it is almost impossible to encode the system state efficiently. This insight also convinced us that we cannot tie our model checking algorithm in with an existing JVM, which is in general highly optimized for speed, but will not allow the memory to be encoded easily. Design philosophy was to keep the states of the JVM in a complex data-structure, but one that would allow us to encode the states in an efficient fashion in order to determine if we have visited states before. Specifically, each state consists of three components: information for each thread in the Java program, the static variables (in classes) and the dynamic variables (in objects) in the system. The information for each thread consists of a stack of frames, one for each method called, whereas the static and dynamic information consists of information about the locks for the classes/objects and the fields in the classes/objects. Each of the components mentioned above is a Java data-structure. In early stages of JPF development we did store these structures directly in a hash table, but with terrible results in terms of memory and speed: 512Mb would be exhausted after only storing _50000 states, and _20 states could be evaluated each second. The solution adopted to make the storing of states more efficient, was a generalization of the Collapse method from Spin, each component of the JVM state is stored separately in a table, and the index at which the component is stored is then used to represent the component. More specifically, each component (for example the fields in a class/object) is inserted in a table for that component; if the specific component is already in the table its index is returned, and if it is unique it is stored at the next open slot and that index is returned. This has the effect of encoding a large structure into no more than an integer3 (see Figure 1). Collapsing states in this fashion allows fast state comparisons, since only the indexes need to be compared and not the structures themselves. The philosophy behind the collapsing scheme is that although many states can be visited by a program the underlying components of many of these states will be the same. A somewhat trivial example of this is when a statement updates a local variable within a method. The only part of the system that changes is the frame representing the method; all the other parts of the system state are unaffected and will collapse to the same indexes. This actually alludes to the other optimization we added: only update the part of the system that changes, i.e., keep the indexes calculated for the previous state the same, only calculate the one that changed (to date we have only done this optimization in some parts of the system). After making these changes the system could store millions of states in 512Mb and could evaluate between 500 and 1500 states per second depending on the size of the state.

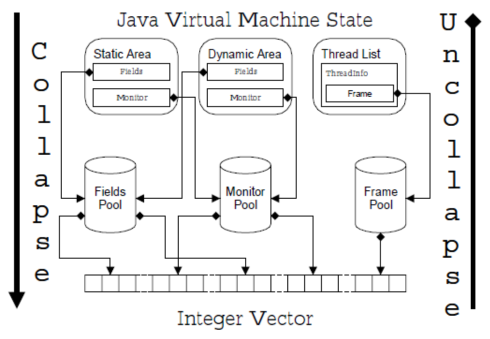

Figure 1. Collapsing and Recreating the JVM state

It was however clear from profiling the system execution that there was still one major source of inefficiency - the collapsing of states was only used for the states stored in the hash table, but in order to allow backtracking the un-collapsed states are stored in a stack. More specifically, whenever a new state is generated a copy of this state is made and put on the stack, during backtracking this state is removed again and execution continues. The Java "clone" operation is used to make copies of states, but this operation is notoriously slow since our states are represented by such a complex data-structure. Memory consumption was also high due to the complexity of each state, and we could seldom analyze a system with more than 10000 states in a depth-first path. A very simple, and above all novel solution, however presented itself: use the reverse of the collapse operation to recreate a state from its collapsed description (see Figure 1). We could now use the collapsed state description in both the hash table and the stack, and during backtracking the collapsed state is uncompressed by reversing the lookup in the tables (i.e. use the index to retrieve the original object from the table). This saves time since recreating the state from its collapsed form is faster than copying the state, and also saves memory since we now only create one collapsed copy of the state, which is stored in the hash table, and we keep a reference to this state in a stack entry. Lastly, as before, since only part of the state changes during each transition we can also just un collapse the parts that changed during backtracking. These last changes improved memory usage 4 fold and the model checker can now evaluate between 6000 and 10000 states per second depending on the size of the state.

3.3 Curbing The State Space Explosion

Maybe the most challenging part of model checking is reducing the size of the state space to be explored to something that your tool can handle. Since designs often contain less detail than implementations, model checking is often thought of as a technique that is best applied to designs, rather than implementations. We believe that applying model checking by itself to programs will not scale to programs of much more than 10000 lines. The avenue we are pursuing is to augment model checking with information gathered from other techniques in order to handle large programs. Specifically, we are investigating the use of symmetry reductions, abstract interpretation, static analysis and runtime analysis to allow more efficient model checking of Java programs. Figure 2 illustrates the architecture of JPF and its companion tools (abstraction tool, static analyzer and runtime analyzer) that will be described in detail below.

3.3.1 Symmetry Reductions

The main idea behind symmetry reductions [9] is that symmetries induce an equivalence relation on states of the system, and while performing analysis of the state space (for example during model checking) one can discard a state if an equivalent state has already been explored. Typically a canonicalization function is used to map each state into a unique representative of the equivalence class. Software programs can in general induce a great many symmetries, but here we will focus on a number of symmetry related problems found when analyzing Java programs: class loading and two forms of symmetry in the heap (dynamic area).

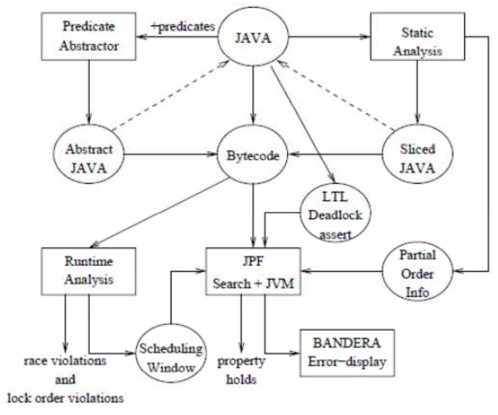

Figure 2. The JPF Tool Architecture. Dotted lines indicate iterative analysis.

The problem we are trying to avoid is the analysis of states that are equivalent to previously analyzed states. Java programs have dynamic behavior and one cannot predict which classes will be loaded, objects will be instantiated, or even in which order these will occur. This lack of order would seem to suggest an appropriate representation for the static area (where static variables for each class are stored) and the dynamic area (where objects are allocated) should be as sets. Comparing sets is however too time consuming, but an obvious ordering can be used, namely, the order in which classes are loaded or objects created. This however means that states will be considered to be different, if their only difference is the order of class loading (similarly if the same objects are placed in different locations in the dynamic area). What is required is to ensure that the static area and dynamic area have a canonical representation regardless of which interleaving of transitions is being executed. A canonicalization function for the static area is simple to define, since we can order the locations where the static variables of a class will be in the static area by ordering the class names. For each class loader in Java the class names must be unique, and since we do not consider the case of more than one class loader being used a simple mapping of class names to positions in the static area is enough. For example, if class A is loaded before class B in one interleaving then the static variables for class A will be stored at position 0 in the static area, and this mapping A → 0 will be remembered, when class B is loaded the mapping B → 1 will be remembered. After backtracking let us assume class B is now loaded before A, then the mapping for B will be recalled and B's static variables will be loaded at position 1 even though position 0 is available (class A's static variables will be loaded there).

Unfortunately, a similar approach with object allocation in the dynamic area is not sufficient since there can be many objects instantiated from the same class. One can however identify each object allocation in a Java program by uniquely identifying each "NEW" byte code4. This is not yet sufficient to define a mapping, since the same "NEW" can be executed more than once, for example when an allocation is in a loop. An occurrence number that is incremented each time the new is executed and decremented whenever the instruction is backtracked over, can then be used to identify each allocation.

Although the combination of the new-identifier and an occurrence number will distinguish many cases where there is symmetry, it does not resolve all cases. For example if the same allocation code can be executed from two different threads the symmetry reduction will be missed and equivalent states will be considered different. A thread reference can be added to distinguish this case. Clearly there is a trade-off between the precision of the canonicalization function and the time taken to calculate it; we chose to rely only on the new-identifier and the occurrence number in our current system.

3.3.2 Abstraction

Recently, the use of abstraction algorithms based on the theory of abstract interpretation, has received much attention in the model checking community. The basic idea underlying all of these is that the user specifies an abstraction function for certain parts of the data-domain of a system. The model checking system then, by using decision procedures, either automatically generates, on-the fly during model checking, a state-graph over the abstract data or automatically generates an abstract system, that manipulates the abstract data, which can then be model checked. The trade-off between the two techniques is that the generation of the state-graph can be more precise, but at the price of calling the decision procedures throughout the model checking process, whereas the generation of the abstract system requires the decision procedures to be called proportionally to the size of the program. It has been our experience that abstractions are often defined over small parts of the program, within one class or over a small group of classes, hence we favor the generation of abstract programs, rather than the on-the-fly generation of abstract state-graphs. Also, it is unclear whether the abstract state-graph approach will scale to systems with more than a few thousand states, due to the time overhead incurred by calling the decision procedures. Specifically we have developed an abstraction tool for Java that takes as input a Java program annotated with user-defined predicates and, by using the Stanford Validity Checker (SVC), generates another Java program that operates on the abstract predicates. For example, if a program contains the statement x++ and we are interested in abstracting over the predicate x==0, written as Abstract.addBoolean ("B", x == 0), then the increment statement will be abstracted to the code: "if (B) then B = false else B = Verify.randomBool ()" where nondeterministic choice is indicated by the randomBool () method that gets trapped by the model checker. The BANDERA tool uses similar techniques to abstract the data-domains of, for example, an integer variable to work over the abstract domains positive, negative and zero (the so-called sign abstraction), by using the PVS model checker. The novelty of our approach lies in the fact that we can abstract predicates over more than one class: for example, we can specify a predicate Abstract.addBoolean ("xGTy", A.x > B.y) if class A has a field x and class B has a field y. The abstracted code allows for many instantiations of objects of class A and B to be handled correctly. Although our Java abstraction tool is still under development we have had very encouraging results. For example we can, in a matter of seconds, abstract the omnipresent infinite-state Bakery algorithm written in Java to one that is finite-state and can be checked exhaustively. Abstractions for model checking often over-approximate the behavior of the system, in other words, the abstracted system has as a subset the behaviors of the original system. Since the properties that are typically checked are universally quantified over all paths, over-approximations preserve correctness. If a property holds in the abstracted system it is also true of the original system. Unfortunately, when it comes to model checking programs, or any other type of system for that matter, it is often the case that we are interested in finding errors, not showing correctness. And here lies a problem:-

Over-approximations do not preserve errors, i.e. errors in the abstract system might be due to new behaviors that were added and are not present in the original system. Eliminating these spurious errors is an active research area. We adopted a pragmatic approach to this problem that seems to work very well in practice. The basic idea is as follows: Any path in the abstracted program that is free of nondeterministic choices is also a path of the original program; hence if an error occurs on such a "choose-free" path then it is not spurious. JPF has a special mode in which it searches for errors only on paths that are choose-free — since nondeterminism in JPF is trapped by recognizing special method calls; it is easy to truncate a search whenever such a call occurs. Of course, if no error is found in this special mode, then the result is inconclusive since an error might exist, but the abstraction is not adequate to find the error in the choose-free mode. The next step is now to look for errors that may contain nondeterministic choices, if such an error exists, we can run this path in a simulation mode on the original program (there is a 1-to-1 mapping of code from the abstract to the original code) and if it diverges, i.e. the abstract path says statement s1 should be executed but the concrete program says s2 should be executed, then we can use the last decision point taken before divergence to refine the abstraction. If the path does not diverge we can also be sure that the error is not spurious. Note that we do not need to symbolically execute the abstract path on the concrete program, since the Java programs we check are by definition closed systems, i.e. they take no unknown input, and also each program has a single initial state.

3.3.3 Static Analysis

Static analysis of programs consists of analyzing programs without executing them. In general, the analysis is performed without making assumptions about the inputs of the program. The analysis results are therefore valid for any set of inputs. A wide variety of techniques fall under the static analysis umbrella; e.g., data flow analysis, set and constraint resolution, abstract interpretation, and theorem proving can all be applied to static analysis problems (with various degrees of success). They all derive some properties about a program. These properties are then used in slicing, code optimization, code parallelization, abstract debugging, code verification, code understanding, or code re-engineering for examples. The main aim of static analysis lies in its potential for reducing the size of the state space generated by a program. Therefore, focus is on three static analysis problems that can result in state space reduction: static slicing, partial evaluation, and partial order computation. Static slicing takes a program and a slicing criterion and generates a smaller program that is functionally equivalent to the original program with regard to the criterion. Partial evaluation propagates constant values and simplifies expressions in the process. Partial order computation focuses on identifying statements that can be safely interleaved with any statement on a different thread. The combined use of these analyses results in smaller state spaces, and therefore, helps reduce the state explosion problem. However, they do it in different manners. On the one hand, static slicing and partial evaluation generate a (functionally equivalent) smaller program that results in a smaller state space as shown in Figure 3. Black states indicate states that directly affect the slicing criterion (e.g., because they modify a variable involved in a property we want to check). After slicing, only the states affecting the slicing criterion remain in the state space. On the other hand, partial order computation does not change the size of the program, but its results can be used to further reduce the state space by eliminating unnecessary interleavings. Static slicing and its application in model checking is discussed ahead also partial order computation approach in brief.

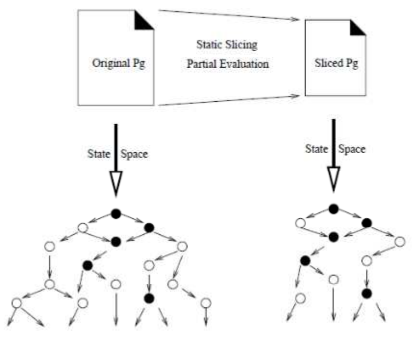

Figure 3 Reduction of programs using static slicing

One approach to reducing the size of programs, and therefore the size of the state space to be model checked, is to eliminate statements that are not relevant to the property one wants to verify. In static analysis, this process is known as program slicing. It has been studied quite extensively and the interested reader can find a detailed survey on slicing in. In general, a program slice is defined by the parts of a program that may affect (or be affected by) a slicing criterion. Typically a slicing criterion consists of a set of program points of interest. The sliced program is smaller than the original program and is functionally equivalent with respect to the slicing criterion. In this paper, we focus on works that use slicing as a program reduction tool for model checking as shown in. When slicing for model checking, criteria are often related to the properties that one wants to check, e.g., for a given property P, the slicing criterion is the set of program points affecting the values of the variables present in P. Therefore, every statement affecting the slicing criterion should be present in the slice (or sliced program); otherwise, the resulting program is not functionally equivalent to the original program. If such a statement was missing from the slice, it could result in a situation where the model checker states that a property holds on the sliced program even though it does not hold on the original program. This type of slicing is called closure slicing: a closure slice of a program P with respect to program point p and variable x consists of all statements that may affect the value of x at p. JPF uses the slicing tool of the BANDERA toolset which implements the work of Hatcliff et al. on static slicing of concurrent Java programs. Their technique consists of computing a set of program dependencies affecting the slicing criteria. These dependencies include the traditional dependencies (data, control and divergence) for sequential programs as well as their counterparts (interference, synchronization and ready dependencies) for concurrent programs. Informally, interference dependencies represent cases where the definition of shared variables can reach across threads. Synchronization dependence focuses on the use of synchronize statements; it basically states that if a variable is defined at a node inside some critical region, then the locking associated with that region must be preserved (i.e., the inner-most enclosing synchronize statement must be present in the slice). Ready dependence states that a statement n is dependent on a statement m if m’s failure to complete (e.g., because a wait or notify never occurs) can block the thread containing n. In BANDERA, slicing is not performed on the

Java source code, but on its (3-address code) representation called Jimple (Jimple is an intermediate representation for Java used in the Soot compiler developed at McGill University). In BANDERA, Jimple code is then translated into Promela or SMV code and then model checked. In order to use slicing and abstraction iteratively, and, since abstraction works on the source code level, we have to convert the sliced Jimple program back to Java source code using annotations that describe the original Java program. This approach has benefited JPF in several ways. First, using BANDERA, we can extract slicing criteria (i.e., program points) automatically from the properties verified by JPF. Second, BANDERA also provides support for partial symbolic evaluation, which yields smaller state spaces.

Third, we can re-use the dependence analysis performed by BANDERA to compute partial order information. Within JPF, static analysis is also used to determine which Java statements in a thread are independent of statements in other threads that can execute concurrently. This information is then used to guide the partial-order reductions built into JPF. Partial-order reduction techniques ensure that only one interleaving of independent statements is executed within the model checker. It is well established from experience with the Spin model checker that partial-order reductions achieve an enormous state-space reduction in almost all cases. We have had similar experience with JPF, where switching on partial-order reductions caused model checking runs that ran for hours to finish within minutes. We believe model checking of (Java) programs will not be tractable in general if partial-order reductions are not supported by the model checker and in order to calculate the independence relations required to implement the reductions, static analysis is required.

3.3.4 Runtime Analysis

Runtime analysis is conceptually based on the idea of executing a program once, and observing the generated execution trace to extract various kinds of information. This information can then be used to predict whether other different execution traces may violate some properties of interest (in addition of course to demonstrating whether the generated trace violates such properties). The important observation here is that the generated execution trace itself does not have to violate these properties in order for their potential violation in other traces to be detected. Runtime analysis algorithms typically will not guarantee that errors are found since they after all work on a single arbitrary trace. They also may yield false positives in the sense that analysis results indicate warnings rather than hard error messages. What is attractive about such algorithms is, however, that they scale very well, and that they often catch the problems they are designed to catch. That is, the randomness in the choice of run does not seem to imply a similar randomness in the analysis results. In practice runtime analysis algorithms will not store the entire execution trace, but will maintain some selected information about the past, and either do analysis of this information on-the-fly, or after program termination. An example is the data race detection algorithm Eraser [10] developed at Compaq, and implemented for C++ in the Visual Threads tool (Harrow, 2000). Another example is a locking order analysis called Lock-Tree which we have developed. Both these algorithms have been implemented in JPF to work on Java programs. Below we describe these two algorithms, and how they can be run stand-alone in JPF to identify data race and deadlock potentials in Java programs. Then we describe how these algorithms are used to focus the model checker on part of the state space that contains these potential data race and deadlock problems. Note that runtime analysis is different from runtime monitoring, as supported in systems such as Temporal Rover and Mac, where certain user-specified properties are monitored during execution. We are, however, currently also exploring the integration of this kind of technology with runtime analysis.

Data Race Detection

The Eraser algorithm detects data race potentials. A concrete data race occurs when two concurrent threads simultaneously access a shared variable and when at least one access is a write; hence the threads use no explicit mechanism to prevent the accesses from being simultaneous. The program is guaranteed data race free if for every variable there is a nonempty set of locks that all threads own when they access the variable. The Eraser algorithm can detect that a data race on a variable is possible (potential) even though no concrete data races have occurred, by observing and remembering which locks are active whenever it is accessed. The algorithm works by maintaining for each variable x a set set(x) of those locks active when threads access the variable. Furthermore, for each thread t a set (t) is maintained of those locks taken by the thread at any time. Whenever a thread t accesses the variable x, the set set(x) is refined to the intersection between set(x) and set (t) (set(x):= set(x) \ set (t)), although the first access just assigns set (t) to set(x). Our algorithm differs from [10] since there the initial value of set(x) is the set of all locks in the program. In a Java program objects (and thereby locks) are generated dynamically, hence the set of all locks cannot be pre-calculated. A race condition may be possible if set(x) ever becomes empty. First of all, shared variables are often initialized without the initializing thread holding any locks. The above algorithm will yield a warning in this case, although this situation is safe. Another situation where the above algorithm yields unnecessary warnings is if a thread creates an object, where after several other threads read the object's variables (but no-one is writing after the initialization). Figure 4 illustrates this state machine. The variable starts in the VIRGIN state. Upon the first write access to the variable, the EXCLUSIVE state is entered. The lock set of the variable is not refined at this point. This allows for initialization without locks. Upon a read access by another thread, the SHARED state is entered, now with the lock refinement switched on, but without yielding warnings in case the lock set goes empty. This allows for multiple readers (and not writers) after the initialization phase. Finally, if a new thread writes to the variable, the SHARED-MODIFIED state is entered, and now lock refinements are followed by warnings if the lock set becomes empty.

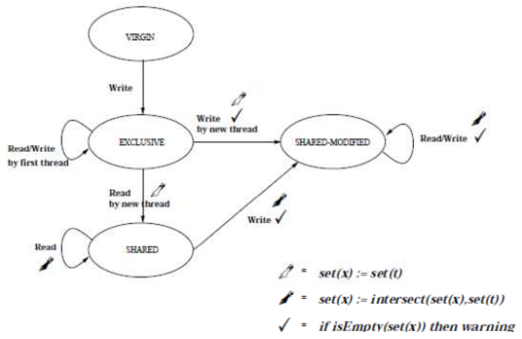

Figure 4. The Eraser algorithm associates a state machine with each variable x.

The state machine describes the Eraser analysis performed upon access by any thread t. The pen heads signify that lock set refinement is turned on. The p sign signifies that warnings are issued if the lock set becomes empty.

The generic Eraser algorithm has been implemented to work on Java by modifying the JVMJPF to perform this analysis when the eraser option is switched on. Each thread is associated with a lock set (a Java object representing a set), and each variable (field) in each object is associated with an automata of the type shown in Figure 4 (a Java object representing the automata and lock set). The JVMJPF accesses the byte codes via the JavaClass package, which for each byte code delivers a Java object of a class specific for that byte code. The JVMJPF extends this class with an execute method, which is called by the verification engine, and which represents the semantics of the byte code. The runtime analysis is obtained by instrumenting the execute methods of selected byte codes, such as the GETFIELD and PUTFIELD byte codes that read and write object fields, the static field access byte codes GETSTATIC and PUTSTATIC, and all array accessing byte codes such as for example IALOAD and IASTORE. The byte codes MONITORENTER and MONITOREXIT, generated from explicit synchronized statements, are instrumented with updates of the lock sets of the accessing threads to record which locks are owned by the threads at any time; just as are the byte codes INVOKEVIRTUAL and INVOKESTATIC for calling synchronized methods. The INVOKEVIRTUAL byte code is also instrumented to deal with the built-in wait method, which causes the calling thread to release the lock on the object the method is called on. Instrumentations are furthermore made of byte codes like RETURN for returning from synchronized methods, and ATRHOW that may cause exceptions to be thrown within synchronized contexts.

Deadlock Detection

A classical deadlock situation can occur where two threads share two locks and attempt to take the locks in different order. An algorithm that detects such lock cycles must in addition take into account that a third lock may protect against a deadlock like the one above, if this lock is taken as the first thing by both threads, before any of the other two locks are taken. In this situation no warnings should be emitted. Such a protecting third lock is called a gate lock. The algorithm for detecting this situation is based on the idea of recording the locking pattern for each thread during runtime as a lock tree, and then, when the program is terminated, comparing the trees for each pair of threads. The lock tree that is recorded for a thread represents the nested pattern in which locks are taken by the thread. As an artificial example, consider the code fragments of two threads in Figure 5. Each thread takes four locks L1, L2, L3 and L4 in a certain pattern. For example, the first thread takes L1; then L3; then L2; then it releases L2; then takes L4; then releases L4; then releases L3; then releases L1; then takes L4; etc.

```
Thread 1:                           Thread 2:
synchronized(L1){                   synchronized(L1){
  synchronized(L3){                   synchronizd(L2){
    synchronized(L2){};                 synchronized(L3){}
    synchronized(L4){}                }
  }                                 };
};
synchronized(L4){                   synchronized(L4){
  synchronized(L2){                   synchronized(L3){
    synchronized(L3){}                  synchronized(L2){}
  }                                   }
}                                   }
```

Figure 5 Synchronization behavior of two threads

This pattern can be observed, and recorded in a finite tree of locks for each thread, as shown in Figure 6, by just running the program. As can be seen from the trees, a deadlock is potential because thread 1 in its left branch locks L3 (node identified with 2) and then L4 (4), while thread 2 in its right branch takes these locks in the opposite order (11, 12). There are furthermore two additional ordering problems between L2 and L3, one in the two left branches (2, 3 and 9, 10), and one in the two right branches (6, 7 and 12, 13). However, neither of these pose a deadlock problem since they are protected by the gate locks L1 (1, 8) respectively L4 (5, 11). Hence, one warning should be issued.

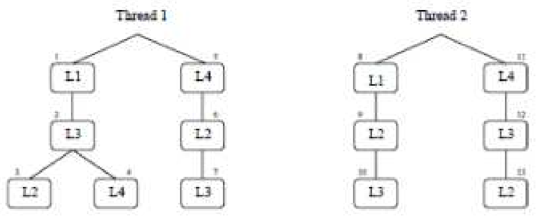

Figure 6. Lock trees corresponding to threads in Figure 5.

When being built, each tree has at any time a current node, where the path from the root (identifying the thread) to that node represents the lock nesting at this point in the execution. The lock operation creates a new child of the current node if the new lock has not previously been taken (is not in the path above). The unlock operation just backs up the tree if the lock really is released, and not owned by the thread in some other way. When the program terminates, the analysis of the lock trees is initiated. Each pair of trees (t1; t2) are compared, and for every node n in t1 it is checked that no node below n is above any occurrence of n in t2. In order to avoid issuing warnings when a gate lock prevents a deadlock, occurrences of n in t2 are marked after being examined, and nodes below marked nodes are not considered until the marks are removed when the analysis backtracks from the corresponding node in t1. The following byte codes will activate calls of the lock and unlock operations in these tree objects for the relevant threads: MONITORENTER and MONITOREXIT for entering and exiting monitors, INVOKEVIRTUAL and INVOKESTATIC for calling synchronized methods or the built-in wait method of the Java threading library, byte codes like RETURN for returning from synchronized methods, and ATRHOW that may cause exceptions to be thrown within synchronized contexts.

4 USING RUNTIME ANALYSIS TO GUIDE MODEL CHECKING

The runtime analysis algorithms described in the previous two sections can provide useful information to a programmer as standalone tools. In this section we will describe how runtime analysis furthermore can be used to guide a model checker. The basic idea is to run the program in simulation mode first, using the JVMJPF simulator, with all the runtime analysis options turned on, thereby obtaining a set of warnings about data races and lock order conflicts. The threads causing the warnings are stored in a race window. When the simulation is terminated, forced or according to the program logic, the resulting race window (in fact an extension of it, see below) will then be fed into the model checker, which will now search the state space, but now only focusing its attention on the threads in the window. That is, the model checker only schedules threads that are in the window. However, before the model checker is applied, the race window is extended to include threads that create or otherwise influence the threads in the original window. The purpose is to obtain a small self-contained sub-system containing the race window, which can be meaningfully model checked. The extended window can be thought of as a dynamic slice of the program. The extension is calculated on the basis of a dependency graph, created by a dependency analysis also performed during the pre-simulation. More specifically, the dependency graph is a mapping from threads t to triples (α, ρ, ω), where α is the ancestor thread that spawned t, ρ is the set of objects that t reads from, and ω is the set of objects that t writes to. The window extension operation performs a fix-point calculation by creating the set of all threads reachable from the original window by repeatedly including threads that have spawned threads in the window, and by including threads that write to objects that are read by threads in the window. The following byte codes are instrumented to operate on the dependency graph: INVOKEVIRTUAL for invoking the start method on a thread; and PUTFIELD, GETFIELD, PUTSTATIC, GETSTATIC for accessing variables.

5 CONCLUSION

The second part of the paper described how we applied this philosophy to the analysis of Java programs. Specifically, we have shown that model checking can be applied to Java programs, without being hampered by the perceived problems often cited as reasons for why model checking source code will not work. In the process we have shown that augmenting model checking with symmetry reductions, abstract interpretation, static analysis and runtime analysis can lead to the efficient analysis of complex (Java) software. Although the combination of some of these techniques is not new, to the best of our knowledge, our use of symmetry reductions for class loading and heap allocation, the semi-automatic predicate abstraction across different classes, the use of static analysis to support partial-order reductions and the use of runtime analysis to support model checking are all novel contributions. Although it is hard to quantify the exact size of program that JPF can currently handle – "small" programs might have "large" state-spaces A nice side-effect of developing our own model checker was the ease with which we are able to extend the. model checker with interesting new search algorithms-this would, in general, not have been easy to achieve with existing model checkers (especially not with Spin). A major design decision for JPF was to make it as modular and understandable to others as possible, but we sacrificed speed in the process - Spin is at least an order of magnitude faster than JPF.

REFERENCES

[1] Booch, G., J. Rumbaugh, and I. Jacobson: 1999, *The Unified Modeling Language User Guide*. Addison-Wesley.

[2] Havelund, K., M. Lowry, and J. Penix: 1998, 'Formal Analysis of a Space Craft Controller using SPIN'. In: *Proceedings of the 4th SPIN workshop, Paris, France*.

[3] Holzmann, G.: 1997b, 'The Model Checker Spin'. *IEEE Trans. on Software Engineering*

[4] Demartini, C., R. Iosif, and R. Sisto: 1999a, 'A Deadlock Detection Tool for Concurrent Java Programs'. *Software Practice and Experience*

[5] Havelund, K. and J. Skakkebaek: 1999, 'Practical Application of Model Checking in Software Verification'. In: *Proceedings of the 6th Workshop on the SPIN Verification System,*

[6] Corbett, J. C., M. B. Dwyer, J. Hatcliff, and Robby: 2000b, 'A Language Framework For Expressing Checkable Properties of Dynamic Software'.

[7] Visser,W.,K. Havelund, and J. Penix: 1999, 'Adding Active Objects to SPIN'.

[8] Godefroid, P.: 1997, 'Model Checking for Programming Languages using VeriSoft'. In: *Proceedings*
of the 24th ACM Symposium on Principles of Programming Languages.

[9] Clarke, E., E. Emerson, S. Jha, and A. Sistla: 1998, 'Symmetry Reductions in Model Checking'. In: *Proceedings of the 10th International Conference for Computer-Aided Verification*.

[10] Savage, S., M. Burrows, G. Nelson, and P. Sobalvarro: 1997, 'Eraser: A Dynamic Data Race Detector for Multithreaded Programs'. *ACM Transactions on Computer Systems*