Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code

Alexander von Rhein, Sven Apel
University of Passau
Passau, Germany
e-mail: {rhein,apel}@fim.uni-passau.de

Franco Raimondi
Middlesex University
London, UK
e-mail: f.raimondi@mdx.ac.uk

Abstract

One of the big performance problems of software model checking is the state-explosion problem. Various tools exist to tackle this problem. One of such tools is Java Pathfinder (JPF) an explicit-state model checker for Java code that has been used to verify efficiently a number of real applications.

We present `jpf-bdd`*, a JPF extension that allows users to annotate Boolean variables in the system under test to be managed using Binary Decision Diagrams (BDDs). Our tool partitions the program states of the system being verified and manages one part using BDDs. It maintains a formula for the values of these state partitions at every point during the verification. This allows us to merge states that would be kept distinct otherwise, thereby reducing the effect of the state-explosion problem.*

We demonstrate the performance improvement of our extension by means of three example programs including an implementation of the well-known dining-philosophers problem.

1. Introduction

Java Pathfinder (JPF) is a Java Virtual Machine that can be configured as a model checker for Java programs. In its basic configuration, JPF is a fast, explicit model checker that explores the whole state space of a system under test represented by a Java program, searching for unhandled exceptions, deadlocks, and data races. JPF executes the program instructions and creates a choice point if multiple execution paths can be considered, for instance, in the case of thread interleaving or in non-deterministic choices. Intuitively, each choice generates a so-called JPF state. Most programs contain so many possible states that the exploration of the whole state space is an extremely, computationally expensive task and is the main limiting factor in the analysis of systems using model checkers. JPF implements state-of-the-art optimization strategies for the exploration of large state spaces, such as state matching, on-the-fly partial-order reduction, and it can be extended with a number of other optimizations such as symbolic execution [1].

Beside improvements of explicit model checkers, a number of alternative techniques have been investigated to tackle the state-explosion problem, including the reduction of the model-checking problem to a satisfiability problem, the reduction to Boolean formulae to be manipulated using efficient data structures, etc. With slight abuse of notation, we call all these approaches "symbolic" when they do not treat states explicitly. Symbolic approaches have a computational cost in addition to the "standard" cost of state exploration, but in some cases they can dramatically condense the state space to be explored by merging sets of states in appropriate ways. Unfortunately, no approach has been shown to be superior to the others.

With `jpf-bdd`, we present a novel hybrid approach to model checking that allows the flexible use of both explicit and symbolic techniques in the same model checker. Our idea is to partition each program state in two parts. The first part of the program state, called *core part*, contains all the JPF kernel variables (program counters, etc.) and all the variables that are not identified with a special annotation. This part is handled in a standard way by JPF.

The second part contains the values of a user-defined set of Boolean variables (identified by a special annotation and called *tracked variables*); these variables are represented as Binary Decision Diagrams (BDDs). We call this second part of the program state the *BDD part* and the extension that handles this part `jpf-bdd`. For disambiguation, we call the original implementation `jpf-core`.

This separation of the program states in a core part and a BDD part allows JPF to perform a number of optimizations:

- choose values for tracked variables only if and when they are used;
- produce a condensed summary of values of the tracked variables at any point in the program;
- merge program states if they do differ only in the values of the tracked variables.

In a number of examples, the last optimization leads to a reduced state space and therefore to a faster verification process, as we show in Section 4. The choice of which variables should be treated by the BDD part is left to the user, thus providing a certain flexibility by exploiting domain knowledge that otherwise would not be accessible by the model checker.

Our implementation works with DFS and BFS based model checking. However with BFS the implemented optimizations have greater effect, as we explain in the following sections.

Our approach is inspired by the work of Classen et al. on symbolic model checking of product lines [2]. They use BDDs to manage variables that activate and deactivate individual features in Promela programs. We generalize their approach to arbitrary Boolean variables in Java programs by allowing users to explicitly annotate variables to be handled by BDDs (e.g., for handling lock variables efficiently).

The rest of the paper is organised as follows. In Section 2, we present background material on JPF and BDDs. In Section 3, we describe our approach and its implementation, and we provide experimental results in Section 4. We discuss related literature in Section 5, and we conclude in Section 6.

2. Background

2.1. Binary decision diagrams

In this section, we provide a short overview of Binary Decision Diagrams (BDDs). As an example, consider the Boolean formula $f(x, y, z) = (x \vee (\neg x \wedge y)) \wedge z$, in which x, y, z are Boolean variables. The truth table of this formula is eight lines long. Alternatively, one could represent the truth value of f by means of a diagram with root node x and two outgoing edges (one representing the value $true$ for x and the other the value $false$), each of which leading to a node y, and similarly with the last variable z. The eight leaves of this diagram represent the truth values of f. This diagram can be simplified (i.e., *reduced*) by merging redundant nodes and by removing redundant tests. Figure 1 shows the Reduced Ordered Binary Decision Diagram (ROBDD) representing f. Notice how the ROBDD for f only has five nodes (instead of fifteen for the non-reduced diagram). This reduction in size is one of the key factors for the efficient manipulation of Boolean formulas using ROBDDs.

If the order of variables is kept fixed (hence the "ordered" in the name), it is possible to combine ROBDDs of different formulae by means of Boolean operators in polynomial time. We refer to Bryant's seminal work [3] for more details on the reduction algorithm and for other operations on ROBDDs. In the remainder of the paper, we will use the term BDD instead of ROBDD, as all the diagrams are considered to be reduced and ordered.

Unfortunately, BDDs may still have an exponential size in the number of Boolean variables used, and finding the best variable ordering is a NP-complete problem, and similarly for the operation of Boolean quantification. Nevertheless, it has been shown [4] that BDDs offer an efficient mechanism for the exploration of large state spaces in practice.

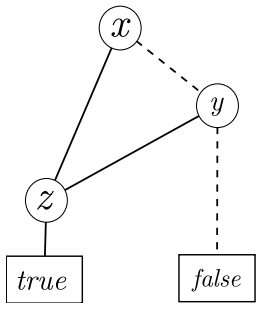


Figure 1. A BDD that represents the formula $(x \vee (\neg x \wedge y)) \wedge z$. The solid (dotted) lines represent the $true$ ($false$) values of the variables. All paths leading to the terminal $true$ are satisfying assignments for the formula.

2.2. Model checking

This section gives a brief overview of the model-checking techniques used in JPF. Model checking tools in general work by constructing and exploring a *reachability tree*. The nodes of the reachability tree represent program states that the verified program can have during the execution. These states include the current program counter (the currently executed statement). Edges in the tree represent the execution semantics of the program. A path in the tree corresponds to one execution of the program as would be done with simple program testing. In contrast to simple testing,

a	b	c	d
42	*false*	*true*	*true*
42	*false*	*true*	*false*
42	*false*	*false*	*true*
42	*false*	*false*	*false*

Table 1. Explicit state space

a model checker explores the entire reachability tree and therefore (theoretically) finds bugs in all possible execution paths. The model-checking tool generates this tree from the source code of the program to be verified. In the case of JPF, the model is given by the Java byte-code, so byte-code instructions are the examined statements.

Modern model checkers such as JPF simplify the reachability tree by merging paths that contain equal states. This saves the model checker from exploring equal sub-trees twice. It also transforms the reachability tree into a directed acyclic graph (DAG). For simplicity, we will still call the data structure reachability tree.

As the reachability tree is generated on-the-fly during verification, the order in which nodes are explored is important. Simple exploration strategies are depth-first search (DFS) and breadth-first search (BFS). JPF allows a user to configure both DFS and BFS as exploration strategies, with DFS being the default configuration. For `jpf-bdd`, we changed this default configuration to BFS because this makes our verification technique more efficient, as we will explain.

3. Implementation of `jpf-bdd`

3.1. Motivation

As a motivating example, consider the situation in which the state space of an application consists of an integer variable a and three Boolean variables b, c, and d. Suppose we need to represent the set of four states listed in Table 1, in which the a takes value 42, b is fixed to $false$, and the remaining variables take all their possible values.

If a BDD is employed to represent the three Boolean variables, the memory footprint of this state space can be reduced compared to the explicit variant in Table 1, as depicted in Figure 2.

In certain circumstances, the BDD representation can result in a dramatic compression of the state space, especially when the state space contains states that differ only for the values of Boolean variables. We provide examples in Section 4.

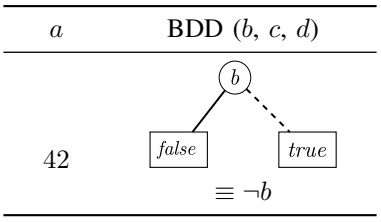


Figure 2. Hybrid state space. The variables b, c, and d are managed with a BDD. During BDD reduction, c and d have been eliminated because they do not influence the formula result. Again, the solid (dotted) line represents the $true$ ($false$) value of variable b. The reduced BDD represents the formula $(\neg b \wedge c \wedge d) \vee (\neg b \wedge c \wedge \neg d) \vee (\neg b \wedge \neg c \wedge d) \vee (\neg b \wedge \neg c \wedge \neg d) \equiv \neg b$.

3.2. Overview of `jpf-bdd`

`jpf-bdd` is implemented as a standard extension of `jpf-core`. We refer to the material available online for further details on `jpf-core`.[1] The source code and installation instructions for `jpf-bdd` are also available on-line[2]; once installed, the user needs to specify `jpf-bdd` as an extension when launching the verification. Additionally, the user needs to annotate the Boolean variables to be tracked by `jpf-bdd` with the annotation `@TrackWithBDD` (see the examples available in the source tree).

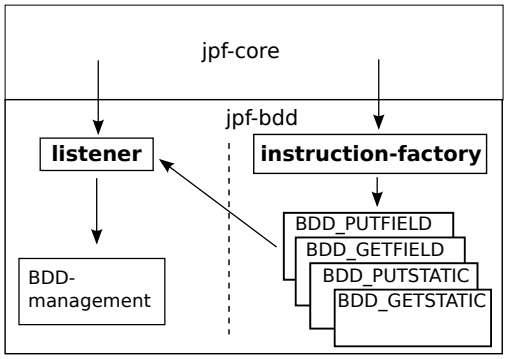


Figure 3. Extended JPF architecture

`jpf-bdd` consists of two main parts: a listener and a custom instruction factory, as illustrated in Figure 3. Furthermore, it uses a third-party BDD library (javabdd[3]), which is included in the source distribution of `jpf-bdd`.

The `jpf-bdd` listener implements the JPF's interface `SearchListener` and maintains the state of the

1. http://babelfish.arc.nasa.gov/trac/jpf/.
2. https://bitbucket.org/rhein/jpf-bdd/
3. http://javabdd.sourceforge.net/

tracked variables as the verification process evolves. The listener manages a mapping of BDDs to JPF search states. The BDD is updated according to the execution of statements in the verification process, and it holds information about the current values of the tracked variables (the BDD part of the state). After a backtracking operation to some JPF state, the BDD that was mapped to this state is reloaded as the currently active BDD. After an advance operation from JPF state x with BDD β_x to JPF state y with β_y, the new BDD of state y is set to $\beta_x \vee \beta_y$. We do not prohibit the merge of unequal states, but merge them into a state with a new BDD. We discuss this further in Section 3.3.

`jpf-bdd` uses special byte-code instructions to maintain information about the value of the tracked variables during the verification process. These special instructions are created by a dedicated instruction factory, which has to be configured in the verification tool. The instructions take action when the value of a tracked variable is accessed.

Accesses correspond to the byte-code instructions `GETSTATIC` and `PUTSTATIC` for static variables and to the instructions `GETFIELD` and `PUTFIELD` for instance variables [5]. In fact, only these instructions have to be considered by `jpf-bdd`: if an instruction does access a non-tracked variable then the handling of the statement is delegated to `jpf-core` (the core part of the state is updated). If an instruction references a tracked variable then our special instruction factory creates an instruction that works with the BDDs in the listener.

In the case an instruction queries the value of a variable (i.e., `GETSTATIC` or `GETFIELD`), we determine which values ($true$ or $false$) would be possible with the current BDD. If it is only one value, then that value will be loaded and normal execution resumes. If both values are possible, we introduce a Boolean choice generator that splits the execution path and forces exploration of both possibilities. The execution of `GETSTATIC` instructions is shown in Figure 4 (`GETFIELD` is treated similarly).

In the case of an assignment instruction (`PUTSTATIC` or `PUTFIELD`), we modify the BDD accordingly: we first remove the possible previous values of the variable using an existential quantification, and then add the new value to the formula.

As mentioned above, only four byte-code instructions need to be modified, and a different semantic is only necessary if these access one of the tracked variables. This means that the majority of the program verification remains unchanged.

There are two possible approaches to implement the

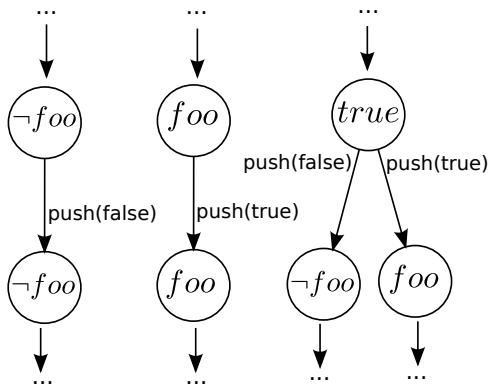


Figure 4. Execution of a GETSTATIC instruction on the variable foo with `jpf-bdd` on different starting states. Circles denote states partially represented as BDDs. The effect of the instruction is the value of foo ($true$ or $false$) being pushed on the stack and a transition to a (potentially) different state.

different semantics of the four byte-code instructions. The first approach is to register an instance of a listener class that is called on each byte-code instruction execution. If the instruction requires a changed semantics, then the listener overwrites the original semantics. Otherwise the listener does nothing. This approach means that a method in the listener class (the `executeInstruction` method) is called once for each instruction execution.

The second approach is to modify the semantics of the byte-code instructions in JPF by defining a special instruction factory. The factory inspects the byte-code instructions when JPF parses the program and, if necessary, replaces the generated byte-code objects by `jpf-bdd` byte-code objects that implement the changed semantics.

We chose to implement the second approach. This choice is motivated by the fact that method `executeInstruction` in the first approach would be invoked most of the time without any effect. These unnecessary method calls would have severe consequences for verification performance. As an example, we employed the dining-philosophers problem given in Section 4.3: for this medium sized program the `executeInstruction` method is called 21 million times when states are merged using `jpf-bdd`, and 76 million times when states are treated explicitly. Most of the times method `executeInstruction` of the first approach invokes only the super-class method because the byte-code does not require a different semantics, thus resulting in degraded performance of

jpf-bdd even when the state space is significantly reduced.

In the current implementation, we replaced the simple listener architecture with the more complex architecture described above, and only the relevant instructions are considered by jpf-bdd. This architecture, however, impacts the compatibility of jpf-bdd with other JPF extensions. In particular, jpf-bdd is currently incompatible with any other extension that needs a special byte-code factory or that tries to access the actual value of tracked variables in jpf-core[4].

3.3. Merging of states

The separation of program states into a core part and a BDD part allows us to treat these parts individually. This is important when it comes to state-equality tests. JPF uses unification of equal states in different program paths to merge these program paths. This results in a reduced number of program states and a reduced number of executed program paths. In turn, this reduction improves both the total verification runtime and the memory requirements of the verification process. For the purposes of our work, we exploit the fact that the state-equality test and the merging of program paths can be extended by overriding a method in JPF listeners. We are using this extension point to modify the equality test and merge program paths in the additional situation in which the core part of the state is equal and the BDD part is different.

An advance from state x to an already known state y means that jpf-core wants to unify the states because they are equal from its viewpoint. This equality does not take into account the values of the variables in the BDD part of the state, because jpf-core does not know of their values[5]. So jpf-core will merge the states x and y when the core parts of x and y are equal, regardless of the equality of the BDD part. We do not prevent the merge of these states but store $\beta_x \vee \beta_y$ as new BDD where β_x and β_y are the BDDs of the old states, respectively.

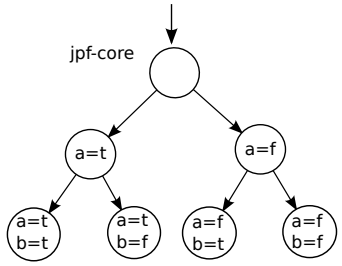
This state-merging implementation has two effects:

- It leads to the merging of states that are not equal. As a consequence, the reachability tree that is constructed is smaller than before and sub-trees do not need to be explored twice.[6]
- It sometimes leads to the elimination of variables from the BDD formula and therefore to a shorter formula (see the example at the beginning of this section).

4. jpf-bdd provides a function to get the current BDD formula that contains these values.

5. In our implementation, the values of the variables in jpf-core are set to constant false.

6. This depends on choosing BFS as tree exploration order.




```
Boolean a, b;

void test() {
  if (a=Verify.getBoolean()) {...}
  if (b=Verify.getBoolean()) {...}
}
```

```
@TrackWithBDD
Boolean a, b;
void test() {
  if (a) {...}
  if (b) {...}
}
```

Figure 5. Reachability trees generated by jpf-core and jpf-bdd on similar pieces of code. This example depends on the fact that the bodies of the if statements do not change the core part of the program state. Both functions are semantically equivalent. The state marked A'' is referred to in the text.

Figure 5 shows a very simple program that spans a reachability tree when being verified. The figure also shows how the reachability tree is modified by jpf-bdd. Both reachability trees are constructed during the verification. Figure 5 shows the trees after the verification is complete.

The verification using jpf-core introduces Boolean choice generators at both calls of Verify.getBoolean(). Because the method is called twice and the return value is saved in the program state, this results in four different states. If the program would not terminate here, then the rest of the program would have to be executed once for every one of these states.

The verification with jpf-bdd introduces a Boolean choice generator when the verification reaches the GETFIELD instruction on variable a. The first explored value is a = *true* and this leads to a new state A (the second state on the right side of the figure). Because we are using BFS-based verification the other value for a (a = *false*) is explored immediately afterwards. This leads to a state A' that has the same program counter as A, and every variable in the core-part of A' is equal to its counterpart in A. The only difference in these states is that the BDD part of A' has a = *false* and the BDD part of A has a = *true*. So these states are merged with the method

described above. The resulting state is A'' with the BDD part $\texttt{a} \vee \neg \texttt{a} = \mathit{true}$ (this state is shown in the figure). Afterwards, the same operation is done with the variable `b` in the next `if` statement. So, in the end, we have one result state. If the program would not terminate here, then the rest of the program would only have to be executed once.

The efficiency of this approach relies on the fact that we are using BFS-based reachability-tree exploration. In the given example, a state A is merged with state A' resulting in A' being replaced by state A''. If the BDD of A'' is different from the BDD of A', we have to re-explore the whole sub-tree of A' because the changed BDD might influence the program behavior in the sub-tree. If we would be using the DFS exploration strategy, this would result in many re-explorations and we would waste the key advantage of `jpf-bdd`.

Notice that the merge of states with `jpf-bdd` can result in the elimination of variables from the BDD, thus resulting in a more compact state-space representation. In the given example, the variable `a` is eliminated when the state A'' is set to true. In the following execution paths `a` is treated as "unknown" again (as at the start of the verification). Every merge of BDDs potentially leads to a reduction in the formula size and therefore makes the further analysis more efficient. This reduction of the Boolean formula is performed automatically by the BDD library.

Our changed state-merging algorithm does not violate the correctness of the model-checking algorithm because both merged paths are represented by the "new" path (and are therefore explored). If an "unknown" variable of the BDD part is used as condition somewhere along the new path, the path will be split again.

3.4. `jpf-bdd` and multi-threading

In this section, we provide an overview on the verification of multi-threaded programs with `jpf-core` and `jpf-bdd`. JPF is able to detect deadlocks and data races in Java programs. When verifying a multi-threaded program, naively one would need to execute every statement in the threads in every possible order. That means if no synchronization happens between two threads then every execution order on the (byte-code) instruction level has to be considered (which leads to a exponential explosion). JPF uses a technique called *partial-order reduction* to overcome this problem. It combines sets of instructions into *transitions*. Each transition ends with a *transition breaker*. These are instructions that might influence the behaviour of other threads and therefore the correctness of the program. This effectively reduces the practically incomputable problem to a practical size because only the order of complete transitions has to be considered.

The correctness of partial-order reduction relies on the reliable detection of transition breakers. Whether a statement is a transition breaker is determined during the verification at every time the instruction is executed. If a statement is identified as a transition breaker, a special choice generator is created that splits the execution path. Every choice schedules one of the active threads to be executed next. This method spans all possible execution orders of all transitions.

The verification process with `jpf-bdd` relies mainly on the existing process in `jpf-core`. So it was relatively easy to extend `jpf-bdd` for verification of multi-threaded programs. Every one of the four special BDD instructions (Section 3.2) is a potential transition breaker because the accessed variables might also be available to other threads. The algorithm used by `jpf-core` to identify transition breakers does not influence the variable value, only the current thread situation and the variable declaration. So we can reuse the implemented algorithm to determine whether the BDD instruction is a transition breaker and issue a thread-scheduling choice generator if appropriate. Afterwards, the normal execution of the BDD instruction resumes.

Consider, for example, a minimal multi-threaded program with two threads that share one tracked Boolean variable. The threads have only one statement each. One assigns the value true to the variable, the other thread assigns false. When `jpf-bdd` verifies this program each possible execution order is considered. Depending on the execution order, the variable (and its BDD value) is first true and then false or vice versa. When the program terminates, the core parts of all verification states are equal. So these states are merged into one state that has the BDD true.

3.5. Detection of errors

JPF is a tool for the detection of defects in Java programs. These defects can be detected by two means:

- Defects can be reported by listeners that inspect every generated program state and the path leading to these states.
- The system under test itself can throw Java exceptions which will be caught and reported by JPF.

`jpf-bdd` does not change the error-handling mechanisms in JPF, so the second method works as before. Whether a `jpf-bdd`-unaware listener does work

correctly depends on whether it uses values of the tracked variables. As mentioned above, we are modifying `jpf-core` to store a constant *false* for the tracked variables, while the real value of the tracked variables is stored in BDDs. If the listener queries the variable value from `jpf-core`, it will get the wrong value (*false*) and probably derive false conclusions. The real value can be found by fetching the current BDD from the `jpf-bdd` listener and examining it. However, we found that most listeners do not use concrete variable values but are more concerned with control flow and variable accesses. For example, the `PreciseRaceDetection` listener works as expected whether the race depends on a BDD-tracked variable or not.

Notice that particular care should be taken when using the `@FilterField` annotation as this interacts with the state-matching mechanism of `jpf-core` and cause `jpf-bdd` to explore otherwise unreachable states. We refer to the details available from the Wiki on `jpf-bdd`'s Web site about when this situation can occur.

4. Examples and experimental results

In this section, we present some examples and experiments that show the performance of `jpf-bdd`. We compare the performance of `jpf-bdd` to the performance of `jpf-core` to address exactly the advantages and disadvantages introduced by `jpf-bdd`. We also discuss the issues raised by the comparison with `jpf-symbc` in Section 4.3. All given statistics were collected on a workstation with the characteristics given in Table 2. The information regarding the performance of the workstation (CPU and reserved RAM) is only relevant for the last example (Section 4.3) because this is the only example where we are comparing execution times.

CPU	Intel Xeon @2.93GHz (4 cores)
OS	Ubuntu 10.10
`jpf-core` version	Mercurial revision 577:7e645ed15e9e
`jpf-bdd` version	Mercurial revision 0:1d365e09f6dc
Java version	OpenJDK version 1.6.0_20
Reserved RAM	5120MB

Table 2. Test system

4.1. A simple example

We introduce a very basic example and compare its verification with `jpf-bdd` to the verification with `jpf-core`. The source code of the example is given in Figure 6. It is a program that only chooses values for a set of Boolean variables. The program has no side effects.

For a meaningful comparison between `jpf-bdd` and `jpf-core`, we need to provide two slightly different programs. In the `jpf-core` variant, the different values of the Boolean variables are generated with calls to method `Verify.getBoolean()`. This method creates a choice generator and forces JPF to explore both possible values for the variable.

The other version is designed for execution with `jpf-bdd`. The Boolean variables are annotated with `@TrackWithBDD` to enable their management with BDDs. When the condition of one of the `if` statements is executed, `jpf-bdd` recognizes that the respective variable is not contained in the BDD. Therefore, it creates a choice generator and explores both possible choices. Up to this point, the behaviour is exactly the same as in `jpf-core`.

The difference becomes clear when it comes to state merging. In `jpf-core`, states can never be merged, because the Boolean variables are always different from one state to another. In `jpf-bdd`, we have, in theory, only one state, because the core parts of the states are always empty and therefore equal. In practice, the verification with `jpf-bdd` generates separate states after each choice generator creation.

```
boolean a1,a2,a3,a4,
  a5,a6,a7,a8;
//gb()==Verify.getBoolean()
void test() {
  if(a1 = gb()) {}
  if(a2 = gb()) {}
  if(a3 = gb()) {}
  if(a4 = gb()) {}
  if(a5 = gb()) {}
  if(a6 = gb()) {}
  if(a7 = gb()) {}
  if(a8 = gb()) {}
}
```

```
@TrackWithBDD
boolean a1,a2,a3,a4,
  a5,a6,a7,a8;
void test() {
  if(a1) {}
  if(a2) {}
  if(a3) {}
  if(a4) {}
  if(a5) {}
  if(a6) {}
  if(a7) {}
  if(a8) {}
}
```

Figure 6. Simple example. Both code snippets are functionally equivalent. We have to add some more code to prevent the compiler from optimizing and deleting the `if` statements but that does not concern the functionality. Method `gb` returns an arbitrary Boolean value.

Table 3 shows some statistics comparing the verification using `jpf-bdd` and `jpf-core`. It shows clearly that `jpf-bdd` generates a smaller state-space and executes less byte-code instructions. Specifically, for the given example, the verification with

	jpf-core	jpf-bdd
# States generated	511	17
# Backtracked	511	17
# Choice generators	512	18
# Instructions	9356	3076

Table 3. Statistics of the simple example

jpf-core generates 511 program states. The corresponding reachability tree is a perfect binary tree with a height of 8 (because there are 8 calls to `Verify.getBoolean()`). The size of this tree is $2^8 - 1 = 511$. Therefore the size of the reachability tree in a corresponding example with x variables is $2^x - 1$.

With jpf-bdd, we have only 17 program states. The generated reachability tree looks very similar to the DAG given for jpf-bdd in Figure 5. It is a chain of nodes with two same-directed edges between each node. The first node is the starting node. After every creation of a choice generator by jpf-bdd, two additional nodes are created (one for each choice). Because the chosen variable value does not change the core state, both of the new states representing these choices are merged. This example contains 8 variables. Therefore, the state space contains $1 + (2 * 8) = 17$ states. The formula for a corresponding example with x variables is $1 + (2 * x)$.

Obviously, the size of the reachability tree generated by jpf-core is exponential in the number of variables, while the one generated by jpf-bdd is linear. The number of executed byte-code instructions reflects this small state space because less edges have to be explored. Notice that this is a very artificial example and therefore not generalizable. Furthermore, the creation and usage of BDDs introduces some overhead, resulting in jpf-bdd being slower than jpf-core for small state spaces (the timing statistics for small problem sizes like $x = 8$ range in milliseconds and would therefore not be relevant). Even though this example is artificial, we want to give an idea of the performance with bigger problem sizes. With $x = 26$, jpf-bdd still completes the task in less than one second while jpf-core needs nearly two hours (with DFS) or runs out of memory (with BFS).

4.2. A more complex example

This section reuses the example of the previous section to construct a more complex example (code is shown in Figure 7). We introduce an integer variable `i` and increment this variable in each body of the `if` statements. This creates different integer values and, more importantly, it creates many states with different core parts (because `i` is maintained by jpf-core). This example also provides many interleavings between different parts of the reachability tree. For example, there are 8 different paths resulting in the integer value 1.

Table 4 contains statistics on the evaluation of this example. The statistics of jpf-core remain unchanged except of the number of executed instructions. This is due to the added integer increments. The generated reachability tree remains exactly the same because we had already generated one path per combination of the Boolean variables in the first example. This example simply adds an integer value to each of these paths.

The statistics of the execution with jpf-bdd show more changes. First of all, the number of states has changed. The reason for this is that we cannot simply merge every two states because sometimes they have different core parts. The generated reachability tree is a DAG that grows in breadth when more integer values become possible. The DAG has 8 levels (corresponding to the number of variables). Each `if` statement creates a new level. Each level has as many nodes as integer values of `i` are currently possible. The DAG starts with one node (the starting node) and one possible value for `i` (0). The next level has two nodes, one with $\texttt{i} = 0$ and one with $\texttt{i} = 1$. In the beginning the third level has the four nodes $([\texttt{i} = 0] \wedge \neg\texttt{a1} \wedge \texttt{a2})$, $([\texttt{i} = 1] \wedge \texttt{a1} \wedge \neg\texttt{a2})$, $([\texttt{i} = 1] \wedge \neg\texttt{a1} \wedge \texttt{a2})$ and $([\texttt{i} = 2] \wedge \texttt{a1} \wedge \texttt{a2})$. Because of the separation of BDD part and core part we can merge the second and the third state into $[\texttt{i} = 1] \wedge ((\texttt{a1} \wedge \neg\texttt{a2}) \vee (\neg\texttt{a1} \wedge \texttt{a2}))$. Therefore, the third level has only 3 states. This scheme continues for the next levels. After each Boolean choice point JPF generates two new states. In our case, all states of one level that have the same integer value are merged before the next level is explored. The merged states do not appear separately in the DAG, but they are counted as separate state creations in the statistics. In the end, we have a state space of size $(2 * (\sum_{n=1}^{8} n)) + 1 = 73$ because the last level is collapsed to one node by JPF. The formula for a corresponding program with x variables is $(2 * (\sum_{n=1}^{x} n)) + 1$. This is still much better than the exponential complexity of using jpf-core.

4.3. The dining philosophers

This section presents the verification of an implementation of the *dining-philosophers problem*. The problem is situated in a dinner of a number of philosophers. They sit at a round dinner table and have

```
int i = 0;
boolean a1,a2,a3,a4,
  a5,a6,a7,a8;
//gb()==Verify.getBoolean()
void test() {
  if(a1 = gb()) {i++;}
  if(a2 = gb()) {i++;}
  if(a3 = gb()) {i++;}
  if(a4 = gb()) {i++;}
  if(a5 = gb()) {i++;}
  if(a6 = gb()) {i++;}
  if(a7 = gb()) {i++;}
  if(a8 = gb()) {i++;}
}
```

```
int i = 0;
@TrackWithBDD
boolean a1,a2,a3,a4,
  a5,a6,a7,a8;
void test() {
  if(a1) {i++;}
  if(a2) {i++;}
  if(a3) {i++;}
  if(a4) {i++;}
  if(a5) {i++;}
  if(a6) {i++;}
  if(a7) {i++;}
  if(a8) {i++;}
}
```

Figure 7. More complex example. Both code snippets are functionally equivalent.

	jpf-core	jpf-bdd
# States generated	511	73
# Backtracked	511	73
# Choice generators	512	74
# Instructions	8338	3420

Table 4. Statistics of the more complex example

one fork between each of them. Each philosopher needs two forks to eat. The problem is that, if each philosopher picks the fork to his right (or left), then no philosopher can get the second fork and they cannot start eating. The dining-philosophers problem is a simple demonstration of a deadlock between asynchronous threads in a program (in this case, each philosopher represents a thread). At a larger scale, this problem appears, for example, in operating systems, so it has real-world relevance.

This section compares the performance of `jpf-core` and `jpf-bdd` on the verification of our Java implementation of the problem, in which forks are represented by Boolean variables (the full code is available from the source tree of `jpf-bdd`). We concentrate on the time needed for verification. We are also looking at structural statistics such as the size of the state space to explain the verification times.

Both verification approaches find a deadlock after a few milliseconds, which is obviously too fast for a meaningful comparison. Therefore, we configure the verification to search for all deadlocks in the program. This means that the whole reachability tree has to be explored, which takes significantly more time. Our implementation of the philosophers problem is scalable in the number of philosophers that take part in the dinner. To optimize the performance of `jpf-core` on the example we have configured it to use the BFS search strategy. BFS causes `jpf-core` to use more memory but it also makes the verification much faster.

Table 5 shows the results for the verification using `jpf-bdd` and `jpf-core`. The most important figures are the times needed to complete the verification (i.e., to explore the whole reachability tree). We found that the numbers are quite stable during multiple runs of the verifications, so we can assume that there is a relatively small variance. In the more complex examples, `jpf-bdd` is faster and consumes less memory than `jpf-core`. The speedup of `jpf-bdd` can be explained by the generally lower number of executed instructions. This is due to the smaller reachability tree (as suggested by the number of generated states). However, this example and the verification process is too complex to provide an analytical expression for the total number of states. The generally very high memory consumption is caused by the usage of the BFS-search algorithm. Our experiments showed that the memory consumption is typically lower when using DFS search but, in this example, DFS needs much more time to complete the verification (not shown in the table).

In the experiment with problem size 3, `jpf-bdd` consumed slightly more memory than `jpf-core`. The reason for this is probably the additional memory needed for the loading and usage of the BDD library. In the more complex experiments, the maximum memory footprint of `jpf-bdd` is lower than that of `jpf-core`. The reason for this difference is twofold: First, `jpf-bdd` generates fewer states and therefore fewer states have to be stored. Second, each state requires less memory because part of it is stored with BDDs, which are usually an efficient memory structure (Section 2.1 gives a intuitive example for the memory-efficiency of BDDs for Boolean variables.)

Both verification approaches fail to verify the experiment with the problem size 6 because they run out of memory. This leads to the question of what can be optimized to further reduce the memory consumption of `jpf-bdd`. So far we have not touched the variable ordering in the BDDs. The ordering is determined by the variables appearance in the program control flow. Dynamic variable reordering might improve the memory consumption and the time needed for every single BDD operation.

We also verified this example with `jpf-symbc`, but we could only verify the examples with 2 and 3 philosophers, running out of memory in the case of 4 philosophers. `jpf-symbc` is a JPF extension that uses symbolic execution of the Java program. In contrast to our extension, which only abstracts Boolean values, `jpf-symbc` abstracts more data types, including integers and strings. However, on this exam-

Problem size		3 phil.	4 phil.	5 phil.	6 phil.
Time (h:mm:ss)	jpf-bdd	0:00:01	0:00:12	0:01:56	-OOM-
	jpf-core	0:00:03	0:00:24	0:04:26	-OOM-
Memory consumed	jpf-bdd	150MB	617MB	2,021MB	-OOM-
	jpf-core	118MB	874MB	2,828MB	-OOM-
States generated	jpf-bdd	5,021	67,015	682,195	-OOM-
	jpf-core	5,505	74,013	762,291	-OOM-
Instructions executed	jpf-bdd	137,855	1,992,917	21,525,880	-OOM-
	jpf-core	404,748	6,355,377	76,061,181	-OOM-

Table 5. Verification statistics for the philosophers example. The table states some statistical figures for the verification with jpf-bdd and jpf-core. The problem is scaled in the number of participating philosopher threads. With a problem size of 6, both approaches were not able to complete the task because they consumed too much memory.

ple, jpf-symbc needs substantially more time than jpf-bdd to complete the verification. We have identified three reasons for the behaviour of jpf-symbc on this example. First, jpf-symbc cannot benefit from the symbolic abstraction as Boolean variables only have 2 values and cannot be further abstracted. Second, the jpf-symbc listener has a method that is called once per executed instruction. We discussed in Section 3.2 why we avoided implementing this strategy. During verification, JPF executes 100,123 instructions in the example with 3 philosophers before finding the first deadlock. In comparison, jpf-bdd executes less than 1,000 instructions before finding the first deadlock in this example. Third, jpf-symbc does not allow to use BFS for state-space exploration. This strategy has been shown to be more efficient on the example, and jpf-bdd and jpf-core use the BFS strategy.

5. Related Work

There is a number of related approaches in the field of (symbolic) verification of software systems. We focus on approaches that use BDDs for verification. Various researchers have used BDDs successfully to verify properties of hardware and software systems. The main focus is on verification of hardware or models of hardware. To the best of our knowledge, verification of programs written in high-level, general-purpose languages such as Java has not been done with BDDs so far. Another novelty of our approach is that we utilize the ability of state merging to gain efficiency.

There are several very mature approaches in the field of symbolic verification. Examples include the model checking tools SLAM [6], BLAST [7] and CPAchecker [8]. Furthermore, jpf-symbc allows the user to start the symbolic execution of the program at any point of time during verification. We have not implemented this functionality in jpf-bdd but it is technically possible. We refer to Section 4.3 for some observations about the performance of jpf-symbc on the dining philosophers example. We note here that dynamic enabling and disabling of BDD management of variables might be a future research avenue for jpf-bdd.

Contemporary symbolic model-checking approaches have in common that they focus on handling the whole state-space with symbolic state representation. In contrast, we are partitioning each state into one part tracked with symbolic state representation and one part tracked with explicit representation. This enables us to do state merges in a simple and flexible way.

There are various approaches that apply BDD-supported verification to low-level systems or models of such systems. An example is the tool RABBIT of Beyer et al. for the verification of real-time systems based timed automata [9]. During verification, the variable orderings are optimized to improve BDD performance. The work of Dill and Hu on BDD-based verification supports BDD-based verification for higher language constructs such as arithmetics, sequential control flow, and complex data structures [10]. They also worked on formal hardware verification with BDDs [11]. In contrast to these approaches, we are verifying a complete multiple-purpose programming language (Java byte-code) including constructs such as classes, exceptions, and threads.

6. Conclusion

We presented jpf-bdd, a model-checking tool for Java that uses Binary Decision Diagrams (BDDs) to represent parts of the program states symbolically. jpf-bdd is implemented as an extension to the Java

Pathfinder (JPF) model-checking framework.

The main aim of software model checking is the detection of defects in software systems. Unfortunately, this task is affected by the state explosion problem, which occurs when all the possible program executions need to be explored, resulting in long verification times or high memory costs. Our `jpf-bdd` tool reduces the impact of the state explosion problem for a specific class of Java programs in which Boolean variables play a role in the control of the program flow.

In this paper, we have made two contributions. First, we have developed a working model checker for Java programs that uses BDDs to represent a part of the program states. This model checker is faster than normal JPF and symbolic JPF on a set of example programs. Additionally, to the best of our knowledge, it is the first BDD-based model checker for Java applications.

Second, we made a theoretical contribution by presenting a methodology to partition the state space into one part that is managed by explicit verification and another part that is manged with BDDs. This partitioning of the states allowed us to efficiently merge states and therefore reduce the state explosion problem; it is applicable to other tools beside JPF.

In addition to the examples presented here, possible other application scenarios are Java programs that have a known set of Boolean variables that have impact on the control flow (e.g., they appear in many if statements) or are changed very often during the program execution. These variables can be annotated with a Java annotation provided by `jpf-bdd`. The annotated variables make up the BDD-managed part of the state space.

Concretely, real-world applications where `jpf-bdd` can play a key role include applications where the changing part of the program state mainly consists of Boolean variables (such as in the philosophers example described in Section 4.3), or software product lines where Boolean variables have great impact on which program statements are executed, while the rest of the program states remain often equal [2], [12].

We are actively working on improving the performance of `jpf-bdd` and evaluating its performance on a bigger set of more realistic systems. The optimization of the variable ordering in BDDs has been identified as a possible point for improvement.

Acknowledgments

We thank the reviewers for their constructive and helpful suggestions. This work was partially supported by Google Summer of Code 2011 and by the German DFG grants AP 206/2, AP 206/4, and LE 912/13.

References

[1] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2008, pp. 15–26.

[2] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 321–330.

[3] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, pp. 293–318, 1992.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[5] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 1999, ch. 3.11.5.

[6] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam," *Communications of the ACM*, vol. 54, pp. 68–76, July 2011.

[7] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 5, pp. 505–525, 2007.

[8] D. Beyer, M. E. Keremoglu, and P. Wendler, "Predicate abstraction with adjustable-block encoding," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. FMCAD, 2010, pp. 189–197.

[9] D. Beyer, C. Lewerentz, and A. Noack, "Rabbit: A tool for BDD-based verification of real-time systems," in *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, ser. LNCS 2725. Springer, 2003, pp. 122–125.

[10] A. J. Hu, D. L. Dill, A. J. Drexler, and C. Yang, "Higher-level specification and verification with BDDs," in *Proceedings of the Fourth International Workshop on Computer Aided Verification (CAV)*. Springer, 1993, pp. 82–95.

[11] A. J. Hu, "Formal hardware verification with BDDs: an introduction," in *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*. IEEE, 1997, pp. 677–682.

[12] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2011.