BUBEN: Automated Library Abstractions Enabling Scalable Bug Detection for Large Programs with I/O and Complex Environment

Pavel Parízek

Charles University, Faculty of Mathematics and Physics,
Department of Distributed and Dependable Systems

Abstract. An important goal of software engineering research is to create methods for efficient verification and detecting bugs. In this context, we focus on two challenges: (1) scalability to large and realistic software systems and (2) tools unable to directly analyze programs that perform I/O operations and interact with their environment. The common sources of problems with scalability include the huge number of thread interleavings and usage of large libraries. Programs written in managed languages, such as Java, cannot be directly analyzed by many verification tools due to insufficient support for native library methods. Both issues affect especially path-sensitive verification techniques.
We present the Buben system that automatically generates abstractions of complex software systems written in Java. The whole process has three phases: (1) dynamic analysis that records under-approximate information about behavior of native methods and library methods that perform I/O, (2) static analysis that computes over-approximate summaries of side effects of library methods, and (3) program code transformation that replaces calls of native methods and creates abstractions of library methods. Software systems abstracted in this way can be analyzed, e.g. for the presence of bugs, without the risk of a tool failure caused by unsupported libraries and more efficiently too. We evaluated Buben on several programs from popular benchmark suites, including DaCapo.

1 Introduction

An important goal of software engineering research is to develop techniques and tools for verification and detecting bugs in software. Out of the many associated challenges, we focus on these two: (1) scalability to realistic software systems that typically include many large libraries and (2) tools unable to directly analyze programs that perform I/O operations (including data storage, GUI and networking) and whose behavior depends on interaction with their environment.

The problems with scalability of verification and bug finding have several causes. For example, one cause is the huge number of thread interleavings that have to be analyzed even for rather small multithreaded programs. Another cause is the usage of large libraries, which greatly increases the total amount of code that a verification tool has to process and the time needed to analyze individual execution paths. Typical realistic programs involve many calls of library methods. However, in practice, developers are usually looking just for bugs in application programs that are relatively small when compared to all the libraries. Therefore, most library methods included with a program can be assumed correct, and many of them do not actually influence the program control flow and visible behavior — for these reasons, execution of such library methods does not have to be checked. In the case of managed languages such as Java, some library methods even involve native code, which is usually not handled well by the respective verification tools.

The presence of native library methods is, in fact, the main reason why many state-of-the-art verification tools cannot be directly used to analyze programs that access files, communicate over network, or involve GUI — where *directly* means without prior substantial modifications of the program code. We say that such programs *manipulate and interact with external entities* (e.g., with files and human users). For example, Java Pathfinder (JPF) [16], a popular verification framework, when run standalone it crashes on many realistic programs due to insufficient support for libraries that perform I/O via native methods.

Both challenges, that means (1) limited scalability to programs using large libraries and (2) inability to analyze programs that manipulate with external entities through I/O operations, affect especially path-sensitive verification techniques based on state space traversal, which aim to analyze every possible execution trace separately. Authors of verification tools usually focus on algorithmic improvements, neglecting the hard and tedious work needed to handle real-world programs that use many libraries and interact with external entities. For example, manually creating models (stubs) of the respective methods and abstractions of the environment is certainly not a practical option in general.

We present the Buben system that automatically creates abstractions of libraries in order to enable analysis of realistic programs with tools like Java Pathfinder. For an input large program, Buben computes abstractions of library methods called from within the application code and then generates an abstract variant of the program by the means of several code transformations. The abstract program does not contain any calls to library methods that are either native, perform I/O, or interact with external entities — thus avoiding calls of library methods that cause problems to verification tools. A particular tool (e.g., Java Pathfinder) can be then successfully run on the abstracted program without the risk of a failure due to missing support for library methods. In addition, usage of library abstractions generated by Buben helps to improve the performance and scalability of verification, because the total amount of code and possible behaviors to be analyzed is much lower. Even though we described the challenges that Buben addresses mostly on the specific case of Java Pathfinder, it can be used also together with other program analysis and verification tools quite easily. Only minor customizations of Buben are needed in order to support a new tool — for example, the user has to define the name of the tool's API procedure that performs a non-deterministic choice.

The key characteristic of the whole process involving Buben is nearly full automation. A user only has to provide a configuration file, which specifies the set of library methods and command line arguments.

In the rest of this paper, first we provide an overview of Buben, and then we present specific details in the following sections.

Overview. When generating abstractions, Buben distinguishes between library methods and the application classes. Only the library methods (and their calls) are to be abstracted. The set of library methods is then split further into two parts, where the first part contains native methods and library methods that manipulate with external entities (I/O), and the second part contains all other library methods. Each part is processed in a different way.

For the given input program, Buben creates its abstracted variant that consists of the original application code and generated abstractions of library methods. The whole process has the following three steps:

1. Dynamic analysis records information about side effects and outputs of library methods in the first group (native, I/O).
2. For every other library method, a summary of its possible side effects is computed using a static analysis.
3. Several code transformations are performed in order to create an abstract variant of the input original program.

We have to use dynamic analysis mainly because the side effects and outputs of native methods, respectively library methods that perform I/O, cannot be determined statically. In order to compute a summary of a given library method (step 2), Buben actually performs *symbolic interpretation* that is based on a linear traversal of method's code. Note that the dynamic analysis must run before the static analysis, since data collected by the dynamic analysis are used by the procedure for computing method summaries. For example, when the static analysis inspects the code of a method m, it needs to have information about possible side effects for its callees (even if they are native) in order to create a proper summary of m.

An important property of the computed summaries is that they approximate the original behavior of library methods. More specifically, Buben uses dynamic analysis to record under-approximate summaries for methods processed in the first step (native, I/O), while for all other library methods it computes over-approximate summaries of their externally visible behavior (side effects) with the help of static analysis.

The intentionally unsound dynamic summaries do not limit the practical usefulness of Buben, because it does not need sound summaries of library methods. It is even not possible to compute a sound general summary for a library method that performs I/O. More details are given in Section 2 and Section 3.

Generated abstractions of library methods have the form of code that reflects the results of static and dynamic analysis (approximate method summaries). Program code transformations (step 3) replace the calls and original implementations of library methods with the respective abstractions. We designed our transformations in a way that preserves mutual exclusion of accesses to individual object fields and array elements by different threads. This was necessary to avoid introducing spurious concurrency errors into the abstracted program. Again, we provide more details in Section 4.

The Buben system is not optimized towards any specific kind of properties and bugs — nevertheless, we had in mind especially fast detection of concurrency errors with tools like Java Pathfinder as our motivation. On the other hand, any bug finding approach that involves Buben would not be sound (i.e., errors could be missed) as a consequence of the under-approximate summaries of native and I/O library methods produced in the step 1 of the whole process. This is, however, not really a big issue with respect to our primary target use case — scalable detection of bugs within the application code.

Contribution. The main research contribution of this paper includes:

- The whole Buben system that combines dynamic analysis, static analysis (method summaries), and program code transformations in a specific way for the purpose of generating abstractions of large real-world programs, which can be subsequently analyzed using tools such as Java Pathfinder.
- Specific approach to dynamic recording of possible side effects and return values of library methods that is based on runtime interpretation of program code and inspection of program state (Section 2).
- Static analysis procedure for computing method summaries that is based on symbolic interpretation (linear traversal) of method's code (Section 3).
- Implementation of the Buben system for Java bytecode programs, and experimental evaluation on six programs selected from the DaCapo [3] and pjbench[1] suites. Results of our experiments show that while standalone Java Pathfinder crashes on all six benchmarks, usage of Buben helps to avoid the failures caused by insufficient support for libraries and enables JPF to find bugs in 4 programs out of 6.

The rest of this paper contains a section for each step of the whole process, followed by experimental evaluation, presentation of an example usage scenario, and discussion of related work.

2 Dynamic Recording

We apply dynamic analysis on a run of the input program to record a *dynamic summary* of each library method that is either native, performs I/O, or manipulates with external entities. A dynamic summary represents an under-approximation of the method's side effects that could be observed at runtime. It is a structure with four items: (1) a set of possible return values, (2) a set of updated object fields together with a set of possible new values for each field, (3) a set of updated array elements, again together with a set of possible new values for each element, and (4) a set of newly allocated objects.

[1] https://bitbucket.org/psl-lab/pjbench

Unlike most of the existing frameworks for dynamic analysis, including Road-Runner [7] and DiSL [9], which are based on code instrumentation, our approach involves runtime interception of program execution and inspection of dynamic states. In the rest of this section, we explain generally relevant technical aspects of the dynamic analysis and construction of the under-approximate dynamic summary. Figure 1 shows the key components of the analysis.

The following steps are performed when a call of a library method m subject to analysis (the sets *nativeMths* and *libextMths* in Figure 1) is reached.

1. Program execution is intercepted just at the entry to m.
2. Then, our analysis temporarily saves relevant parts of the program state at the time of entry to m — in particular, the content of arrays given as parameters of the method call.
3. Execution of the library method m is resumed.
4. Dynamic analysis intercepts the program execution again just at the exit (return) from the method m.
5. Next, the analysis temporarily saves the content of relevant arrays at the time of method exit.
6. All outcomes and side effects of the method's execution are recorded into the dynamic summary for m by handlers of respective events — this includes the return value (line 13), updates of object fields (line 23) and array elements (line 36), and newly allocated objects (line 27).

Our analysis does not record values of method call parameters in the dynamic summary, because the parameter values are not needed to generate abstractions (see details in Section 4). The content of arrays given as parameters is saved temporarily just for the purpose of creating a list of array elements updated by the method, which is then recorded in the summary.

Updated array elements are identified through the search for differences between two snapshots of a given array — its old content at the method entry (variable *oldArray* at line 32) and new content at the method exit (variable *curArray* at line 33). These snapshots of array content are saved in the respective handlers (lines 5-8 and 14-17), and then corresponding array elements are compared pair-wise. For performance reasons, as the fast track path we use hash values to find out whether the array was modified at all by the method. We are aware that collisions of hash values may occur, but dynamic summaries capture under-approximations of the sets of possible side effects anyway.

Possible new values of object fields and array elements are saved as symbolic expressions — constant values, access paths, or arithmetic expressions. An access path is a local variable name followed by a sequence of field names.

From the perspective of usage in Buben, a very important aspect of the dynamic analysis is that it has to record only updates to object fields and arrays either defined within application classes or visible from them. Accesses to other variables can be safely ignored, such as those internal to libraries. This is practically realized by tracking just updates to objects given as call arguments to library methods or returned from them.

```
INPUT : $nativeMths$, $libextMths$

procedure onMethodEntry( $mth$, $args$ )
   for $arg \in args$ do
      if $arg$.isArray then
         saveArrayContentAtEntry( $mth$, $arg$ )
         computeArrayHashAtEntry( $mth$, $arg$ )
         markTrackedArray( $mth$, $arg$ )
      end if
   end for

procedure onMethodExit( $mth$, $res$ )
   recordCallReturnValue( $res$ )
   if $res$.isArray then markTrackedArray( $mth$, $res$ )
   for $arr \in$ trackedArrays( $mth$ ) do
      saveArrayContentAtExit( $mth$, $arr$ )
      computeArrayHashAtExit( $mth$, $arr$ )
   end for
   recordArrayDifferences( $mth$ )

procedure onFieldWrite( $loc$, $field$, $newVal$ )
   if isMethodArgType( $loc$.method, $field$.class ) then
      recordFieldUpdate( $loc$.method, $field$, $newVal$ )
   end if

procedure onNewObject( $loc$, $obj$ )
   recordNewObjectAlloc( $loc$.method, $obj$ )

procedure recordArrayDifferences( $mth$ )
   for $arr \in$ trackedArrays( $mth$ ) do
      if hashAtEntry( $arr$ ) == hashAtExit( $arr$ ) continue
      $oldArray$ = getArrayContentAtEntry( $mth$, $arr$ )
      $curArray$ = getArrayContentAtExit( $mth$, $arr$ )
      for $i \in 0 \ldots$ length( $curArray$ ) $-1$ do
         if $oldArray( i )$ != $curArray( i )$ then
            recordArrayUpdate( $mth$, $arr$, $i$, $curArray( i )$ )
         end if
      end for
   end for
```

Fig. 1. Dynamic analysis that records side-effects of library methods

Implementation. Here we describe selected technical details that affect the performance and practical applicability of the dynamic analysis.

We have implemented the dynamic analysis on top of the JPDA framework[2] that is a part of the Java platform. In particular, we have used two components of JPDA — the JVM TI monitoring interface and the JDI front-end API.

[2] https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html

JDPA provides all the necessary information about runtime program behavior and states through its API, including call arguments at method entry and return values at method exit, and it supports tracking of calls to native methods. Moreover, JPDA allows to inspect heap data structures, dynamic call stack, and runtime values of program variables (object fields, array elements) — something that is not easily achievable probably in all the dynamic analysis frameworks based on code instrumentation (e.g., RoadRunner [7] or DiSL [9]).

Based on our preliminary experiments, we have also found that the performance overhead of a dynamic analysis based on JVM TI (with respect to normal program execution) is much smaller if we separated tracking of method entry (invoke) events from method exit (return) events — namely, such that all method entry events are tracked in one run of the dynamic analysis, while all the method exit events are tracked in another distinct run. This optimization helps to achieve at least practical running times in our scenario.

In order to keep also the size of dynamic summaries within practical limits, so they can be applied during the program transformation step, we had to put an upper bound on the number of recorded possible distinct return values from a library method. This is motivated especially by certain system library methods, such as System.getCurrentTimeMillis, that are invoked many times during the run of a program and return a different value on each occasion. We set the value of the upper bound to 256 for our experiments; however, it is configurable.

3 Static Computation of Method Summaries

We use static analysis to compute over-approximate summaries of possible side effects for the remaining library methods, which do not perform I/O and do not interact with any external entities. In this section, first we define the content of method summaries, and then we present our algorithm for computing them.

Summaries. We designed summaries that capture just externally-visible side effects of the library methods' execution that may influence runtime behavior of the application part of the whole program. This includes all the changes of a runtime program state that may occur during execution of a given library method, and that may be visible in the scope of application code.

Therefore, here we define a *static method summary* as a data structure that contains the following items:

– A list of all object fields possibly updated in the method.
– A list of all possibly updated static fields.
– A list of all array elements updated in the method.
– For each updated field and array element, a list of possible new values.
– A boolean flag saying whether the method may return a value corresponding to its arguments, and indexes of the respective arguments.
– A set of all objects newly allocated in the method, including arrays.
– A boolean flag saying whether the method may return a new object.
– A set of all possible return values.

– A list of fields and array elements updated outside of any code region that is protected by a lock or through another mechanism of thread synchronization.

Data are stored as symbolic access paths, respectively symbolic expressions (constants, variable names, new objects, arithmetic). A *symbolic access path* begins with a local variable name and then contains any valid mixture of field names and indexed accesses to array elements (such as `o.f.a[i].g`). Only those symbolic expressions that involve local variables other than method arguments cannot be recorded in summaries, because they are not visible in the application code.

The static summary of a library method m captures also the results and side effects for all methods called within the scope of m.

Algorithm for computing summaries. Our static analysis-based approach combines (1) a fixpoint worklist algorithm over the list of all reachable methods with (2) linear symbolic interpretation of the code of every method.

However, as a prerequisite of the main algorithm, it is necessary to compute the set of may-aliased access paths for each local variable that may appear on the left-hand-side of some assignment. We use the aliasing information to properly handle code such as **void** `myproc(Obj o)` { `v` = `o`; `v.f.g` = 2; } where the field update is effectively performed also upon the parameter `o`. The procedure for computing the sets of possibly aliased expressions repeatedly processes the list of assignment statements, until it reaches a fixed point.

Our approach for computing static method summaries was inspired by Naeem and Lhotak [11]. At the start of a run of the main algorithm, the worklist is filled with all library methods reachable in the call graph. In each iteration, the algorithm removes a method m from the head of the worklist and performs intra-procedural analysis of m (see below), which collects the necessary information about side effects of m and updates its summary. When the summary of m changes, all the callers of m that belong to the set of library methods are added into the worklist, because their summaries depend on m and have to be recomputed. The algorithm terminates when the worklist is empty, at which occasion the summary of each method captures all its results and side effects.

Summaries of individual methods are generated by *symbolic interpretation* that is based on linear code traversal. We do not have to use a full-fledged static analysis that involves fixpoint computation over the method's control-flow graph and SSA IR. Our symbolic interpretation of the code of a library method is an intra-procedural control-flow-sensitive analysis that recognizes all side effects of the method, together with other information that make up its summary. By the term *control-flow-sensitive*, we mean that our analysis distinguishes among control-flow branches within the method's code, but does not process each control-flow path separately. In this respect, our notion of control-flow-sensitivity is a weaker form of path-sensitivity. The analysis traverses the sequence of instructions just once in a linear fashion, i.e. it performs only a single linear pass through the code of a given method.

Figure 2 illustrates the key aspects of our symbolic analysis — specifically, how it processes control-flow branches and selected instructions. During its run, our symbolic interpretation algorithm maintains a stack of expressions that is

```
1  for insn ∈ mth.instructions do
2     setActiveControlFlowBranch()
3     if insn.type == condBranch then
4        if insn.jumpTarget > insn.index then
5           startNewControlFlowBranch(insn.jumpTarget)
6        end if
7     end if
8     if insn.type == goto then
9        if insn.jumpTarget > insn.index then
10          startNewControlFlowBranch(insn.jumpTarget)
11          suspendCurrentBranch()
12       end if
13    end if
14    if insn.type == getfield then
15       obj = removeExprFromStack()
16       addExprToStack(obj + "." + insn.fieldName)
17    end if
18    if insn.type == arraystore then
19       newValue = removeExprFromStack()
20       indexExpr = removeExprFromStack()
21       arrayObj = removeExprFromStack()
22       recordArrayUpdate(arrayObj, indexExpr, newValue)
23    end if
24    ...
25 end for
```

Fig. 2. Key aspects of symbolic interpretation with control-flow sensitivity

used to store instruction operands and results. Handlers for individual instructions manipulate with the stack. For illustration, in Figure 2 we show handlers for the getfield and arraystore instructions (lines 14-17 and 18-23, respectively).

When the analysis processes a method m, it propagates available summaries of other methods called from within m (transitively) into the summary of m to reflect their side effects and outcomes. At each call site, data about formal parameters of the callee (including `this`) are associated with the actual arguments. Existing dynamic summaries are used for the calls of native methods inside m. In the case of library methods that manipulate with external entities, our algorithm uses the results of dynamic analysis to initialize their summaries and refines them later during the run of static analysis. The set of possible return values from m is computed through a backward traversal of def-use chains and nested method calls that starts at the explicit return statements. If a possible new value of an updated field, respectively updated array element, is a symbolic expression that refers to the return value of another method m' called inside m, then it is expanded with actual return values captured in the summary of m'.

Now we describe the approach for processing of control-flow branches. At every moment during the run of our symbolic interpretation, one control-flow branch is marked as active. The active control flow branch is changed if needed just before processing of an instruction (line 2) — e.g., when the current instruction is the target of a forward jump. Upon reaching of a conditional branch (jump) instruction, the interpreter creates a new branch and schedules the branch to be active at the jump's target location (line 5). Execution of the current active branch then continues at the next instruction. Goto instructions are processed in a slightly different way. The interpreter creates a new control-flow branch and makes it active at the target location of a jump also in this case. However, the currently active branch is then suspended until the instruction corresponding to the jump target is reached (line 11). Another branch will then become active, i.e. its execution will resume, at the next instruction in a sequence, which must be a target location of another jump from elsewhere in the method. Our approach ensures that, for every instruction that may be a jump target, the symbolic interpreter distinguishes between all the possible symbolic stack contents at the instruction. Note also that our interpreter can safely ignore backward jumps, because they do not start new control-flow branches.

Like in the case of the dynamic analysis (Section 2), our algorithm for computing static summaries records just updates to object fields and arrays defined either in the application classes or visible from them. This is again realized by tracking only updates to method call arguments. Information about variables internal to library methods are not used for constructing abstractions (Section 4).

Implementation. We implemented the algorithm for computing static method summaries on top of the WALA library[3]. A very important aspect of our implementation is the usage of fast but imprecise approach to call graph construction (0-CFA), which finishes quickly even for large software systems. BUBEN does not need a precise call graph when generating summaries of library methods. In order to compensate for the imprecision of alias analysis, our symbolic interpreter considers as possible new values of updated fields and array elements only the symbolic expressions whose prefix is the source value of some assignment statement within the method. We apply this optimization also on the set of possible return values. This greatly improves the precision of method summaries.

4 Program Code Transformations

We already said that BUBEN creates an abstraction of the given program by the means of code transformations. An input for this procedure consists of (1) the original program code and (2) method summaries computed either by the dynamic analysis or static analysis. Method summaries contain all the information needed to generate abstractions of the respective library methods.

BUBEN performs especially two kinds of program code transformations:

- Replacing the calls of native methods and library methods that perform I/O or manipulate with external entities.
- Creating new abstract implementations (bodies) of all other library methods.

[3] T.J. Watson Libraries for Analysis (`http://wala.sourceforge.net/`)

We want to emphasize that the only parts of application code affected by these transformations are the calls of library methods.

An abstraction of a library method is generated by a procedure that follows the template in Figure 3. It specifies how the data captured by a method summary are translated into actual code, i.e. how the abstraction looks like.

```
$summm$ = retrieveMethodSummary($mth$)

generateBeginAtomic()

for ($o.f, vals$) $\in$ $summ$.updatedObjectFields do
   if $o.f \in$ $summ$.unsynchFieldAccesses continue
   $v$ = generateNondetChoiceOverSet($vals$)
   generateFieldUpdate($o, f, v$)
end for

for ($a[i], vals$) $\in$ $summ$.updatedArrayElements do
   if $a[i] \in$ $summ$.unsynchArrayAccesses continue
   $v$ = generateNondetChoiceOverSet($vals$)
   generateArrayUpdate($a, i, v$)
end for

generateEndAtomic()

for $o.f \in$ $summ$.unsynchFieldAccess do ...
for $a[i] \in$ $summ$.unsynchArrayAccesses do ...

$c$ = generateNondetIntChoice(0, $summ$.returnValues.size)
generateLoadExpression($summ$.returnValues[$c$])
```

Fig. 3. Template for method abstraction

The same template is used by the module for program code transformation both (1) to replace calls of library methods and (2) to create their new implementations. Only low-level adjustments have to be made in each case to ensure the abstraction seamlessly fits into the existing code around the target location. For example, when replacing a call of some library method, it is also necessary to generate code that removes original method call arguments from the stack.

When generating the abstract variant of the body of a library method, the first step is to remove the whole original control-flow structure. New statements defined by the template can then be inserted in any order, because there are no dependencies between them. Our transformation procedure does not strive to preserve the order of statements that corresponds to the original method body, for two reasons: (1) method summaries do not capture the order of statements anyway; (2) in the main use case for BUBEN, verification tools will receive only the transformed program as input (with the new order of statements in abstracted methods), without any reference to the original program.

Our template supports both (i) updates to fields, respectively array elements, that are protected by some kind of thread synchronization in the original program, and (ii) updates to the other fields and array elements (lines 19-20). Protected updates are enclosed within a single atomic block (lines 3 and 17). Besides that, no special processing of multithreading-related code is needed. Calls of library methods that control threads and synchronization (such as Thread.start and Object.wait) are not affected by code transformations in any way. In general, transformations preserve concurrency-related behavior, including concurrent accesses to shared variables, and therefore also possible concurrency bugs.

Implementation. We used the ASM bytecode manipulation framework for Java[4] to implement all the program code transformations. Generated abstractions call the API for non-deterministic choice provided by a target verification tool (e.g., Java Pathfinder).

5 Evaluation

We evaluated the implementation of Buben on multiple large Java programs taken from popular benchmark suites, including DaCapo [3] and pjbench that is available at the url `https://bitbucket.org/psl-lab/pjbench`.

The list of programs from DaCapo contains batik, lusearch, pmd, and sunflow. We used the smallest available configuration for each of them, and in some cases (e.g., sunflow) we set the number of threads to 2 — all that with the goal of reducing the running time of dynamic analysis, because it needs to observe just few executions of each library method in order to create an under-approximation that is useful as input for program code transformations. In addition, we picked the jspider benchmark from the pjbench suite and the SPECjbb2005 benchmark.

We decided to choose these specific programs because of their size, high degree of concurrency, and usage of many libraries. Program size was the relevant criterion especially in the case of DaCapo, because tools like Java Pathfinder do not yet scale to really large programs that are included within the DaCapo suite (e.g., Tomcat and Eclipse). On the other hand, most other programs in the pjbench suite (besides jspider) are quite small.

Source code of the Buben system, together with small examples, scripts, and configuration files needed to run all experiments, is available at `https://github.com/d3sformal/buben`.

We organized our evaluation around the goal of answering the following research questions related to practical usefulness of Buben:

Q1) Does the process of generating abstractions preserves interesting behaviors of the original input program and bugs present in its source code?

Q2) Whether the generated abstract program can be successfully analyzed by verification tools such as Java Pathfinder?

Q3) How much time it takes Java Pathfinder to analyze the abstracted program and to find real bugs in the application code?

[4] http://asm.ow2.io/

	Init (CG)	Dynamic analysis	Computing summaries	Program transform
batik	15 s	1285 s	3 s	17 s
lusearch	9 s	13665 s	4 s	10 s
pmd	9 s	2396 s	1 s	9 s
sunflow	12 s	11695 s	1 s	9 s
jspider	9 s	94 s	1 s	13 s
specjbb	6 s	40680 s	1 s	5 s

Table 1. Running time of Buben

Our answer to the first question is positive based on the way Buben creates abstract programs. Since just library methods and their calls are replaced with corresponding abstractions, other parts of the application code are not affected by the respective transformations, and therefore interesting behaviors of the program at the application level (including bugs) are preserved by construction. The remaining questions 2 and 3 have to be answered empirically.

Table 1 shows the running times of the main components of Buben. For each program, we measure the running times of initialization (which includes call graph construction), dynamic analysis, static computation of method summaries (that includes alias analysis), and program code transformations.

Data presented in Table 1 indicate that our approach is practically feasible. However, individual steps of the whole process still have to be optimized — especially the dynamic analysis, which run over 11 hours in the case of SPECjbb2005.

We also run Java Pathfinder (JPF) [16] on each generated abstract program to see whether JPF can analyze it successfully and find some bugs. For that, we had to extend the implementation and configuration of Buben to accommodate three special features of JPF: (1) use of a custom Java virtual machine, JPF VM, (2) hand-written models (stubs) for selected classes from the Java standard library, and (3) hand-written plain Java models for selected native methods. Some of the models for native methods used by JPF implement key aspects of the JPF VM functionality, and therefore we tweaked Buben to preserve calls of the respective native methods — including, for example, most of native methods defined in the class java.lang.Thread. The variant of Buben tailored for JPF also automatically generates simple models for those native methods, which are invoked from within standard Java library classes but for which the hand-written models do not yet exist in the JPF distribution, in order to avoid crashes of JPF. It is necessary because modifications of classes from the standard Java library cannot be saved persistently, meaning in particular that replacing the calls of native methods by code transformations described in Section 4 is not applicable in the case of such classes. Finally, we manually configured Buben to generate abstractions also for some application methods that perform lot of I/O-related actions or load classes explicitly via reflection.

The results of applying JPF both on the original programs and transformed programs (abstractions created by Buben) are presented in Table 2. We provide the descriptions of reported crashes and bugs, together with execution times.

	Original input program		Transformed abstract program (Buben)	
	result description	time	result description	time
batik	crashed: exception inside libraries	1 s	found bug: unhandled null pointer exception	1 s
lusearch	crashed: exception in file I/O	1 s	found bug: unhandled null pointer exception + deadlock	1 s
pmd	crashed: incomplete stubs for libraries	1 s	found bug: uncaught file--not-found exception	1 s
sunflow	crashed: exception inside libraries	1 s	did not report any error: run out of memory (12 GB)	(764 s)
jspider	crashed: incomplete stubs for libraries	1 s	found bug: unhandled null pointer exception	1 s
specjbb	crashed: incomplete stubs for libraries	1 s	did not report any error: run out of memory (12 GB)	(3073 s)

Table 2. Experiments with JPF

The left part of Table 2 shows that JPF crashed for all the original programs, and thus failed (i) to verify their application code and (ii) to find at least some bugs in them. For three programs, JPF crashed due to incomplete stubs of library methods and classes. In the other cases, JPF reported an uncaught exception thrown deep within libraries responsible, e.g., for GUI or file I/O.

The right part of Table 2 shows that, with abstractions generated using Buben, JPF could successfully analyze all six programs and even find bugs in four of them (batik, lusearch, pmd, jspider) very quickly. Unhandled null pointer exceptions in the case of batik, lusearch, and jspider were caused by missing checks for null references in the application code, while the uncaught file-not-found exception reported for pmd is related to a file actually present within the software package. By manual inspection of the source code, we checked that these particular bugs (detected in abstractions) exist also in the original programs.

Overall, results for the programs that we used in our experimental evaluation, presented in both tables, indicate that Buben is useful. It can generate abstractions of realistic large programs, which are amenable to verification and search for bugs by tools like JPF. Scalability of verification could be further improved by marking additional classes and methods as libraries in the user configuration.

6 Example Use Case

We illustrate the usage of Buben in more detail on the jspider program that we used also in our evaluation. JPF crashes when run on the original version of jspider, which is the main reason why one might consider to use Buben.

First, the user must define the configuration of Buben, similar to our example in Figure 4. It specifies the main class (entry point) of the program, command-line arguments, Java packages that represent libraries, and Java packages containing methods that manipulate with external entities (files, network).

As the second step, the user executes Buben to generate abstraction of each library method in the call graph. Figure 5 shows both the original code (at

```
mainclass = net.javacoding.jspider.JSpiderTool
runtimeargs = download, www.google.com, index.html
libmethods = org.apache.commons.logging, junit, org.apache.log4j, \
    org.apache.commons.collections, org.apache.log, org.apache.velocity
appclasses = org.javacoding.jspider
externmethods = java.io, java.net, java.nio, javax.mail, javax.jms, jdk.net
```

Fig. 4. Example configuration of Buben for jspider

the top) and transformed code (bottom) of the method warn from the class org.apache.log4j.Category. We picked this method because its code is quite short and therefore suitable for illustration purposes. Deeply nested within the call of forcedLog at line 5 (original variant) is a synchronized access to the field LogRecord._seqCount that is captured by a static summary of the method warn. Abstracted variant contains the field access inside an atomic block. The Verify class is a part of the JPF API.

The last step is to run JPF on the abstracted program in order to find bugs.

```
1  // original
2  public void warn(Object message) {
3    if (repository.isDisabled(Level.WARN_INT)) return;
4    if (Level.WARN.isGreaterOrEqual(getEffectiveLevel())) {
5      forcedLog(FQCN, Level.WARN, message);
6    }
7  }
8
9  // transformed
10 public void warn(Object message) {
11   Verify.beginAtomic();
12   int c = Verify.getInt(0,0);
13   if (c == 0) LogRecord._seqCount += 1;
14   Verify.endAtomic();
15 }
```

Fig. 5. Library method Category.warn: original (top) and transformed code (bottom)

7 Related Work

Lot of existing work is related to Buben and its components. In particular, we know about (1) few approaches with similar goals as the whole Buben system and (2) several techniques closely related to static and dynamic analysis performed by Buben during its run. We characterize the related approaches and techniques briefly in this section, and compare them with our solution.

Tkachuk and Dwyer [14] proposed an approach based on an idea similar to ours. It creates a model of environment for each component of a given system in order to enable its modular verification. Environment models correspond to sets of method summaries, which are computed by an intra-procedural flow-sensitive and context-sensitive side effect analysis. The main difference between this approach [14] and BUBEN is that the former captures just updates of the target component's state in order to create a minimal valid abstract environment, while the analyses performed by BUBEN collect all visible side effects of library methods. Other differences include (1) usage of dynamic analysis within BUBEN, which enables creating summaries for library methods that perform I/O, (2) better support for concurrency, such as tracking fields accessed by multiple threads, and (3) experimental evaluation on large programs.

State-of-the-art program verification frameworks handle libraries and I/O in different ways. For example, KLEE [4] uses a symbolic file system where effects of read and write operations are captured by constraints. Java Pathfinder contains manually created stubs for I/O library methods (but only the most often used are supported). Ceccarello and Tkachuk [5] improved the situation by developing a tool for automated construction of abstract property-specific models of library methods, which can be used only together with Java Pathfinder. The tool is based on two key ideas: (1) use of program code slicing that removes accesses to every field not relevant with respect to a given property, and (2) abstraction of read accesses to irrelevant fields by random values or default values of the respective data types. On the contrary, BUBEN automatically generates abstractions of library methods based on dynamic and static summaries that are computed by the corresponding analyses.

Many static analysis-based techniques for computing method summaries were proposed recently [6, 15, 11, 10, 13, 1, 12]. Each technique in this group computes some information about possible behavior and side effects of library methods, but none of them is directly applicable in our case — especially because none generates summaries that contain all the information that BUBEN needs, including new values of updated object fields and array elements.

For example, the analysis proposed by Cherem and Rugina [6] computes just the following information: a set of object fields updated by each method (up to a given bound on the length of field access chains), which fields of objects given as method call arguments may become aliased, and whether a returned value may be aliased with some argument or with a field of some argument.

The technique of Matosevic and Abdelrahman [10] computes method summaries that involve symbolic access paths (especially to object fields). It also uses pointer analysis to determine aliasing between method call arguments. The main difference from BUBEN is that, in our approach, we do not need to track heap locations and their possible aliasing.

In general, there is a large space of static analyses that compute summaries of some kind, where each technique is tailored for a particular use case.

One can also use slicing [2, 8] to create a simplified version of an input program for the purpose of efficient and scalable verification. Nevertheless, program slicing is done with respect to some property, and takes into account dependencies between statements and threads. Buben completely replaces the original code of library methods in a general property-independent manner.

Acknowledgments. We would like to thank Ondřej Lhoták for all his suggestions regarding the paper content and presentation. This work was partially supported by the Czech Science Foundation project 18-17403S.

References

1. S. Artzi, A. Kiezun, D. Glasser, and M. Ernst. Combined Static and Dynamic Mutability Analysis. In Proceedings of ASE 2007, ACM.
2. D. Binkley and K.B Gallagher. Program Slicing. Advances in Computers, 43, 1996.
3. S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In Proceedings of OOPSLA 2006, ACM.
4. C. Cadar, D. Dunbar, and D.R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proceedings of OSDI 2008, USENIX.
5. M. Ceccarello and O. Tkachuk. Automated Generation of Model Classes for Java PathFinder. In Proceedings of Java Pathfinder workshop 2013, ACM SIGSOFT Software Engineering Notes, 39(1), 2014.
6. S. Cherem and R. Rugina. A Practical Escape and Effect Analysis for Building Lightweight Method Summaries. In Proceedings of CC 2007, LNCS, vol. 4420.
7. C. Flanagan and S.N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In Proc. of PASTE 2010, ACM.
8. D. Giffhorn and C. Hammer. Precise Slicing of Concurrent Programs. Automated Software Engineering, 16(2), 2009.
9. L. Marek, A. Villazon, Y. Zheng, D. Ansaloni, W. Binder, and Z.Qi. DiSL: A Domain-Specific Language for Bytecode Instrumentation. In Proceedings of AOSD 2012, ACM.
10. I. Matosevic and T.S. Abdelrahman. Efficient Bottom-up Heap Analysis for Symbolic Path-based Data Access Summaries. In Proceedings of CGO 2012, ACM.
11. N.A. Naeem and O. Lhotak. Faster Alias Set Analysis Using Summaries. In Proceedings of CC 2011, LNCS, vol. 6601.
12. A. Rountev, M. Sharp, and G. Xu. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In Proceedings of CC 2008, LNCS, vol. 4959.
13. A. Salcianu and M. Rinard. Purity and Side Effect Analysis for Java Programs. In Proceedings of VMCAI 2005, LNCS, vol. 3385.
14. O. Tkachuk and M. Dwyer. Adapting Side Effect Analysis for Modular Program Model Checking. In Proceedings of ESEC/FSE 2003, ACM.
15. G. Yorsh, E. Yahav, and S. Chandra. Generating Precise and Concise Procedure Summaries. In Proceedings of POPL 2008, ACM.
16. Java Pathfinder verification framework (JPF), `https://github.com/javapathfinder/jpf-core/wiki`