Trail-directed Java Program Verification

Tilman Mehler and Stefan Edelkamp

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51
D-79110 Freiburg
mehler@informatik.uni-freiburg.de
edelkamp@informatik.uni-freiburg.de

Abstract This paper introduces the application of trail-based heuristics for software verification. Trail-directed program verification serves the purpose of shortening an error trail, so that it will be more comprehensible for the user. We explain the concept of two heuristics that are based on the Hamming- and FSM-distance between states of a Java program. We then utilize the Java software verifier JPF to implement and test the new heuristics and to compare them to the heuristics that are already provided by this tool.

1 Introduction

Safety aspects of software are becoming increasingly important. Particularly in the fields of space travel and aviation, a fault-free behavior of the underlying software is indispensable, since failures in practice may cause heavy financial damage or even loss of human lives. However, purely manual inspection of source code quickly proves to be insufficient for these purposes, since less obvious errors usually remain undiscovered. These problems call for the development of formal methods that enable the user to automatically prove, that his/her software satisfies certain properties.

Some first approaches forwards this field were made with the tools Bandera[1] [12] and Java PathFinder (JPF)[2] [18]. While Bandera transforms Java source code into the appropriate representation of existing model checkers - like e.g. Spin [14] - JPF uses a different approach with its own Java Virtual Machine (JVM) that can directly operate on Java byte code.

The name of JPF originates from the rover operating on Mars in 1997 which, due to a software failure, caused some trouble [13]. There are actually two versions of JPF. JPF1, the first version, is a Java-to-Promela[3] converter

1 Kansas State University
2 NASA Ames Research Center
3 Promela: Description language of the Spin model checker.

which, for several reasons, is no longer under development. The second version - JPF2 - uses a different approach. Instead of extracting a model from the Java program, which in turn can be verified by traditional model checking methods, JPF2 implements its own JVM that makes it possible to directly search in the byte code. In the following, JPF2 will simply be referred to as JPF. JPF allows to define assertions within the source code of a program, thereby enabling the user to check safety properties. Future versions of JPF [4] are supposed to have additional support for the verification of LTL-properties [7,1].

As we will see, exploring Java programs can result in huge state spaces where the application of pure undirected search methods no longer make sense. Either such methods fail to find an error due to time and memory restrictions, or the the resulting error trail is too long to be understood by the user. The use of heuristics in program verification is one possible approach to handle these problems. JPF already has some built-in heuristics for directed verification and also provides the user a comfortable way of implementing his/her own heuristics. The internal heuristics of JPF exploit knowledge about the structure of Java byte code [11,10] rather than knowledge about an actual goal state. In contrast, trail-based heuristics exploit the information about a given error state to find a short and easy to understand error-trail. We believe, that program verification can greatly benefit from trail-based heuristics, since finding a long or sub-optimal error-trail is often not very difficult[5]. In this work we study, implement and test two such heuristics with JPF.

This paper is structured as follows. Sections 2 and 3 provide an introduction to the interpretation of Java programs as search spaces. This will, on the one hand, give insights to the complexity of program verification, on the other hand, it will show, how traditional model checking techniques can be applied in this new field. We will give a small example for a Java program and its byte code as produced by the compiler. Section 3 also explains the principles underlying the built-in heuristic estimates of JPF. Section 4 introduces the concept of our heuristics, namely the FSM-distance and Hamming-distance, as well as their realization in JPF. Section 5 discusses the results that were obtained by experimentally comparing the new heuristics to the ones that came with JPF. In Section 6 we discuss some related work that deals with directed model checking or program verification. Section 7 draws some concluding remarks and gives motivations for further efforts in this field.

[4] The version of JPF used in this work is JPF/JVM v0.1.

[5] A good example for this is provided by the experimental results for the leader election protocol in Section 5.

2 State Space of a Java Program

The *Java byte code* (JBC) is the key to the platform independence of Java. Essentially it is a high level machine language, which can be interpreted by a *Java Virtual Machine* (JVM) or translated to the platform specific code of the target machine by so-called *just-in-time* compilers. The *operand stack* of a JVM is used to temporarily store values e.g. the operands for algebraic operations.

Figure 1 shows the source code for a simple program, called *IncDec*, involving two threads *inc* and *dec*. These threads continually increase and decrease the same static variable c in *IncDec*. Figure 2 depicts the byte code of the respective run-method of the two thread classes as produced by the Java compiler *javac*[6]. The only difference between the two is the instruction at Offset 7. The sequence of instructions from Offset 3 to Offset 8 are read as follows:

- getstatic: get a static field and push it on the operand stack.
- iconst_1: push a constant 1 on the operand stack.
- isub (iadd): take the two uppermost elements from the operand stack, calculate the difference (sum) and push the result back onto the operand stack
- putstatic: take the uppermost element form the operand stack and write it to a static field.

Figure 3 depicts the resulting state space of *IncDec* for one iteration of the *while*-loop of both *inc* and *dec*. For the sake of brevity we select a subset of all elements that form the state of a Java program; namely the values of static fields, the program counters and the content of the operand stack. A state is represented by a vector $(c, PC_{inc}, PC_{dec}, OS)$, where c is the value of the static field c in *IncDec*, while PC_{inc} and PC_{dec} are the program counters of inc and dec, respectively, and OS is the content of the operand stack. A step to the right indicates the execution of an instruction in *inc*, a step down means an execution of an instruction in *dec*. This shows how large the state space of a Java program can get even for a very small program. For the $IncDec$ example there is an infinite number of possible configurations if we set aside the case of a variable overflow of c.

3 Heuristic Search in Program Verification

Heuristic search methods use an estimate function h to guide the search. From the set of all possible successor nodes, the one with the best h-value

[6] Note, that the line numbers in Figure 2 denote byte offsets and not positions in the instruction sequence of a method.

```
class Dec extends Thread {
  public void run() {
    while(true) {
      IncDec.c-=1;
    }
  }
}

class Inc extends Thread {
  public void run() {
    while(true) {
      IncDec.c+=1;
    }
  }
}
```

```
class IncDec {
  public static int c=0;
  public static void main(String[] args) {
    Dec dec = new Dec();
    Inc inc = new Inc();
    inc.start();
    dec.start();
  }
}
```

Figure 1. The source code for IncDec.

run-method of the Inc-class	run-method of the Dec-class
==========================	==========================
0 goto 3	0 goto 3
3 getstatic #2 <Field int c>	3 getstatic #2 <Field int c>
6 iconst_1	6 iconst_1
7 iadd	7 isub
8 putstatic #2 <Field int c>	8 putstatic #2 <Field int c>
11 goto 3	11 goto 3

Figure 2. Byte code fragment for the IncDec program

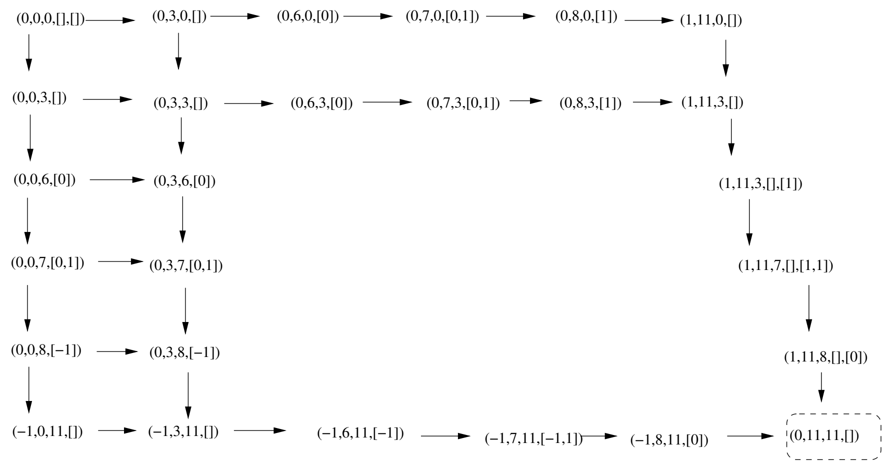

Figure 3. State space of *IncDec*.

is chosen to be expanded next. Two commonly used heuristic search methods are best-first search and A*. Best-first calculates the h-value solely as the estimated distance from a goal node, while A* uses the formula $h(n) = f(n) + g(n)$, where $f(n)$ is the estimated distance of node n to a goal node and $g(n)$ denotes the path costs from the start node to n. If possible, a heuristic should be *consistent* or at least *admissible*. A heuristic h is admissible, if the estimated distance to the goal is a lower bound of the real distance, i.e. for each state u, $h(u) \leq h^*(u)$, were $h^*(u)$ denotes the actual distance of s to a goal state. A heuristic is *consistent*, if for each pair of states u, v, the condition $h(u) \leq h(v) + c(u, v)$ is satisfied. It is evident, that each consistent heuristic is also admissible. Admissibility ensures optimality of A*, i.e. each path from a start state to a goal state found by A*using an admissible heuristic is guaranteed to have minimal length. Consistency of a heuristic prevents reopening of a node when using A*.

Heuristic search was already applied in the fields of model checking and program verification (see [9,10]). Here, it serves at least three purposes. First, it is expected to find errors faster. Second, heuristic search methods are expected to require less memory and hence to be able to explore larger search spaces. Third, one expects short error-trails, that can easily be traced by the user.

Our experimental data compares some the new heuristics to those that are already provided by JPF, namely *interleaving, most-blocked* and *branch-global*, briefly summarized as follows: The interleaving heuristic is based on maximizing thread interleavings [11] using a thread history. The branch-global heuristic maximizes the branch coverage of a partial verification run, i.e. for conditional branching instructions in the byte code, it prefers branches that have not already been taken. The most-blocked heuristic aims specifically to finding deadlocks by maximizing the number of blocked[7] processes (threads in terms of Java).

4 Two Trail-Based Heuristics

Trail-based heuristics exploit the knowledge about a given error-state which might be obtained by a preceding depth-first search (DFS). DFS usually finds an error quickly, but the length of the resulting error-trail is in most cases significantly longer than the optimal one. It is expected, that a directed search procedure that has information about an error-state, will find a shorter trail, while having lower time and memory requirements than an uninformed optimal search method - like e.g. breadth-first search.

4.1 Hamming-Distance

Given two vectors $s = (s_1, ..., s_n)$ and $g = (g_1, ..., g_n)$, the Hamming distance between s and g is defined as $\sum_{i=1}^{n} \delta(s_i, g_i)$, where

$$\delta(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise.} \end{cases}$$

The model checker HSF-Spin[8] makes use of the a Hamming-distance heuristic by comparing system states which are encoded as bit vectors [8]. We exploit the capability of JPF to return the current state as a vector of natural numbers. Two versions of the Hamming distance were implemented. The first version, denoted HM_{eq}, counts the number of different components, while HM_{diff} - the second version - sums the absolute values of the individual differences. I.e. given a current state $s = (s_1, ..., s_m) \in \mathbb{N}^m$ and a goal state $g = (g_1, \ldots, g_n) \in \mathbb{N}^n$, we define: $HM_{eq}(s, g) = \sum_{i=1}^{min\{m,n\}} \delta(s_i, g_i)$ and $HM_{diff}(s, g) = \sum_{i=1}^{min\{m,n\}} |s_i - g_i|$ respectively. Note that m can be different from n, since the size of the state-vector can dynamically grow and shrink during the program execution.

Theorem 1. *HM_{eq} is consistent, if we regard only those components of the state vector, that denote the referenced variables in a system state and if we assume, that a transition can at most change the value of one variable.*

Proof. For goal state g let $h(s) = HM_{eq}(s, g)$ denote the heuristic estimate for system state s. By assumption for each state s and all its immediate successor states $t_1, \ldots, t_m$ we have $h(s) - h(t_i) \in \{-1, 0, 1\}$, $1 \le i \le m$ and therefore (*) $h(s) \le h(t_i) + 1$. Because of the uniform costs for the transitions $c(s, t_i) = 1$ holds. Together with (*) this implies $h(s) \le h(t_i) + c(s, t_i)$ *(qed)*.

[7] A process is blocked, if it waits for access to a locked resource.

[8] http://www.informatik.uni-freiburg.de/~lafuente/hsf-spin/

Theorem 2. *HM_{diff} is not admissible*

Proof. Consider system state $s = (s_1, \ldots, s_n)$ and goal state $g = (g_1, \ldots, g_n)$. Let $h(s) = HM_{diff}(s, g)$ denote the heuristic estimate and s_i, g_i two components from the system and goal state respectively that refer to the same variable x of type *int*. If $s_i = g_i - 2$ and $s_j = g_j$ for all $j \neq i$, we have $h(s) = 2$. If an instruction like *iinc* [15] is executed, which increases s_i by two, the goal state is reached by a single transition and the path costs and hence we have $h(s) = 2 > h^*(s) = 1$.

Despite not being admissible, the HM_{diff} heuristic is still interesting, since in some cases it gives a better estimate for the real distance to a goal state, for example when comparing variables which denote the counter of a loop.

4.2 FSM-Distance

The FSM-distance requires the representation of one or more components of the system as a finite automaton. Given a current state s and goal state g, the FSM-distance is defined as the minimal number of operations required to reach g from s. In the following $n \to n'_1 \to \ldots \to n'_m \to n''$ denotes a path from a node n to a node n'' that leads through nodes $n'_1, \ldots, n'_m$. For a path p, $|p|$ denotes its length.

As opposed to Spin and HSF-Spin, JPF does not explicitly operate on automata models. Therefore, the implementation of the FSM-distance needs to extract an automaton from the byte code before starting the verification process. The construction of the automata is based on the program counters of the involved threads. The actual term will be *transition graph*, since only the successors of a state are of interest and there is no need to explicitly represent the operators[9]. We do not explicitly generate a global transition graph for all running threads. Instead we construct the transition graph for each thread and sum the individual distances. Since the program counter is used as the modeled system component, the execution of an instruction can at most cause a change of state in one of the transition graphs. Therefore our method is equivalent to building the *sysnchronized cartesian product* p [1] of all individual transition graphs and using p to calculate the FSM-distance.

Theorem 3. *The FSM-distance is consistent.*

Proof. Given uniform costs for the execution of an operation, a heuristic h is consistent, if for each state u and each successor state v we have $h(u) \leq h(v) + 1$. We need to distinguish two cases. In the first case v is part of a

[9] One might think of a transition graph as a Kripke-structure [7] without propositions.

shortest path from u to a goal state g with length l. By definition we have $h(u) = l$ and hence $h(u) - 1 = l - 1$(*). Furthermore, the shortest path from v to g must be of length $l-1$. Therefore we have $h(v) = l-1$. Combined with (*) this results in $h(u)-1 = h(v)$ which implies $h(u) \leq h(v)+1$. In the second case v is *not* part of a shortest path from u to g with length l. This means, for each path $p = u \to v \to \ldots \to g$, we have $|p| \geq l+1$ and hence for each path $p' = v \to \ldots \to g$, it holds that $|p'| \geq l$. We can, therefore, conclude that $h(v) \geq l$ and together with $h(u) = l$ we have $h(u) \leq h(v)$ and particularly $h(u) \leq h(v) + 1$ (qed).

Again two versions of the heuristic were implemented. The first version, called *operation-based FSM-distance*, takes each instruction of the byte code as a separate state, while the second version, called *branching-based FSM-distance,* forms states from consecutive byte code instructions separated by branching instructions. The following subsets of the Java byte code instruction set form the basis for our implementation of the FSM-distance:

CB= {ifeq, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq,
if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_acmpe,
if_acmpne, if acmplt, if_acmpgt, if_acmple},

UB= {goto, goto_w},

RT= {return, ireturn, lreturn, freturn, dreturn, areturn}

where CB is the set of conditional branches, UB is the set of unconditional branches and RT is the set of *return*-statements. Together these sets unify to the set $CC = CB \cup UB \cup RT$ of instructions that influence the program counter.

In the following, we regard a *Java class* as a set of *Java methods* and a method as a sequence of *operations*. Formally, an operation is a tuple (m, P), where m is a mnemonic, and P a - possibly empty - sequence of parameters. Given an operation $m = (o, P)$, let $\mu(o) = m$. Given a Java method m with a sequence of operations $O = [o_1, ..., o_n]$, let $ops(m) = O$, $op(m, i)$ the i-th operation in O (if it exists) and $numops(m) = n$. Each Java class will be divided in a set of *method blocks*. Each method block describes a subsequence of the instructions of a method m, denoted as a triple (f, l, m) were $f \geq 1$ is the index[10] of the first instruction and l the index of the last instruction with $1 \leq f \leq l \leq numops(m)$. For a given method block *b*, $method(b)$ returns its method.

A *partitioning function* maps a Java class to a set of method blocks. Given an operation o and a partitioning function β, the target function $target_\beta(o)$ returns the set of method blocks o can branch to. If $\mu(o) \in CB \cup UB$, then

[10] These indexes describe a position within the instruction sequence of a method and should not be confused with byte offsets as in e.g. Figure 2.

$target_\beta(o)$ returns the set of method blocks o can branch to. If $o \in RT$, $target_\beta(o)$ returns all method blocks that contain a branching operation into the method block of o. For a given target function $target_\beta$, the set $succ_\beta$ of *successors* of a method block is defined as

$$succ_\beta(b) = \bigcup_{f \leq i \leq l} target_\beta(op(m, i)).$$

Given a Java class c and a partitioning function β, the *transition graph* of c with respect to β represents a graph (V, E), where $V = \beta(c)$, $E \subseteq V \times V$ and

$$(v_1, v_2) \in E \Leftrightarrow v_2 \in succ_\beta(v_1).$$

Figure 4 illustrates the relationship between the formal constructs which we use to build the transition graph of a Java class. A link between the nodes n and n' labeled with l indicates that an object of type n contains one or more references to an object of type n', which can be obtained by l. Here, l is either a component of n or a function that takes parameters of type n.

The Operation-Based Transition Graph The operation-based transition graph $G = (c, \beta_{op})$ of a Java class c is used to compute the FSM-distance estimate with respect to the number of required operations (i.e. byte code instructions) for reaching a designated target state. In this type of graph a method block contains exactly one operation. The partitioning function β_{op} for the operation-based transition graph is defined as:

$$\beta_{op}(c) = \bigcup_{m \in M} \left(\bigcup_{1 \leq i \leq numops(m)} \{(i, i, m)\} \right).$$

The iterative calculation of the set of method blocks V returned by β_{op} according to a class c is performed by Algorithm 1.

Runtime of Algorithm 1: Let n be the number of operations in all methods of c. Since each operation is regarded only once, the runtime is $O(n)$.

The Branching-Based Transition Graph In this approach the heuristic value is calculated as the estimated number of branches required to reach a target state. The function value of the partitioning function β_{br} for the branching-based transition graph $G(c, \beta_{br})$ is defined as

$$\beta_{br}(c) = \bigcup_{m \in M} \beta'(0, 0, m),$$

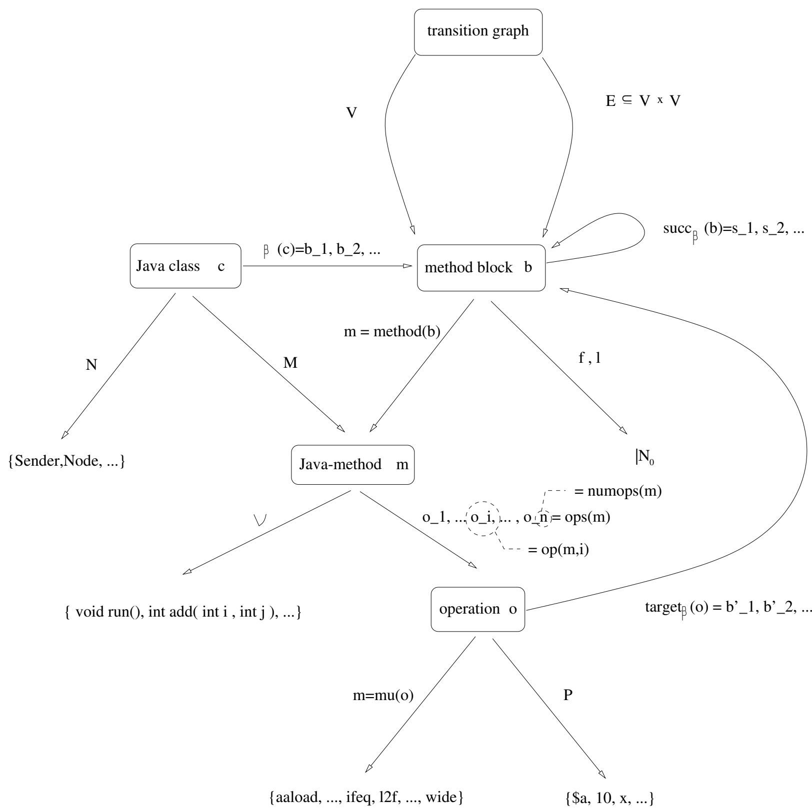

Figure 4. Dependency graph of formal constructs.

Algorithm 1 Calculation of the return value of β_{op}

Input : A Java class c
Output: A set of method blocks V

1. $V \leftarrow \emptyset$
2. for each method m in c do
3. for $i = 1$ to $numops(m)$ do
4. $V \leftarrow V \cup \{(i, i, m)\}$
5. loop
6. loop
7. return V

---

where

$$\beta'(i,j,m) = \begin{cases} \{(i,j,m)\}, & \text{if } j = numops(m) \\ \beta'(i,j+1,m), & \text{if } j \neq numops(m) \text{ and} \\ & \mu(op(m,j)) \notin CC \\ \{(i,j,m)\} \cup \beta'(j+1,j+1,m), & otherwise \end{cases}$$

Algorithm 2 shows the pseudo-code to compute the set of method blocks for a Java-class c as returned by β_{br}. Here the operations are scanned using a pair of counters f (first) and l (last). We increase l by 1 until a branching or return statement is reached and create a new state representing all operations between f and l. Then f is set to l.

Algorithm 2 Calculation of the return value of β_{br}

Input : A Java class c
Output: A set of method blocks V

$V \leftarrow \emptyset$
for each method m in c do
    $f \leftarrow 1$
    $l \leftarrow 1$
    while($l \leq numops(m)$) do
        if($op(m,l) \in CC$)
            $V \leftarrow V \cup \{(f,l,m)\}$
            $l \leftarrow l+1$
            $f \leftarrow l$
        endif
        else $l \leftarrow l+1$
    loop
loop
return V

Runtime of Algorithm 2: Here, the same holds as for β_{op}. Since each operation is regarded only once, the runtime is $O(n)$, where n denotes the number of operations in the class.

Based on the definitions above, we can calculate the transition graph of a Java class using Algorithm 3.

For the calculation of $target_\beta$, we use a dynamic approach which maintains a set of lists while the method blocks of a class are determined. We will not

Algorithm 3 Extraction of a transition graph.

Input: A Java class c and a partitioning function β
Output: The transition graph $G(c, \beta) = (V, E)$ of c

1. $V \leftarrow \beta(c)$
2. $E \leftarrow \emptyset$
3. for each $n \in V$ do
4.   for each $n' \in target_\beta(n)$ do
5. $E \leftarrow E \cup \{(n, n')\}$
6.   loop
7. loop
8. return (V, E)

discuss this approach in detail to avoid getting too technical. However, it can be shown, that it does not alter the asymptotical runtime of the entire algorithm.

Runtime of Algorithm 3: From a graph-theoretical point of view, the runtime has an upper bound that is quadratic with respect to the number of operations in a class. If m denotes the number of operations in c, then $|\beta(c)| \leq m$ and in consequence $|V| \leq m$. Execution of $\beta(c)$ is possible in time $O(m)$, since for the discussed partitioning functions each operation is regarded only once. Since for each $n \in V$ we have $|target_\beta(n)| \leq |V| \leq m$, the inner and outer *for*-loop can be executed at most m times, which accumulates the runtime to $O(m^2)$ for the entire algorithm. However, in the cases discussed above, the number of potential successors of a node is at most 2, because form a conditional branch it is only possible to reach the goal of the branch or the immediate successor of the branching operation. An unconditional branch has exactly one potential successor. Therefore, the inner *for*-loop can be executed at most twice, reducing the total run time to $O(m)$.

Figure 5 depicts the operation-based and branching-based transition graphs for the *run*-method of the *Dec*-class from Section 2.

The Heuristic Estimate From the point of view of the FSM-distance, a system state is a vector $s = (p_1, \ldots, p_m)$ of processes. For each process p let $(V_{p,\beta}, E_{p,\beta})$ be the transition graph of the class of p with respect to a partitioning function β. For the current position $\phi(p)$ of the program counter of p, let $\pi(p) \in V_{p,\beta}$ be the method block containing $\phi(p)$. For $n_1, n_2 \in V_{p,\beta}$, the function $c_\beta(p, n_1, n_2)$ returns the shortest path from n_1 to n_2 with respect to β. For a goal state $g = (p'_1, \ldots, p'_m)$ - where it is assumed, that for $1 \leq i \leq m$ processes p_i and p'_i have the same class - the FSM-distance with respect to

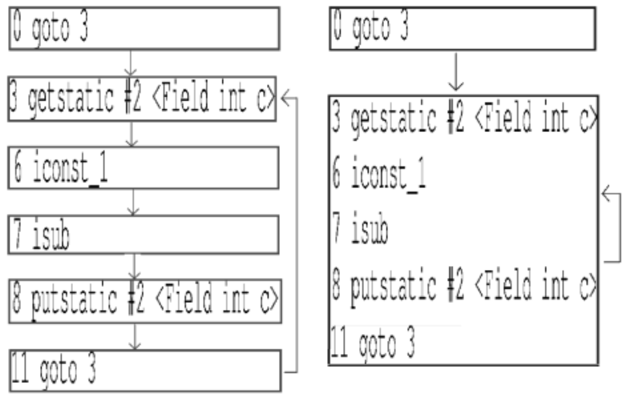

Figure 5. Transition graphs for the Dec class.

the partitioning function β is defined as

$$FSM_{\beta}(s,g) = \sum_{i=1}^{m} c_{\beta}(p_i, \pi(p_i), \pi(p'_i)).$$

For notational convenience, in the following we will write FSM_x for FSM_{β_x}.

5 Experimental Results

The experiments are performed on a Linux-PC with an Athlon-Duron processor at 700MHz and 384MB of RAM. The tested programs are Java implementations[11] of two scalable and one non-scalable protocol, namely the *dining philosophers,* the *alternating bit* protocol and the *leader election protocol*. Where needed, errors are seeded in the source code of the implementation of a protocol, so JPF has something to search for. In the case of the *alternating bit* protocol, we experimented with two seeds: One obvious error that should be easy to find and another more subtle (and thus harder to find) error. From now on, when referring to the *first* alternating bit protocol, we mean the version with the easier to find error. The following search strategies were used:

[11] http://www.informatik.uni-freiburg.de/~mehler/uni-stuff/proj_index.html

DF	=	depth-first search
BF	=	breadth-first search
int	=	interleaving-heuristic (with best-first search)
mb	=	most-blocked-heuristic (with best-first search)
bg	=	branch-global-heuristic (with best-first search)
HM_{eq}	=	equality Hamming-distance as explained in section 4.1 (with best-first search)
HM_{diff}	=	difference Hamming-distance as explained in section 4.1 (with A*)
FSM_{op},FSM_{br}	=	FSM-distance as explained in section 4.2 (with A*)

Table 1 shows some statistics for the calculation of transition graphs (number of nodes, number of edges and the time required to extract a transition graph from a Java class). Here C_{op}/C_{br} indicates, whether an operation- or branching-based graph was extracted from class C.

	$Philosopher_{br}$	$Philosopher_{op}$	$Sender_{br}$	$Sender_{op}$
#nodes	4	28	11	117
#edges	2	26	13	119
time	0.91	0.98	1.56	1.34

	$Receiver_{br}$	$Receiver_{op}$	$Node_{br}$	$Node_{op}$
#nodes	13	90	19	193
#edges	16	93	26	200
time	0.99	1.95	1.15	1.74

Table 1. statistics for the transition graphs

As quality measure we choose the number of instructions executed by the JVM (inst.), the length of the error trail , as well as the time in seconds and the amount of memory (mem) required to find the error. A cross indicates, that no error could be found due to memory restrictions. In the leftmost column of the following tables, *Philo n=x* denotes the results for an instance of the dining philosophers involving x threads, *AB(1)* the results for the fist alternating bit protocol, and *AB(2)* the results for the second one. *LE n=x* denotes the results for an instance of the leader election protocol involving x threads.

Depth-first search

For depth-first search, we expect that it generally finds an error faster than the other strategies, but also returns the longest error trail. The left part of Table 2 depicts the obtained results for the protocols we have selected.

Our expectations proved true particularly in the *leader-election* protocol. For a problem instance of size $n = 4$, depth-first is the only method that can

	inst.	trail	time	mem
Philo n=2	98	18	2.19	0.6
Philo n=3	411	27	2.51	0.8
Philo n=4	2262	36	3.46	0.8
Philo n=5	13291	45	3.62	1.0
Philo n=6	76932	54	6.24	2.5
Philo n=7	434035	63	29.87	9.4
Philo n=8	×	×	×	×
AB (1)	8498	744	8.4	2.9
AB (2)	×	×	×	×
LE n=2	1659	227	6.52	1.9
LE n=3	5237	751	8.80	3.1
LE n=4	6962	979	9.58	3.9
LE n=5	8687	1242	10.20	4.6
LE n=6	10428	1492	11.84	5.3
LE n=32	59528	8676	50.45	55.0
LE n=33	61566	8971	53.43	58.1
LE n=34	×	×	×	×

	inst.	trail	time	mem
Philo n=2	129	18	2.26	0.7
Philo n=3	793	27	2.67	0.6
Philo n=4	5077	36	4.65	1.4
Philo n=5	31483	45	6.13	3.7
Philo n=6	187386	54	17.59	13.5
Philo n=7	×	×	×	×
AB (1)	20755	125	8.9	2.4
AB (2)	238304	122	28.34	21.1
LE n=2	86321	95	14.2	4.7
LE n=3	×	×	×	×

Table 2. Results for depth-first (left) and breadth-first search(right).

find an error. We can even go as high a $n = 33$. However, the resulting error trail already has a length of 979 for a problem size of $n = 4$ and is, therefore, hard to understand for the user. It may seem counterintuitive that, in contrast to breadth-first search, depth-first fails to find an error in the second *alternating-bit* protocol. This phenomenon can be contributed to the fact that, due to the infinite number of possible messages, the state space of the alternating bit protocol is also infinite, which, in case of depth-first search, means the loss of completeness.

Breadth-First Search

From breadth-first search we expect the shortest error trails and in return the longest search times. The results are shows in the right part of Table 2.

Again, the data largely coincide with our expectations. Breadth-first search always returns the shortest error trails, but needs the longest time - or at least the largest number of executed byte code instructions.

Interleaving

For the interleaving heuristic we expect good results for all test runs, since all seeded errors are based on the interaction of communicating processes. The left part of Table 3 shows the results.

	inst.	trail	time	mem
Philo n=2	136	18	2.25	0.7
Philo n=3	709	27	2.5	0.5
Philo n=4	4777	36	3.39	1.0
Philo n=5	31325	45	5.81	2.5
Philo n=6	31325	54	16.87	10.9
Philo n=7	×	×	×	×
AB (1)	19943	125	8.9	3.6
AB (2)	252310	122	29.49	20.6
LE n=2	40939	95	10.32	6.2
LE n=3	×	×	×	×

	inst.	trail	time	mem
Philo n=2	107	18	2.22	0.7
Philo n=3	237	27	2.36	0.7
Philo n=4	343	36	2.61	0.8
Philo n=5	425	45	2.7	0.8
Philo n=6	506	54	2.79	0.9
Philo n=7	588	63	2.85	0.93
Philo n=8	640	72	2.86	0.95
Philo n=9	752	81	2.89	1.1
Philo n=100	8214	900	18.78	28.6
AB (1)	11034	134	7.41	2.0
AB (2)	×	×	×	×
LE n=2	69492	111	7.5	4.7
LE n=3	438612	297	44.2	25.5
LE n=4	×	×	×	×

Table 3. Results for the interleaving (left) and most-blocked (right) heuristic.

Our expectations do not seem to prove right in this case. Especially the bad results for the *dining philosophers* are surprising. This might be due to an implementation detail and is, therefore, a candidate for further inspection.

Most-blocked

Since the most-blocked heuristic is specialized on finding deadlocks, we expect it to perform particularly good for the *dining philosophers* protocol and bad for the remaining protocols. The results are shown in the right part of Table 3.

As expected, the most-blocked heuristic gives very good results for the *dining philosopher*. Surprisingly, it also finds an error for the problem size 3 of the *leader election* protocol, where all other methods except depth-first fail. However this is probably pure accident, since this heuristic is not designed for finding violations of safety properties.

Branch-Global

In principle the branch-coverage heuristic is a promising approach. However, it is hard to make any predictions on how good they will perform on the protocols at hand. The results are shown in Table 4.

For the *dining philosophers* problem the results of branch-global heuristic are almost identical to breadth-first search. The reason becomes clear by the investigation of the byte code: It does not contain any conditional

	inst.	trail	time	mem
Philo n=2	129	18	2.24	0.6
Philo n=3	793	27	2.47	0.7
Philo n=4	5077	36	2.64	1.4
Philo n=5	31483	45	2.72	3.7
Philo n=6	187386	54	2.81	13.0
Philo n=7	×	×	×	×
AB (1)	21896	134	7.41	4.1
AB (2)	105827	142	14.79	11.8
LE n=2	68765	95	11.73	8.5
LE n=3	×	×	×	×

Table 4. Results for the branch-global heuristic.

branches and, therefore, branch-global degenerates to breadth-first search. Branch-global gives moderate results for both versions of the *alternating bit* protocol, but at least finds the error in the second version.

Hamming-Distance

Since the Hamming-distance is a relatively simple heuristic, we expect only a marginal improvement compared to the uninformed search methods.

The results for both versions are shown in Table 5.

	inst.	trail	time	mem
Philo n=2	71	18	4.0	0.7
Philo n=3	129	27	4.1	1.0
Philo n=4	190	36	2.54	0.8
Philo n=5	260	45	5.22	1.0
Philo n=6	339	54	7.64	2.7
Philo n=7	588	63	7.85	0.93
Philo n=8	640	72	7.86	0.95
Philo n=15	752	81	12.89	1.1
Philo n=20	×	×	×	×
AB (1)	19110	276	12.3	3.5
AB (2)	109244	719	22.56	23.1
LE n=2	6032	103	9.79	2.9
LE n=3	×	×	×	×

	inst.	trail	time	mem
Philo n=2	71	18	4.05	0.7
Philo n=3	129	27	4.19	0.7
Philo n=4	2456	36	4.74	0.8
Philo n=5	19870	45	6.73	1.0
Philo n=6	141331	54	16.14	7.5
Philo n=7	914277	63	7.85	71.5
Philo n=8	×	×	×	×
AB (1)	18868	127	11.9	3.6
AB (2)	×	×	×	×
LE n=2	43860	98	13.19	5.2
LE n=3	×	×	×	×

Table 5. Results for the HM_{eq} (left) and HM_{diff} (right) heuristic.

HM_{eq} returns good results for the *dining philosophers* protocol. It is one of the three methods that still find an error for a problem size of 15. In contrast, the HM_{diff} heuristic returns relatively good results for problem sizes up to 6, but fails for $n = 8$. For the rest of the protocols the Hamming-distance performs only moderate, and for the second alternating bit protocol HM_{diff} fails.

The FSM-distance

We expect the FSM-distance to return good results particularly in the *dining philosophers* protocol, since the deadlock that might occur is only determined by the program counters of the threads and not by any variable values. Furthermore we expect, that the operation-based FSM-distance generally performs better, since by the higher granularity of the transition graph there is also a higher degree of correspondence to the actual byte code. The results are shown in Table 6.

	inst.	trail	time	mem
Philo n=2	71	18	6.1	0.8
Philo n=3	127	27	7.24	0.7
Philo n=4	206	36	8.49	1.1
Philo n=5	322	45	10.17	1.4
Philo n=6	505	54	13.47	2.6
Philo n=7	838	63	28.19	9.9
Philo n=15	59074	135	49.85	54.8
Philo n=20	×	×	×	×
AB (1)	11260	130	13.33	4.3
AB (2)	×	×	×	×
LE n=2	32183	97	13.57	4.5
LE n=3	×	×	×	×

	inst.	trail	time	mem
Philo n=2	82	18	6.05	4.8
Philo n=3	230	27	6.84	0.7
Philo n=4	685	36	8.2	1.0
Philo n=5	2084	45	10.1	1.4
Philo n=6	6326	54	13.48	3.4
Philo n=7	19010	63	28.68	12.4
Philo n=15	×	×	×	×
AB (1)	19401	125	13.96	4.08
AB (2)	38464	122	35.54	21.6
LE n=2	36527	95	13.39	3.9
LE n=3	×	×	×	×

Table 6. Results for the FSM_{op} (left) and FSM_{br} (right) heuristic.

FSM_{br} does not perform very well on the *dining philosophers* protocol. As it turns out, it does so for same reason as the branch-global heuristic. Since there are no conditional branches in the byte code, there are very few choices for a branching-based heuristic. In contrast, the operation-based heuristic performs very well, as it was expected. However the branching-based FSM-distance finds the error in the second *alternating bit* protocol, while the operation-based (though better for the first one) fails.

6 Related Work

There are several other projects, that deal with directed model checking and/or program verification. First of all we should mention the heuristic model checker *HSF-Spin* [9] which uses heuristic search methods for the verification of Promela models. The trail-based heuristics discussed in the work relate to the ones in HSF-Spin[8][12] and were implemented for JPF[16]. The Bandera [12] project of the Kansas State University constitutes a multi functional tool for Java program verification. Bandera is capable of extracting models from Java source code and converting them to the input language of several renowned model checkers like Spin or SMV. Also, Bandera serves as a slicer for JPF. However, the tool is still under development making it difficult to test, if this approach is suitable for directed program verification.

Another topic of program verification is type-correctness of programs [5,6]. Besides the classical data flow algorithms that are normally used to check a program for type correctness, some newer approaches make use of formal methods to check the type correctness of Java source code. In [4,2,3] model checking techniques over an automata model of Java byte code are used to check for type correctness. A different approach is used by Nipkow et al. [17], using a theorem prover to check the type correctness of programs from a sub-language of Java, called $\mu - \mathrm{Java}$.

7 Conclusion

This paper contributes to the application of trail-based heuristic in the field of Java program verification. We compared the existing heuristics of JPF to our new heuristics. At this moment it is not possible to nominate a clear-cut winner from the tested heuristics, since all of them have individual advantages and drawbacks. One should point out the good performance of the most-blocked heuristic when detecting deadlocks. This observation was already made in the field of model checking (see e.g. [9]). An important issue is the fact, that all methods perform very bad in the *leader election* protocol[13]. The source code for the protocol merely consists of 150 lines including comments. This implies that even for small programs, software verification is a complex problem. Another important conclusion is, that methods from the classic field of model checking are also applicable in software verification - with considerable modifications where necessary. In the future some effort should be invested to further improve the heuristics. The current implementations do not fulfill the formal properties proven in section 4. We are currently working on more sophisticated versions of the heuristics, which are

[12] http://www.informatik.uni-freiburg.de/~lafuente/hsf-spin/.index.html

[13] Though the depth-first search finds solutions for problem sizes up to 33, the length of the error trails are unacceptable.

supposed to realize the admissibility and consistency properties as well as to give overall better results. The experimental results in section 5 suggest, that the implementation of the *leader election* protocol would be a good platform for testing new methods and heuristics. For example one could claim that a heuristic is good, if it solves an instance of $n = 4$ with a significantly shorter error trail than depth-first (979 namely).

References

1. B. Bérard, M. Bidoit, A. Finkel, F. Laroussine, A. Petit, L. Petrouccim, and Ph. Schnoebelen with P. McKenzie. *Systems and Software Verification*. Springer, 2001.
2. D. Basin, S. Friedrich, and M. Gawkowski. Verified Bytecode Model Checkers. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 47–66. Springer-Verlag, 2002.
3. D. Basin, S. Friedrich, M. Gawkowski, and J. Posegga. Bytecode Model Checking: An Experimental Analysis. In *Model Checking Software*, pages 42–59. Springer-Verlag, 2002.
4. D. Basin, S. Friedrich, J. Posegga, and H. Vogt. Java byte code verification by model checking. In *11th International Conference on Computer-Aided Verification (CAV)*, pages 491–494. Springer-Verlag, 1999.
5. L. Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.
6. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
8. S. Edelkamp and A. Lluch-Lafuente. *HSF-Spin User Manual*, 2001.
9. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Model Checking Software*, pages 57–79. Springer, 2001.
10. A. Groce and W. Visser. Heuristic Model Checking for Java Programs. *SPIN Workshop on Model Checking of Software*, pages 242–245, 2002.
11. A. Groce and W. Visser. Model checking java programs using structural heuristics. In *International Symposium on Software Testing and Analysis*, pages 12–21. ACM Press, 2002.
12. J. Hatcliff and O. Tkachuck. The Bandera Tools for Model-checking Java Source Code: A User's Manual. Technical report, Kansas State University, 1999.
13. K. Havelund. *Java PathFinder User Guide*. NASA Ames Research Center, Moffet Field, CA, USA, 1999.
14. Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
15. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. The MIT Press, 1999.
16. T. Mehler. Gerichtete Java Programmverifikation. Master's thesis, University of Freiburg, 2002.
17. T. Nipkow, D. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In *Foundations of Secure Computation. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.

18. W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. Post-CAV Workshop on Advances in Verification, Chicago, 2000.