

Automated Generation of Model Classes for Java PathFinder

Matteo Ceccarello
Department of Information Engineering
University of Padua
Padova, Italy
ceccarel@dei.unipd.it

Oksana Tkachuk
SGT, Inc./NASA Ames Research Center
Moffett Field
California 94035
oksana.tkachuk@nasa.gov

ABSTRACT

Model checkers like Java PathFinder (JPF) often have to combat the state space explosion problem. One solution adopted to tackle this problem is to abstract away parts of the system, e.g., to model complex library classes at a higher level of abstraction. The model classes have the same interface as the actual library classes but exhibit reduced behaviour and state. Writing such model classes is both error prone and time consuming. In this paper we propose a tool that can automatically derive a model class from the original class. To achieve this goal, the tool uses different algorithms, including slicing and value generation, each yielding a model class with different behaviour and state.

Keywords

Model Checking, Slicing, Code Generation

1. INTRODUCTION

Model checkers, such as Java PathFinder (JPF), are affected by the notorious state space explosion problem. Most Java applications rely on external libraries. When model checking these applications, the dependence on external libraries intensifies the state space explosion problem. These library classes may contain many fields that are not relevant to the verification effort, yet contribute to the size of the state space.

One approach used to mitigate this problem is to model library classes and force the model checker to use them instead of the actual library classes. Library models have the same interface as the actual libraries, so the model checker can use them in place of the original classes. One strategy for modeling library classes is to partition library classes' fields into those that are relevant and irrelevant to the application properties being checked. The main idea is to remove irrelevant fields and behavior related to them from the library classes, while retaining the relevant fields and behavior related to their state.

The JPF model checker [9] employs library models, along with native peers, to tackle the state space explosion problem. Many extensions of JPF use library models: `jpf-concurrent` [8] models the Java Concurrency Utilities; `jpf-awt` [4] provides model classes for the Java AWT/Swing libraries; `net-iocache` [1] provides models for the networking capabilities.

One of the main drawbacks of the model class approach is

that the model classes are usually written by hand. This is a tedious and error prone task. Developers should first identify the classes that are needed to verify an application, then they should determine which methods they need to implement. Finally the behaviour of the model class has to be defined, as well as its state. A tool to free the developer from the burden of writing model classes would be useful.

In our previous work [2, 7], we have developed tool support for automated model generation. The tool presented in [2] generates minimal models for the library classes needed by an application. The generated models are minimal in the sense that they contain only the methods needed by the application. Moreover these methods are modeled as empty stubs, with no behaviour. The developer may then add the desired behaviour using the stubs as a starting point. The approach in [7] employs the side-effects analysis to analyze the data interactions between the application under test and the libraries. The side-effects analysis is capable of determining methods that have no side-effects on the data of interest; such methods can be safely modeled with empty stubs. For other methods, the analysis calculates whether some data may be modified, e.g., whether an array or an integer field has been written to. However, the analysis may not be able to distinguish between adding or removing from an array or between incrementing or decrementing an integer value. In [5] a tool called `jpf-nhandler` is described. This tool addresses the problem of delegating on-demand and automatically native calls during a JPF run. The purpose of this tool is complementary to the purpose of the tool presented in this paper that deals only with model classes, whereas `jpf-nhandler` deals only with native calls.

In this paper, we propose and evaluate two additional approaches for model generation, based on slicing and value generation (using default or random values). These algorithms generate models with behaviour derived from the original class. The behaviour that is generated is specified by giving to the tool a set of fields that are relevant for the analysis. The two algorithms differ in how they handle the dependencies on unwanted fields.

The paper is organized as follows. The next section presents an example used throughout the description of the algorithms. Sections 3 and 4 present definitions and describe the two algorithms. Finally we analyze a case study and draw our conclusions.

Listing 1: Example application

```
1 import java.io.ByteArrayOutputStream;
2
3 public class ByteArrayOutputStreamCount {
4
5     public static void main(String[] args) {
6         ByteArrayOutputStream baos =
7             new ByteArrayOutputStream();
8
9         baos.write('h');
10        baos.write('e');
11        baos.write('l');
12        baos.write('l');
13        baos.write('o');
14
15        baos.write(new byte[]
16            {'w','o','r','l','d'}, 0, 5);
17
18        int count = baos.size();
19
20        String s = baos.toString();
21
22        System.out.println(s);
23        System.out.println(count);
24    }
25 }
```

2. MOTIVATING EXAMPLE

Consider the application given in Listing 1. This simple application writes some characters to a `ByteArrayOutputStream`, retrieves the count of the written bytes and then prints it along with the string representation of the stream contents. Suppose that the property we want to check is that the variable `count` never goes below zero. To do so we can use JPF’s `NumericValueListener`.

Using the standard library version of `ByteArrayOutputStream`, JPF can correctly verify that the `count` variable does not go below zero. However, to verify this property, only parts of the `ByteArrayOutputStream` class, related to the state of the field `count` are needed. The updates to the internal buffer of the output stream could be ignored, since the string that is returned by the `toString` method is not relevant to this analysis.

Listing 2 contains the source code of the standard library version of some methods of class `ByteArrayOutputStream`. The method `write` is the one used directly by the application. The other two private methods are called by `write` and can modify the state of the object. Methods `toString` and `size` are the other two that are used in the example application. Figure 1 shows the control flow graph (CFG) of the `write` method. The CFG nodes represent statements and the edges represent the control flow between the statements. For checking the example’s sample property, statements affecting the buffer `buf` are not relevant. For instance, on line 15 of Listing 2 we have a method call that can only modify `buf`. Thus this statement is not relevant for the analysis of the value of `count`, since `count` does not depend on `buf`.

Next, we describe our algorithms used to generate model classes with reduced behaviour and evaluate if these models

Listing 2: Source code of part of the standard library version of `ByteArrayOutputStream` class.

```
1 public class ByteArrayOutputStream
2 extends OutputStream {
3
4     private int count;
5     private byte[] buf;
6
7     public synchronized void write(
8         byte b[], int off, int len) {
9         if ( (off < 0) || (len < 0) ||
10            (off > b.length) ||
11            ((off + len) - b.length > 0) ) {
12             throw new IndexOutOfBoundsException();
13         }
14         ensureCapacity(count + len);
15         System.arraycopy(b, off, buf, count, len);
16         count += len;
17     }
18
19     public synchronized int size() {
20         return count;
21     }
22
23     public synchronized String toString() {
24         return new String(buf, 0, count);
25     }
26
27     // other methods ...
28 }
```

enable the property verification for the application under test.

3. DEFINITIONS

The *state* of a class C is the set of its static and non-static fields \mathbb{F}_C . In this paper we also refer to the *relevant state* of a class, a set $\mathbb{R}_C \subseteq \mathbb{F}_C$ of fields we are interested in preserving in the models. For example, consider the `ByteArrayOutputStream` class and the analysis introduced in Section 2. For the purpose of the analysis, we are interested in the `count` field. In this case, the relevant state of the class consists only of the `count` field. Since it is expensive to model check classes with a large state space, we are interested in dealing with a relevant state that is as small as possible.

A *model class* of a class C is a class with the same fully qualified name as C . C may be for instance a class from the standard library. The two classes, the model and the original one, have the same public interface, i.e., they have the same public, protected and package method signatures. However the implementation of these methods may differ. Ideally, to enable reductions during model checking, the model class would have fewer fields and behaviors compared to the original class.

A statement is said to *affect a variable* (a field or a variable local to a method) if it has a read or write dependency on it. This includes assigning a new value to the variable (write dependency) or calling a method on the object referenced by the variable (read dependency), since side effects of the method may modify it. A statement affects a variable of mutable type (i.e., not a primitive type or a `String`) also if the variable is used as a parameter in a method call (read

dependency), since the called method may modify it. Note that, in the context of this paper, saying that a statement affects a variable does not imply that it will modify it but means that a statement may possibly modify it. For instance we can call an instance method that does not modify the object on which is called. In the algorithms that will be defined in the next section, such statements are considered to be affecting the variable that references the object, even if they do not actually modify it.

The analysis presented in this paper is thus a *may* analysis, which safely overapproximates the results and may consider some statements as relevant, even though at run-time the statements may not be. In addition because the analysis is intra-procedural, it considers methods invocations as potentially relevant. This is a sound approach but can be made more precise by adding the inter-procedural analysis later.

4. ALGORITHMS

In this section we describe two algorithms that are used to generate model classes. The first one, *slicing based*, tries to remove fields and statements that do not affect relevant fields. However, due to various dependencies, this algorithm may retain statements that have dependencies on irrelevant fields, thus forcing the tool to generate models containing irrelevant fields as well. The second algorithm tries to overcome the limitations of the previous one by combining a slicing approach with value generators: dependencies on irrelevant fields are replaced with dependencies on local variables, that can be generated in a variety of ways.

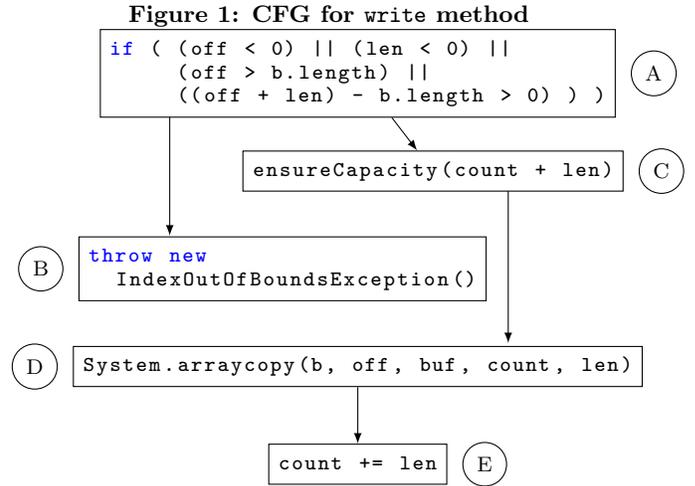
Both approaches generate models that are tied to properties being checked. If the set of relevant fields changes when a different property is verified, the model class should be generated from scratch. However, with our tool generating model classes from scratch is a quick process. Moreover, for specific libraries (like Swing/AWT) the models can be generated based on a class of properties. For instance, we can generate models based on GUI components' logical properties excluding their look-and-feel. These models can be reusable across a wide array of property checks.

4.1 Slicing Based Approach

Program slicing [6] is a technique that automatically decomposes a given program, based on the specification of a slicing criterion, into parts that affect the slicing criterion and the parts that do not. The result of the slicing operation is a program containing the statements that affect the slicing criterion.

In our context, the slicing criterion is a set of user-defined fields deemed relevant based on domain-specific information or properties being checked. If a statement affects (directly or indirectly) some part of the relevant state then it has to be kept, otherwise it can be sliced away. In this way we can create model classes that exhibit a behaviour restricted to relevant state and similar to that of the original class.

Slicing is performed by analyzing the control flow of a method and the data dependencies between its statements [3]. To compute method slices we employ *data flow analysis* [3]. In this kind of analysis the control flow graph (CFG) of a method is used to propagate abstract information along ex-



ecution paths. In particular, we use *backward* flow analysis, which considers all the edges of the CFG reversed.

The algorithm proceeds as follows. Flows are initialized as empty sets at the exit points of the method and are propagated backwards along the CFG edges. If a node v of the CFG affects some part of the relevant state or one of the variables in its incoming flow, then it is marked as relevant. Moreover, the variables on which it has read dependencies are added to its outgoing flow. A fixed point computation is used to calculate a transitive closure of all information flows.

Based on analysis results, all the statements that are not marked as relevant can be removed, since they do not affect the relevant state. In fact the statements that affect the relevant state are all marked, since they (a) affect directly some part of the relevant state or (b) affect variables that are used in other statements affecting the relevant state in the execution path. Moreover, if after this pruning there are empty branching statements, they are removed.

The drawback of this approach is that it can include in the model statements involving irrelevant state. For instance, suppose a statement, which is marked as relevant, has a data dependency on an irrelevant field f . Hence f will be added to the outgoing flow of the statement, with the result of having all the predecessors in the CFG that affect f marked as relevant. In order to get a model class that compiles, these irrelevant but necessary fields have to be added to the model class.

4.1.1 Example

Consider the example introduced in Section 2 and in particular the method `write`, shown in Listing 2. Its CFG is shown in Figure 1.

The algorithm applied to this method proceeds as follows. The incoming flow of nodes B and E is empty, since they are the entry points of the backward flow analysis. Both nodes are marked as relevant: node E because it affects `count`, which is part of the relevant state, node B because it is a throw statement. Since statement E has a read dependency on `len`, its outgoing flow includes it. The incoming flow of

node D consists of the `len` local variable. Both `len` and `count` are integers, they are passed by value to the method `arraycopy`. This means that the method call cannot modify them. Since the statement does not affect the relevant state nor any variable in its incoming flow, it is not marked as relevant. Its outgoing flow is the union of all incoming ones. Hence it is made up by the sole `len` variable. Statement C is a method call on the object instance itself. Since it may modify it, the statement is marked as relevant. Finally, node A is marked as relevant because nodes C and B are control dependent on it.

After the data flow analysis, we can remove the statements that are not marked as relevant. The modeled method is then

```
public synchronized void write(
    byte b[], int off, int len) {
    if ( (off < 0) || (len < 0) ||
        (off > b.length) ||
        ((off + len) - b.length > 0) ) {
        throw new IndexOutOfBoundsException();
    }
    ensureCapacity(count + len);
    count += len;
}
```

Notice that this model method has no dependencies on irrelevant fields. Conversely, consider the method `toString` from Listing 2. In this case the value returned by the method depends on the irrelevant field `buf`. In order to have a compilable model class, we have to include this field. Note however that this buffer may not be updated by other modeled methods, like `write`.

4.2 Combined Approach

This approach to model generation mixes the previous one with value generators, in order to overcome the limitations of the slicing based approach. The goal is to keep the behaviour associated with relevant fields while removing dependencies on irrelevant fields.

The algorithm proceeds as follows. First, each method of the original class is sliced. The slicing criterion is the same for the slicing based approach: a statement is kept only if it affects, directly or indirectly, some part of the relevant state. When a statement is found relevant, we check if it contains references to an irrelevant field. In this case, we replace the reference to the irrelevant field with one to a local variable. The value assigned to the local variable can be either a default value or a value generated randomly at runtime. The type of this variable is the same as the replaced one. Thus, dependencies on irrelevant fields are removed.

4.2.1 Example

Consider again the example from Section 2. The algorithm proceeds in the same way as the slicing based approach for method `write`, producing the same result. The difference between the two algorithms lies in how the dependency on `buf` in method `toString` is handled. The slicing based algorithm was forced to introduce an unwanted dependency on field `buf`. The combined approach can either a) insert a dependency on a local default value (that can be configured arbitrarily)

```
public synchronized String toString() {
    byte[] defaultBuf = new byte[]{0, 0, 0, 0};
    return new String(defaultBuf, 0, count);
}
```

or b) insert a statement that generates random values at runtime.

```
public synchronized String toString() {
    byte[] rndBuf = Generator.genByteArray();
    return new String(rndBuf, 0, count);
}
```

The string returned by this method is unrelated to what was put in the stream. However, the return value is not relevant to the property being checked for the example in Section 2.

4.3 Limitations

The approaches described in this section have some limitations. Not all the information at our disposal is used. For instance, the information that can be gathered by points-to and side-effects analyses is ignored. Moreover the slicing analysis is limited to a single method at a time, i.e., it is an instance of intra-procedural analysis. None of the algorithms deals with chains of field accesses. For instance, if `rel` is a relevant field, the statement `rel.f1.f2` is not considered relevant.

4.4 Implementation

The algorithms we described are implemented in a tool called `modgen`¹. The tool is built on top of Soot². It is composed of several modules, divided in two families: model generation modules and output modules. The former modules implement the algorithms described in this paper, sharing a common interface. The output modules allow the user to choose between several output formats: decompiled Java source, Java bytecode, and Jimple, an intermediate bytecode representation. The modules are glued together by a Groovy configuration infrastructure, that provides ease of configuration through a dedicated Domain Specific Language.

5. CASE STUDY

In this section we analyze the behaviour of the JPF model checker using models generated by the various algorithms. The application under test is the one shown in Listing 1. We verify that the value of variable `count` (line 18) is always greater or equal than 0. To perform this check we use `NumericValueListener`. We run JPF with the original library class and then with the models generated by the two algorithms described in this paper.

Original class. JPF correctly detects that the value taken by `count` never goes below zero. This is what we expect, since it represents the number of elements pushed in the stream during the execution of the application. The drawback of using the original class is that it includes behaviour irrelevant for the analysis. JPF executes all the statements, including the ones that affect only the field `buf`. These statements do not influence the outcome of the analysis. Hence we would like to skip them.

¹<https://bitbucket.org/cecca/modgen>

²<http://www.sable.mcgill.ca/soot/>

Slicing based approach model. Running the slicing based algorithm produces a model whose code includes all the statements affecting `count` but not all the ones affecting `buf`. For instance the statements inserting elements in the `buf` array are excluded from the model. However the ones that use the array to build the string representation of the stream are included.

This leads to a problem when we run JPF with this model. The execution of the system under test proceeds until line 20, where there is a call to the `toString` method. The model of this method tries to build a string from the `buf` array contents. However the array was never updated, so we get an `ArrayIndexOutOfBoundsException` exception, since it is a zero-length array.

Thus the attempt to verify the property fails with this model, since JPF stops, printing the exception. Note that if the system under test did not include lines 20-23 or if side-effects analysis was used to stub out the `toString` method, everything would have worked correctly. This shows the main flaw in the models generated by this approach: as long as an application depends only on the fields that are part of the relevant state, everything works fine. As soon as the application relies on the value of a field that is not part of the relevant state, the fact that that field was not updated in the model is likely to make the verification process fail.

Combined approach model. The model generated by the combined algorithm has some desirable features: it contains all the statements affecting the `count` variable and none of the ones affecting `buf`. When a statement depends on `buf`, this dependence is replaced with one with a default value. Take for instance the `toString` method. Instead of depending on the `buf` field, it depends on a byte array filled with zeroes. This means that the string returned by the method is meaningless. However for the analysis performed, that involves only the value assigned to `count`, the string returned by `toString` is irrelevant.

Running JPF with this model produces the correct result for the verification of the property: the `count` variable never goes below zero. The advantage of running this model class instead of the original one was that JPF does not execute the statements that modify the `buf` array.

The tool has been applied to some classes of the Swing framework, namely `javax.awt.JTextField`, `javax.awt.JToggleButton` and `javax.awt.JTabbedPane`. These classes were missing support in the `jpf-awt` extension. Using our tool we were able to generate models for these classes using the various approaches described in this paper. An analysis of the performance and effectiveness of such models is on our future work list.

6. CONCLUSIONS AND FUTURE WORK

In this paper we proposed two algorithms to automatically generate model classes for model checkers. The first algorithm generates models using slicing. This yields model classes that implement the behaviour associated with all the relevant fields. However the algorithm may introduce dependencies on irrelevant fields that can lead to problems during

the actual usage of the model class.

The second algorithm is a combination of slicing and value generation. The generated model classes include the behaviour associated with relevant fields. Dependencies on irrelevant fields are replaced with dependencies on local fields with a default value or values generated at random at runtime.

One direction of future work is to combine slicing with other kinds of analysis, like side-effect analysis. This may produce more accurate models since more information about the code is used. For example, methods that have no side-effects on relevant fields can be safely modeled with empty stubs. Another improvement can come from the usage of inter-procedural analysis. This kind of analysis will make it possible to distinguish between method calls that really affect the relevant fields.

Moreover, the tool can be adapted to generate model classes “on-demand”: instead of generating the models statically before the verification process, the model generator could run as a JPF extension, intercepting accesses to missing model classes and generating them on the fly.

Furthermore, more case studies are needed to evaluate the performance of the algorithms. Finally, we plan to use the algorithms to extend existing model classes used in JPF extensions like `jpf-awt`.

Acknowledgment

The authors would like to thank Franck van Breugel for his comments, and the reviewers for their invaluable feedback. Development of this project has been funded by Google as part of the Google Summer of Code 2013 program.

7. REFERENCES

- [1] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto. Cache-based model checking of networked applications: From linear to branching time. In *ASE'09*, page 447–458, 2009.
- [2] M. Ceccarello and N. Shafiei. Tools to generate and check consistency of model classes for Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.
- [3] U. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis: Theory and Practice*. CRC Press, Mar. 2009.
- [4] P. Mehrlitz, O. Tkachuk, and M. Ujma. Jpf-awt: Model checking gui applications. In *ASE'11*, page 584–587, 2011.
- [5] N. Shafiei and F. van Breugel. *Automatic on-demand delegation of calls in Java PathFinder*. 2013.
- [6] F. Tip. A survey of program slicing techniques. *Journal Of Programming Languages*, 3:121–189, 1995.
- [7] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *FSE*, Sept. 2003.
- [8] M. Ujma and N. Shafiei. jpf-concurrent: An extension of java PathFinder for java. util. concurrent. *arXiv preprint 1205.0042*, 2012.
- [9] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.