Abstraction Engineering

Michael Achenbach, University of Aarhus, ma@cs.au.dk

Abstract

Model checking and testing are widely used techniques for software verification and falsification respectively. Both techniques, however, suffer from the state space explosion problem and from practical problems with IO and missing program parts. In model checking, abstractions have been proposed to tackle these problems, but they are not commonly applied in practice yet. In testing, mock objects are already state of the art for simulating missing program parts. However, little research has been done on how to engineer abstractions and mock objects to be applicable in a wide range of use cases. We generalize abstractions and mock objects, since they show many similarities on an engineering level. We analyze the issues imposed by the application of common abstractions and mock objects on software design and engineering. Our project also comprises a case study on current program transformation tools like AspectJ or Javassist and discusses implementations of typical abstractions for Java PathFinder. We compare the tools to extract requirements for a new abstraction tool that has little coupling to its back-end technology.

Problem & Motivation

In model checking, program abstractions are used to tackle the state space explosion problem, while in testing, mock objects are used to enable integration tests. Unlike abstractions in model checking, mock objects in testing are already state of the art and are widely used in practice. Recent developments show, however, that explicit state model checkers like Java PathFinder (JPF) generalize traditional testing methods in the falsification setting [JPF05]. Java PathFinder has its own built-in virtual machine and is able to execute analyzed programs with all language features of Java. The application of different test cases can be simulated with Java PathFinder by analyzing multiple execution paths modeled by non-deterministic choice. The analysis process benefits from backtracking and state matching. Backtracking enables resetting the state of the analyzed program to the next non-deterministic choice point. This saves computation time compared to testing, since there, each test case executes from the beginning. State matching prevents the repetition of already analyzed states. Moreover, analyzing program executions with an explicit state model checker includes different thread interleavings of multi-threaded applications. We generalize therefore testing as a possible configuration of explicit state model checking. In our generalized setting our goal is to analyze the engineering requirements of abstractions for model checking on the one hand and of mock objects in testing on the other.

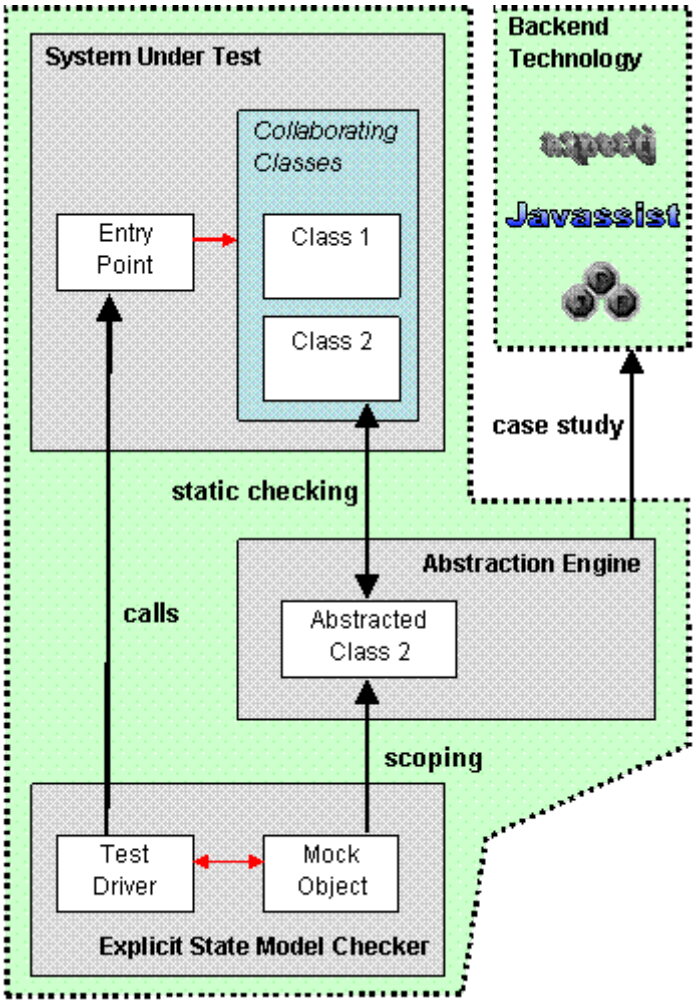
```
FileReader fileReader = null;
try {
  fileReader = new FileReader(inputFile);
  BufferedReader in = new
    BufferedReader(fileReader);
  try {
    while (in.readLine() != null)
      ...;
  } finally { in.close(); }
} catch (IOException e) {}
finally {
  try { fileReader.close(); } // (*)
  catch (IOException e) {}
}
...
```

The code example above describes a typical implementation in Java that reads the contents of a file. Our first test goal is to verify the robustness of this code regarding exceptions, i.e., that no unchecked exception can be thrown to the surrounding scope. The second goal is to check if all possible executions obey the open/close protocol of streams, i.e., if in every possible execution the stream is closed and if there is never a read in the *closed* state. A systematic test of the code should reveal a possible null

pointer exception at the line marked with (*), which can occur if `FileReader` causes a `FileNotFoundException` during construction. Since it is difficult to put the system file reader into all its possible failure states, an abstraction should simulate this behavior and also integrate an FSM modeling the open/close protocol.

```
class MyFileReader abstracts FileReader {
  boolean opened = true;
  public AFileReader (Reader in) {
    if(?)
      throw new FileNotFoundException();
  }
  public void close() {
    if(?) throw new IOException();
      opened = false;
  }
  ...
}
```

For the example above, it would be convenient to exchange the Java `FileReader` class with an abstracted version that models the test criteria above. The second example shows a desired implementation of such an abstraction class. The question mark models non-deterministic choice of the underlying model checker. The new keyword `abstracts` enables static checking of the compatibility of the abstraction class with its concrete counterpart.



In our work, we formalize our notion of abstractions for explicit state model checking and testing and analyze implications of these abstractions to software development. We discuss the requirements of an abstraction tool and evaluate them in our case study on current program transformation tools. We analyze which abstraction settings can be implemented using these tools without undesired design modifications of the analyzed client code.

The picture on the left shows our experimental setup. The system under test (SUT) is the client code to test containing several collaborating classes. The explicit state model checker serves as test driver and delivers execution traces and counter examples. The depicted abstraction engine serves as template for an engineering method introducing abstractions at runtime. Static checking and scoping are requirements on such a method. Static checking of an abstraction class guarantees the compatibility of the class with the abstracted concrete class in the analysis. Scoping gives a degree of control, when to use which abstraction at runtime. The backend technology comprises current program transformation and class loading methods like AspectJ [KHH+01], Javassist [Chi00], or the Java Model Interface (JMI) of JPF [JPF05]. In our case study we analyze to which extend these existing technologies can serve as an abstraction engine. In our further research we discuss how a new abstraction engineering tool can be developed on top of these methods.

Background & Related Work

Both model checking and testing suffer from the state space explosion problem. Due to program variables with a huge domain like integer or string and reference variables, which are only practically bounded by

memory size, the state space of a program is in theory infinite. In the last decade, much research has been done to facilitate explorative methods like model checking. The analyzed program is translated into a finite state automaton containing acceptance and error states. The automaton can then be exhaustively explored. It can be checked if every computation ends in an acceptance state and if the error state is never reached. The transformation is performed by using two different kinds of abstractions:

Over-approximation: The analyzed program is approximated, so that each property that holds in the approximation is guaranteed to hold in the concrete program. This approach is sound in the verification setting, since programs can be proven correct. Due to approximation, however, reported errors can possibly be false positives. Examples are various applications of abstract interpretation [CC77] and predicate abstraction [VPP00].

Under-approximation: Only a subset of the feasible paths of the analyzed program is explored. Abstractions can be used to prune certain behavior and to guide the exploration to suspicious program parts. This approach is sound in the falsification setting; reported bugs are guaranteed to originate from feasible behavior. Due to approximation, however, not all errors are guaranteed to be found. Examples are testing in general and systematic test input creation methods [GKS05].

Recently, also the combination of both approximations in one search algorithm gave promising results [GHK+06].

Testing

In program testing, the state space explosion problem is tackled by selecting only a small number of program inputs for exploration. In practice, besides infinite state space, other problems arise both in model checking and testing. System calls of the analyzed programs, like IO calls, can not be easily simulated, since certain error sources, like a lost network connection or a defect hard drive, are practically not available. Either system calls are cashed and made reversible, or they have to be removed by abstraction. Some resources like a database connection might not be available or might not be implemented for explorative methods that support backtracking of the program state.

In Test Driven Development, tests are specified first and program parts are incrementally implemented to meet the test criteria [Bec02]. To test as early as possible in the development process, mock objects for integration testing are required to simulated missing program parts [FMPW04]. Other examples are mock objects that serve to test EJB applications independently of a remote server, or mock objects that simulate a database connection to enable backtracking.

Bandera

The Bandera toolkit is an abstraction and slicing tool for software model checking that enables the extraction of finite state automata of Java programs [CDH+00]. It is a flexible framework that provides interfaces to several model checkers including Java PathFinder. It focuses on the verification of programs and allows sound abstractions of primitive values like the *signs* abstraction for integers. Bandera builds on top of the Soot framework for bytecode manipulation and program transformation. Unfortunately, the Bandera toolkit does not cope with all Java language features in its latest build. The project is currently in hibernation.

Uniqueness of the Approach

Program transformation tools like AspectJ [KHH+01] are already used in testing to introduce mock objects into the system under test without heavy design changes [Les02]. Unlike these approaches, our research focuses on the general engineering of abstractions in testing and model checking. We unify the notion of

abstraction in model checking and the concept of mock objects in testing and analyze the general engineering requirements of both. We don't focus on a specific tool like AspectJ, but analyze a broader range of program transformation methods and discuss extensions of these tools to meet the requirements of a generalized abstraction engine.

Unlike the Bandera toolkit [CDH+00], our approach focuses on the abstraction of classes in an object oriented language. With Bandera, only the abstraction of primitive types and primitive class members is possible. There are several approaches in literature that focus on the abstraction of a specific class of reference types, like linked lists or trees. However, all these approaches focus on implications for soundness and completeness and not on the engineering requirements of actually using the abstractions in practice. To the best of our knowledge, our research proposes the first approach that focuses on the engineering of class abstractions generalizing model checking and testing.

Results & Contributions

We made a case study of several engineering methods that facilitate the use of abstractions in model checking and testing. We analyzed the usage of AspectJ [KHH+01], Javassist [Chi00], and the Java Model Interface (JMI) of Java PathFinder [JPF05] and compared the methods. For the comparison we defined criteria that influence important conceptual and technical aspects of engineering and design:

Static checking denotes if static type safety of the abstraction regarding the program to analyze is guaranteed. The absence of static checking could for instance lead to incompatible abstractions or missing methods at runtime.

Scoping defines the level of control that is given to specify when to use which abstraction or replacement. A fine-grained scoping would for instance allow to specify different abstractions of the same type in different scopes at runtime.

Non-monotonicity denotes in how far concrete client data can be removed or their construction can be omitted in an analysis. A monotone approach can only add data to client code.

Abstraction dependencies arise if abstractions of different types are connected. E.g., a sound abstraction of integer values in the verification setting, like the signs abstraction *{neg,0,pos}*, leads to uncertainty in the expression *pos<pos*. This uncertainty can be modeled either with non-deterministic choice of *{true,false}* or with a 3-valued abstraction of boolean. If abstraction dependencies are not allowed, the interface of < can not be changed to make use of another abstraction.

Invasiveness states if a modification of the client code is necessary.

Counter example mapping is the ability of a model checker to map a counter example to valid source code for further analysis. The criterion defines if the abstraction engineering supports such a mapping.

Java library support is a technical criterion that focuses on the Java programming language. It denotes if an abstraction technique can manipulate library and core classes of Java.

Without the use of advanced program transformation techniques, the application of abstractions often requires heavy design changes. Either abstraction or simulation code is directly implemented in the class containing the behavior to abstract, or the client code uses unique interfaces for concrete and abstraction classes. The introduction of the corresponding object at runtime, however, might require the exposure of

local variables and other design changes that are not desirable [Les02]. In the following, we call this approach the hand-written approach and compare it with the other methods.

Comparison

We compare the advantages and disadvantages of the analyzed methods using the criteria above. The table on the right gives an overview of our results, which we explain in the following. The abbreviation def. stands for dependent, h.w. for hand-written solution; the fields marked with (*) are explained below.

	AspectJ	Javassist	JMI	h.w.
static checking	yes	no	no	yes
scoping	yes	(*)	no	dep.
non-monotonicity	no	yes	yes	dep.
abs. dependencies	yes	yes	no	no
invasiveness	no	no	no	yes
c.ex. mapping	no	no	no	yes
library support	yes	(*)	yes	no

AspectJ enables static checking of the replacement behavior, such that the abstraction code is guaranteed to execute on runtime. There is a fine-grained level of scoping using `cflow` and `withincode` pointcuts. This scoping is encapsulated in aspects, so that different test cases can be associated with different abstractions. Abstraction dependencies are possible since additional data fields introduced with inter type declarations are visible within an aspect. We see the major disadvantage of AspectJ in its monotonicity. It is not possible to remove certain data fields and to circumvent their construction. The intuition of directly exchanging a class with an abstraction is not supported by AspectJ. Another disadvantage is that counter example mapping does not work for aspects or for classes manipulated with the weaver.

Javassist outperforms AspectJ in its ability to exchange classes. The intuitive notion of replacing a class with an abstraction class can easily be implemented. Dependencies between the abstractions are possible since references to a dependent class can be exchanged at load-time. Unlike in AspectJ, replacement classes are not guaranteed to be compatible. We marked the scoping criterion with a (*), since the usage of different abstractions at runtime is not impossible, but conceptually very difficult to implement. The exchanged class could serve as a root in an inheritance tree that introduces abstraction variants in subclasses. Even though also Javassist facilitates scoping constructs like `cflow`, the instrumentation code is not encapsulated in an aspect and has to be performed separately for each loaded class. How to manipulate library classes with Javassist is currently unclear, since the class loading mechanism that enables these manipulations does not work together with JPF.

The **Java Model Interface** of JPF is comparable to a class loading mechanism, but unlike in Javassist, dependencies between different abstractions are not possible, since abstraction classes are forced to have the same interfaces as the concrete classes to replace. There is no level of scoping that would facilitate the usage of more than one abstraction at runtime. Like Javassist, JMI laks static abstraction checking. Even though the method comes along with JPF, counter examples do not map back to the exchanged but to the original code.

The characteristics of the **hand-written solution** are, as expected, opposed to the other methods in our experiment. How far this approach facilitates scoping and non-monotonicity, depends highly on invasiveness. To enable an analysis setup with abstractions, a modification of the client code is necessary. Detailed scoping leads to undesired design changes like the exposure of local variables or the introduction of common interfaces. A manipulation of library code is not possible. However, opposed to the other methods, the hand-written solution still preserves a valid counter example mapping.

Conclusion

Our work on abstraction engineering generalizes abstractions in model checking and mock objects in testing. In our cases study we analyzed current program transformation methods focusing on the

requirements of an abstraction tool. In our future work we plan to propose extensions of these methods to develop a new abstraction tool for model checking and testing on top of them.

References

[Bec02] Kent Beck. Test Driven Development: By Example. Addison-Wesley Professional, November 2002.

[BNRS08] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In Proceedings of the 2008 international symposium on Software testing and analysis, pages 3–14, Seattle,WA, USA, 2008. ACM.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, Los Angeles, California, 1977. ACM.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Psreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In Proceedings of the 22nd international conference on Software engineering, pages 439–448, Limerick, Ireland, 2000. ACM.

[Chi00] Shigeru Chiba. Load-Time structural reflection in java. In Proceedings of the 14th European Conference on Object-Oriented Programming, pages 313–336. Springer-Verlag, 2000.

[FMPW04] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, not objects. In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 246, 236. ACM Press, 2004.

[GHK+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pages 117–127, Portland, Oregon, USA, 2006. ACM.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. SIGPLAN Not., 40(6):213–223, 2005.

[JPF05] Java PathFinder. http://javapathfinder.sourceforge.net/, 2005.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming, pages 327– 353. Springer-Verlag, 2001.

[Les02] Nicholas Lesiecki. Test flexibly with AspectJ and mock objects. http://www.ibm.com/developerworks/java/library/jaspectj2/, 2002.

[VPP00] Willem Visser, Seungjoon Park, and John Penix. Using predicate abstraction to reduce Object-Oriented programs for model checking. IN PROCEEDINGS OF THE 3RD ACM SIGSOFT WORKSHOP ON FORMAL METHODS IN SOFTWARE PRACTICE, pages 3—12, 2000.