Method summaries for JPF

Lasse Berglund
Pivotal Labs
London, UK
lberglund@pivotal.io

Cyrille Artho
KTH Royal Institute of Technology
Stockholm, Sweden
artho@kth.se

ABSTRACT

Java Pathfinder (JPF) is a virtual machine executing Java bytecode that is able to perform model checking using backtracking execution. Due to backtracking, parts of a program may be executed multiple times during model checking. Hence, we explore whether method summaries can be used to make JPF's model checking more efficient. We present the design and implementation of dynamically generated summaries as an extension of JPF.

While our summaries incur an overhead that outweighs the benefits in most cases, the approach shows promise in certain cases, in particular when stateless model checking is used. We also provide some results related to cases when our summaries are applicable that could provide guidance for future work within this field.

1. INTRODUCTION

Java Pathfinder (JPF) is a framework for Java bytecode analysis [19, 2]. An explicit-state model checker [11] is the core of JPF. JPF executes Java bytecode in its own Virtual Machine (VM), which in addition to standard capabilities of a VM supports non-deterministic *choices*. Non-determinism can arise due to choices over parameters or due to thread-level concurrency, as the thread schedule in Java depends on the environment [19, 2].

A choice causes JPF to explore all possible outcomes. Concurrency is analysed by exploring all serialised schedules one by one, in a sequential way. In order to avoid repeating program execution from the initial state, JPF supports *backtracking* to reset the VM to a previously visited state. As a result of this approach, JPF executes parts of a program under a particular schedule in a linear fashion, one thread at a time, until it hits a choice. Each outcome of the choice is again explored sequentially. It can be observed that some operations are explored repeatedly under different choice outcomes, even if the result of a particular method call does not differ between them.

Due to the size of the state space, and the redundancy of some of the choice outcomes, the *state space explosion problem* continues to be a limiting factor for software model checking. We propose *method summaries* to capture the preconditions, side effects, and postconditions of a method succinctly, in order to apply them when a method is executed again in the same state.

In this paper we describe our approach for improving the performance of Java Pathfinder's (JPF) model checking procedure through the use of method summaries. We present our approach, the results of our experiments, and guidance for the future development of the technique in the context of JPF.

This work is based on a Master's thesis [3]. We summarise the findings and show how often summaries are applicable, and useful, in a variety of benchmarks. Our implementation of method summaries is available at `https://github.com/lassebe/jpf-summary`.

2. BACKGROUND

The architecture of JPF lends itself to the implementation of method summaries, which in turn can be used to augment stateful or stateless model checking.

2.1 Java Pathfinder

The main architectural component of JPF is a Java virtual machine, implemented in Java. This component supports functionality for executing bytecode as well as backtracking over already executed code [2]. When analyzing a concurrent program with multiple threads, the number of thread interleavings is in principle exponential in the number of threads and statements [4]. However, most bytecode operations are thread-local, and do not affect the global state of the program. Interleavings between such operations are therefore not relevant and can be ignored as a partial-order reduction [4]. Therefore, JPF groups instructions in an atomic *transition* as long as they cannot have any visible effect on other threads [2]. If an instruction has a globally visible effect, JPF uses a *transition break* in order to explore interleavings between executions where the outcome may depend on the schedule. Such instructions may be instructions that access shared memory, or a call to a native method, which often has a globally visible side effect, such as printing to the console.

2.2 Method Summaries

A procedure summary is defined as concise representation of a part of a program; usually a function or a method [6]. They are heavily used in program analysis, as they allow for compositional verification, where previous results (the summaries) can be reused throughout the verification process [16, 13, 5].

The method summaries we describe in this work are summaries over methods in Java bytecode. We consider the dependencies of a summary to consist of the arguments of the method, and all the fields which the method reads from. Note that this does not include local variables, which exist only within the scope of the method. Similarly, we consider the effects of a method to be its return value and the fields which the method writes to.

2.3 Stateless Model Checking

JPF uses *stateful* model checking, in that it matches each state with previously visited state. At the expense of extra memory, this allows JPF to avoid revisiting redundant states.

In contrast to this, *stateless* model checking is an approach where the model checking procedure does not store the states as they are explored and checked [9]. Godefroid argued that in order to apply

model checking to actual programs, one could not maintain the assumption that each state could be assigned a unique id, which is required if one performs stateful model checking.

The key to making stateless search work is to reduce the amount of non-determinism to a fixed set of schedules [12]. The model checker then explores each schedule in turn, until a property violation occurs, or they have all been explored. Stateless model checking remains an active field [1, 14].

JPF is able to perform stateless model checking, but does not do so by default, as it reduces performance significantly.

3. IMPLEMENTATION

This section defines method summaries and their implementation.

3.1 Design

We define a summary of a Java method as $S = (P, I, O, R)$. We refer to P and I as the *context* of a method. P is a an ordered list representing the arguments of the method by the values of the parameters. For non-static methods, P includes a reference to the callee object (*this*). I is a set of tuples $(id, value)$ and contains all fields that are read by the method, during a particular execution path, and the values that were read from those fields. We only store the first value read from a particular field, as any subsequent reads will not affect the execution path of the method. If a method reads from a field inside a non-primitive field, we store a reference to the outer field, as well as the inner value. O is a set of tuples $(id, value)$ containing all fields that are written by the method, capturing the effects of the method. As we are interested only in the state after the method completes, we overwrite the stored value if a field is written to multiple times. So the contents of O will be the last values written to fields. R represents the return value of the method, either a primitive value or the reference of an object, or for methods with return type **void** it is simply $\emptyset$. As a method might be called in multiple different contexts, we create multiple summaries for each method, up to a user-defined limit.

As an example, consider the method in Listing 1: if we call `method(37)` we will create the following summary:
$S = ([r, 37], \{this.value, 5\}, \{(this.flag, true)\}, 37)$. P contains the reference to the callee object, r, and the argument x with the value 37. The method reads the value of the field **value** at the condition in the loop-statement, so it is added to I. O contains the new value written the field **flag**. The return value is simply stored as 37; if a method returns a non-primitive type, we store a reference to the returned object.

Listing 1: A small example class.

```
public class Example {
  private boolean flag;
  private int value = 5;

  public int method(int x) {
    for(int i=0; i<value; i++) {
      x++;
    }
    flag = (x == 42);
    return x-5;
  }
}
```

We create summaries dynamically through a process we call *recording* (see Fig. 1). We say a summary is *recorded* for a given context if we have a summary with that context. *Recording* starts at method invocation, and finishes successfully when the method returns. When we perform the modifications contained in the summary, we say that the summary is *applied*. A summary can only be safely applied if the context in which the method is called matches that of the context stored in the summary.

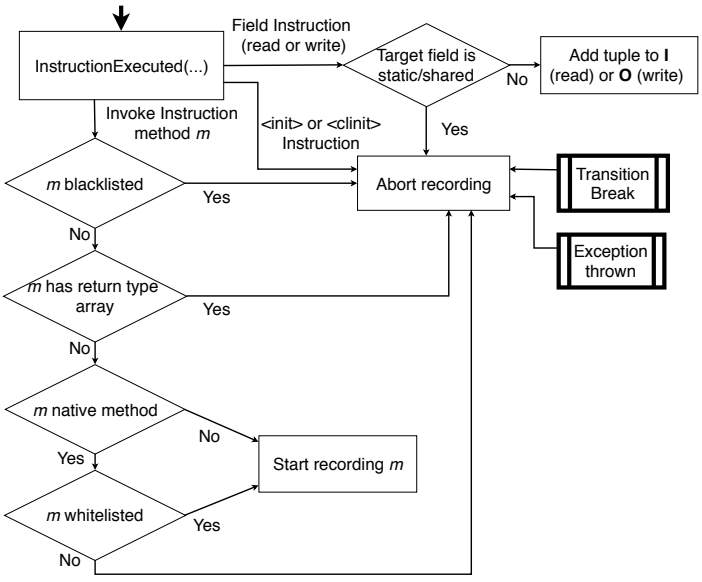

Figure 1: Diagram showing the process of recording, and possible interruptions.

3.2 Construction in JPF

We implement our summary creation using a listener. The listener primarily listens to notifications related to instruction execution. When JPF executes a method invocation, the listener checks if the method has been recorded; if so, the listener compares the context of existing recordings to the current state of the system. This comparison may contain non-primitive types, such as the callee object **this** for non-static methods. In this case, we compare the object reference and the fields that the referred object contains.

If there is a recorded summary with a matching context, the invocation is skipped, and the summary is applied instead. If the method has not been recorded in that context, the listener will start recording and continue until the corresponding return instruction is executed, at which point the summary is stored.

As described above, I contains all fields and their values that are read by the method (inputs). We only consider the *first* read of each field, as updates during the execution of the method will be reflected in the output set O, which contains the fields that are written by the method, with their values. Intermittent updates are not important for the summary, so O contains only the values of the *last* write of each field used by the method.

In order to ensure that we do not impact the soundness of JPF's model checking, we ensure that summaries take into account whether or not other threads are live during recording. A method might be safe to summarise in a single-threaded context, but could be interrupted in a multi-threaded context.

There are a few reasons why we might stop recording prematurely: the method might be interrupted by a transition break, or the method might call a native peer that we are unable to summarise, or the method calls a method that has been previously blacklisted because it was interrupted during recording.

In order to increase the number of methods we can successfully summarise, we manually configure a so-called white-list of native methods. This list contains the names of native methods that are known to have no side-effects that we cannot summarise. Without this list, we would have to abort recording unconditionally when a method calls a native method, even if the method in question only has side-effects that are irrelevant to the verification. Examples of methods that we add to the white-list are `print` and `println`, as their side-effects are generally not of concern to the verification process, and `desiredAssertionStatus`, which is the native method called when evaluating an `assert`-statement in Java. The latter can be summarized safely because repeated method invocations do not change the outcome of the assertion.

The application of a summary involves a few more steps than described above. To create summaries of nested method calls, the listener has to propagate context and modifications to the summaries of methods that are being recorded, as the read and write instructions of the summarised method would not be executed and captured in those summaries otherwise. After that, the listener has to get the instruction that follows directly after the method invocation, remove any arguments from the current stack frame, get the return value from the summary's *modifications* and place it on the stack, and finally set the program counter of JPF to the instruction directly following the method invocation.

As the listener is executing on the Host VM, rather than inside the JPF VM, we should see a similar performance benefit to the one described by d'Amorin et al. [8].

3.3 Interruptions

Our summaries are able to capture the effects of methods that read and write to fields of complex objects. However, there are a number of scenarios that our summaries are not able to capture, in which case we abort recording. Most importantly, we cannot create summaries over transition boundaries, because a summary compresses a method's execution to an "atomic" event. A transition boundary occurs when accessing shared or static data or invoking an operation that may affect other threads.

We currently cannot capture the effects of a method that adds new objects to the heap, as this would require tracking references that escape the context of the summary. Therefore, we interrupt recording if a method creates a new object. For this reason, we do not summarize methods that throw an exception or return an array type, as these operations typically allocate a new exception object or array.

Most native methods also interrupt recording, as we cannot observe their effects via the listener, though some may be manually whitelisted, if they are known to be safe and free from relevant effects, such as **println()**.

4. EVALUATION

In order to measure the effect of our summaries, we ran a number of experiments. For each experiment program, we run JPF with and without summaries enabled, and compare the time it takes until the search terminates.

4.1 Setup

The majority of the systems under test (SUTs) were picked from the Software Infrastructure Repository (SIR) [7], an initiative that aims to make Software Engineering research reproducible and comparable by providing a number of programs that can be used to evaluate testing and analysis techniques. Each SUT contains some type of fault that JPF is able to identify. Some SUTs also supply a version where the fault has been remedied, these are identified by the suffix `-fixed`.

Experiment	JPF-core (ms)	JPF-summary (ms)	Relative change
boundedBuffer	145.7	155.3	1.07
log4j1	237.1	261.7	1.10
log4j2	285.8	299.6	1.05
groovy	304.7	340.9	1.12
groovy-fixed	639.5	715.0	1.12
pool3	868.4	1218.9	1.40
pool6	1081.6	1268.5	1.17
linkedlist	1086.7	1410.0	1.30
pool4	1272.8	1451.4	1.14
log4j3	1474.0	1758.4	1.19
pool2	3014.2	3913.7	1.30
pool1	3061.6	3646.0	1.19
log4j3-fixed	11298.4	12301.4	1.09
log4j1-fixed	22358.4	24751.6	1.11
pool5	28818.2	29633.9	1.03
pool1-fixed	259168.7	322284.8	1.24
pool2-fixed	261422.8	324724.4	1.24
pool3-fixed	271131.1	343094.4	1.27
pool6-fixed	966102.5	1075927.3	1.11
loseNotify-stateless	85.8	96.8	1.13
twoStage-stateless	425.6	464.3	1.09
alarmclock-stateless	20950.4	22196.2	1.06
groovy-stateless	31786.2	34923.6	1.10
lang-stateless	116648.6	125244.6	1.07
pool3-stateless	386734.6	320813.3	0.83

Table 1: Mean run time of experiments, where the difference was significant for $\alpha = 0.1$, for normal JPF (top part) and JPF in stateless mode (bottom six rows).

In addition to looking at JPF's standard model checking procedure, we also ran our experiments with state matching turned off. This effectively causes JPF to perform stateless model checking. As state matching is one of the cornerstones for JPF's performance, removing it decreases performance significantly. Many of our experiment's programs caused JPF to run out of memory, or not terminate in several hours, in stateless mode.

All experiments were run on an Intel Core i5-4200M CPU 2.50 GHz with 8 GB of RAM, running Ubuntu 16.04 LTS, and Oracle's VM version 1.8.0_151, no experiment exceeded the default memory limit of 1 GB. For each experiment SUT we run JPF ten times and present the mean run times. Because the setup phase of JPF involves a large amount of variance, we only start measuring once the search procedure actually starts.

4.2 Run Time Results

Overall we see that our summaries have a negative or negligible impact on the run time of JPF [3]. Table 1 shows cases with a significant difference in the run time with and without summaries. The top part of the table uses JPF in normal mode (with state matching), while the last six entries show JPF without state matching (in stateless mode). A number greater than 1 indicates that execution took longer using summaries, while a smaller number means run time was reduced.

The size and complexity of the SUTs varies a lot, from the very small and simple **deadlock**, which is only 24 lines, to examples

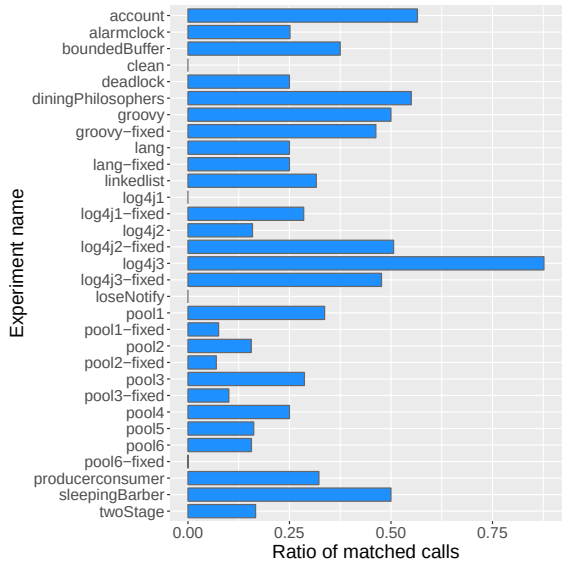

Figure 2: Percentage of calls to recorded methods in matching contexts for stateful experiments.

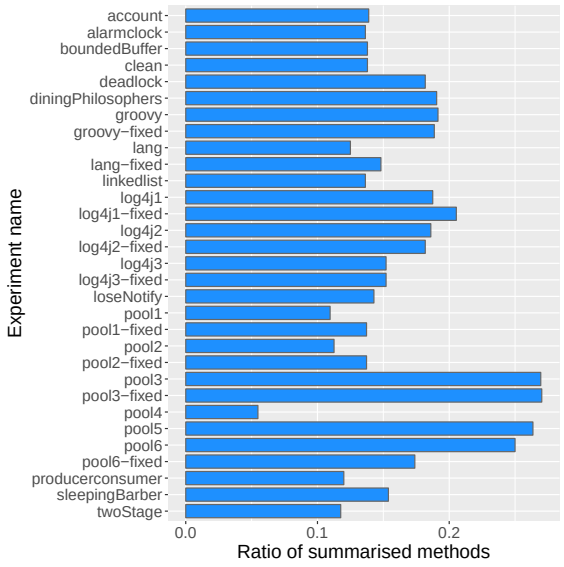

Figure 3: Percentage of methods that are summarised for stateful experiments.

from real bugs from open source projects such as **pool6**, which is 2043 lines [7]. While the larger targets are more interesting, the smaller are included for the sake of completeness.

We see that, for now, the overhead of creating and managing summaries mostly outweighs any performance gains when applying them. To look further into this, we also investigated how often summaries were actually used, and how often they could be created at all.

4.3 Usage of summaries

Having a recorded summary is not very useful if the method is never called in the same context a second time. Figure 2 shows the percentage of calls to methods where a summary is applicable (across all benchmarks). The differences in the percentages were small or non-existent when comparing the stateful and stateless model checking, though the absolute numbers were larger for some stateless experiments, as such we only present the stateful results. In some smaller programs, the ratio of applied summaries is very low or even 0 % as in **clean**, **log4j1**, and **loseNotify**. This is because a very small number of methods is recorded, and the methods are not executed often because the state space is small. On the other hand, 87% in **log4j3** shows a case where summaries are very effective. On average, we are able to apply summaries 29 % of the times we call summarised methods.

4.4 Interruptions

In our experiments, we see a large variance between different programs in terms of how many methods we are able to summarise, as seen in Figure 3, ranging from 5 % to 27 %. With these results, we again saw no significant differences between stateful and stateless model checking, so we only present the stateful results.

As presented in Section 3.3, there are multiple reasons why we might stop creating a summary. Figure 4 show a distribution of the reasons why summary recording failed on our benchmarks. The distributions remain the same when running in stateless mode. We can see that the primary cause is methods creating new objects. This is visible in Java bytecode as a call to **init**, the bytecode instruction corresponding to calling **new** in Java. The two other most common reasons are native methods that we cannot safely summarise, and that the execution of a method is interrupted by a transition break. Note that exceptions do not show up as a cause in Figure 4, because recording was always interrupted by **init**, which was used to create the Exception instance before it was to be thrown, in all experiments that we ran.

5. RELATED WORK

Method or procedure summaries have been used with some success in the domain of symbolic execution. Godefroid et al. use on-demand summaries for static analysis or test case generation to speed up static and dynamic analysis [10]. Sery et al. use summaries based on Craig interpolation to avoid analyzing functions again after an upgrade, if their effect has not changed [18].

Rojas and Pasareanu use partial evaluation to summarise the set of all symbolic paths of a method [17]. Each summary of a symbolic path represents specific path and heap constraints and contains a specialised version of the method code for those constraints. The specialised code of the summaries is guaranteed to contain no branch conditions, meaning that when it is applied, there will be no additional solver calls. The authors show that their summaries improve the run time at somewhat high memory costs [17].

Qiu et al. [16] expand on this work and build summaries using *memoisation trees*. These summaries do not encode the effects of methods, but rather the information about feasible paths in each method, and they improve the performance of SPF by an order of magnitude in certain cases [16].

In symbolic execution, summaries are particularly useful as they reduce the number of computationally intensive constraint solver calls; furthermore, the summary context is more likely to match in a symbolic rather than a concrete representation.

6. CONCLUSIONS AND FUTURE WORK

JPF uses a backtracking search to analyse the full state space of a program; because methods are executed repeatedly after backtracking, the same computations inside methods may be executed

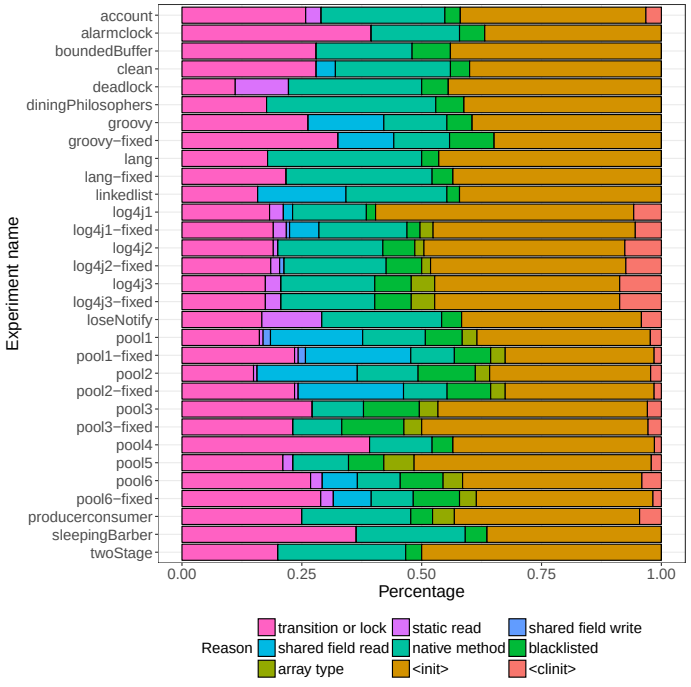

Figure 4: Interruptions for each experiment.

many times. Method summaries promise to eliminate redundant computations. A summary records in which program state a method call with given parameters produces a given result, including modifications to the program state.

We implement summaries that capture the effects of instructions sequences that are executed inside the same transition and do not create new objects or call native methods. With these summaries, the overhead of managing them usually but not always outweighs speed-ups gained by not executing the same code repeatedly. The reason for this is that larger methods usually cannot be summarised. The main cause of interrupting the creating of a summary is the creation of new objects.

If we can extend our approach to enable summaries to contain object creation, it would be potentially more impactful, as up to 50 % of all methods cannot be summarized due to object creations.

Another way to improve upon our implementation would be to make the summaries more general. Our current summaries are fixed to cover entire methods. Furthermore, even if only a single variable in the context changes, this requires an entirely new summary. In this context, it might be useful to create summaries over symbolic values [15] to enable sharing summaries in similar but not equal contexts. Finally, it may be possible to keep summaries across multiple executions.

7. REFERENCES

[1] Abdulla, P. A., Aronis, S., Atig, M. F., Jonsson, B., Leonardsson, C., and Sagonas, K. Stateless model checking for tso and pso. *Acta Informatica 54* (2017), 789–818.

[2] Artho, C., and Visser, W. Java Pathfinder at SV-COMP 2019 (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems — 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III* (2019), pp. 224–228.

[3] Berglund, L. Executive summaries in software model checking. Master's thesis, KTH, Theoretical Computer Science, TCS, 2018.

[4] Clarke, E. M., Grumberg, O., and Peled, D. *Model checking*. MIT press, 1999.

[5] Cobleigh, J. M., Giannakopoulou, D., and Păsăreanu, C. S. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2003), Springer, Springer, pp. 331–346.

[6] Dillig, I., Dillig, T., Aiken, A., and Sagiv, M. Precise and compact modular procedure summaries for heap manipulating programs. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 567–577.

[7] Do, H., Elbaum, S. G., and Rothermel, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal 10* (2005), 405–435.

[8] d'Amorim, M., Sobeih, A., and Marinov, D. Optimized execution of deterministic blocks in Java PathFinder. *Formal Methods and Software Engineering* (2006), 549–567.

[9] Godefroid, P. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1997), ACM, pp. 174–186.

[10] Godefroid, P., Nori, A. V., Rajamani, S. K., and Tetali, S. D. Compositional may-must program analysis: unleashing the power of alternation. *ACM Sigplan Notices 45*, 1 (2010), 43–56.

[11] Holzmann, G. *The SPIN Model Checker*. Addison-Wesley, 2004.

[12] Jhala, R., and Majumdar, R. Software model checking. *ACM Comput. Surv. 41* (Oct. 2009), 21:1–21:54.

[13] Jonker, C. M., and Treur, J. Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. In *Compositionality: The Significant Difference*. Springer, 1998, pp. 350–380.

[14] Musuvathi, M., and Qadeer, S. Fair stateless model checking. In *ACM SIGPLAN Notices* (2008), vol. 43, ACM, pp. 362–371.

[15] Păsăreanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., and Rungta, N. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering 20* (2013), 391–425.

[16] Qiu, R., Yang, G., Pasareanu, C. S., and Khurshid, S. Compositional symbolic execution with memoized replay. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on* (2015), vol. 1, IEEE, pp. 632–642.

[17] Rojas, J. M., and Pasareanu, C. S. Compositional symbolic execution through program specialization. *BYTECODE'13 (ETAPS)* (2013).

[18] Sery, O., Fedyukovich, G., and Sharygina, N. Incremental upgrade checking by means of interpolation-based function summaries. In *2012 Formal Methods in Computer-Aided Design (FMCAD)* (2012), IEEE, pp. 114–121.

[19] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model checking programs. *Automated Software Engineering 10* (2003), 203–232.