2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation

Engineering Abstractions in Model Checking and Testing

Michael Achenbach and Klaus Ostermann
Department of Computer Science
University of Aarhus
Aarhus, Danmark
{ma,ko}@cs.au.dk

***Abstract*— Abstractions are used in model checking to tackle problems like state space explosion or modeling of IO. The application of these abstractions in real software development processes, however, lacks engineering support. This is one reason why model checking is not widely used in practice yet and testing is still state of the art in falsification. We show how user-defined abstractions can be integrated into a Java PathFinder setting with tools like AspectJ or Javassist and discuss implications of remaining weaknesses of these tools. We believe that a principled engineering approach to designing and implementing abstractions will improve the applicability of model checking in practice.**

I. Introduction

All methods of program analysis have to deal with the curse of Rice's theorem, by which all interesting properties of a program are not decidable. Therefore, abstractions which approximate program entities are necessary to say anything useful about the program. Abstract interpretation is probably the best-known theory of approximation [1], since it gives a precise mathematical framework to design abstractions that are sound with respect to the operational meaning of the program. Typically, one can distinguish two different kinds of abstractions: Over-approximating abstractions, which are sound (but incomplete) from the verification perspective, and under-approximating abstractions, which are sound (but incomplete) from the falsification perspective. Types in static type checking and predicate abstraction [2] are examples of over-approximating abstractions, whereas systematic test input creation methods [3], [4] and test cases in general are examples of under-approximating abstractions.

In this work, we are interested in user-defined abstractions of program entities. For example, an abstraction of a list may only store the size of the list but not the actual list elements. An abstraction of a file stream for checking exception handling may throw exceptions whenever the file is used instead of accessing any physical file. An abstraction of an integer may only record whether the integer is zero or not. We are *not* concerned with the soundness of such abstractions from the perspective of verification or falsification — this is of course an important and challenging problem, but it is not in the focus of this paper. Rather, our focus is on the engineering challenge: How can such abstractions be programmed and integrated into a software system?

We will analyze this problem in the context of Java PathFinder (JPF) [5], an explicit state model checker for Java. Moreover, we apply JPF in the falsification setting, i.e., for error detection by exploration. Software model checking has received a lot of attention in the last decade [5], [6]. Recent developments show, however, that due to the state space explosion problem and other obstacles, model checking is not applicable to large programs without the use of abstractions [5]. We believe that one obstacle to the practical application of such abstractions is the absence of a principled engineering approach for their design and implementation.

In practice, program testing is still the state of the art of falsification. It is applied in all phases of software development and used in unit, integration, and system testing. In integration testing, mock objects — which are, in our terminology, user-defined abstractions — are used to abstract non-implemented or missing program parts [7]. In software model checking, predefined abstractions for primitive types (such as abstracting integers to their sign) are well-known. Recently, also abstractions for some specific reference types like linked lists or trees have been proposed [8], [9]. Due to engineering difficulties, however, generalized user-defined abstractions for complex types (such as classes) have received little attention.

The hypothesis of this work is that user-defined abstractions are both useful and feasible for model checking and testing. The contributions of our paper are as follows:

- We make the case for applying user-defined abstractions in testing and model-checking.
- We identify the engineering requirements for integrating such abstractions into a software project.
- We analyze different modularization technologies, namely AspectJ, Javassist, and JPF's Java Model Interface, with regard to their applicability to integrate abstractions. We also identify a list of limitations of these approaches.
- We discuss potential extensions to these tools that will make them more applicable to abstraction engineering.

The remainder of this paper is organized as follows: In Section II, we motivate user-defined abstractions for model checking and testing and present examples that will be

978-0-7695-3793-1/09 $26.00 © 2009 IEEE
DOI 10.1109/SCAM.2009.25

IEEE computer society

used throughout the paper. Section III identifies the requirements of abstraction engineering, exemplifies how current engineering methods perform, and compares the strengths and weaknesses of these methods. Section IV discusses concepts that address the limitations of the analyzed methods and presents research challenges that focus on remaining weaknesses. In Section V, we give an overview of related approaches, Section VI concludes.

II. The Need for Abstractions

The use of abstractions to deal with the state space explosion problem in model checking is well-known in literature. There are other issues which motivate the usage of abstractions of program entities, such as the usage of I/O resources like networks or databases, which may not be available during the analysis. In Test Driven Development, another concept of abstraction is used to enable integration testing [10]. So called mock objects are used to simulate missing program parts at runtime [7], [11]. Typically, mock classes are alternative implementations of some concrete classes, either handwritten or generated. They are introduced into the system under test for simulating behavior, monitoring, or checking of specifications.

While mock objects often are practical ad-hoc solutions, their expressiveness is limited. Using an explicit state model checker as test driver would enrich modeling possibilities, e.g., the behavior of a mock object could be modeled with non-deterministic choice. The term abstraction used throughout the paper therefore addresses abstractions in a broader sense, generalizing both over- and under-approximations as well as generated or handwritten mock objects. Furthermore, our notion of abstraction includes also specifications defined in source entities, e.g., user-defined method preconditions.

We focus on the abstraction of classes, which in Java excludes primitive types, such as integer and boolean. We discuss the abstraction of primitive types in Section V.

In the following, we show two examples where user-defined abstractions are useful for analyzing runtime behavior. Later we discuss different engineering methods to implement and integrate such abstractions.

A. Example: IO Abstractions

The example shown in Figure 1 describes a typical Java implementation for reading the contents of a file. When an exception is catched during execution, some logging data is written to a separate log file using the method `logException`. Our first goal is to verify the robustness of this code regarding exceptions, i.e., that no unchecked exception can be thrown to the surrounding scope. Our second goal is to check if all executions obey the IO stream protocol. As shown in Figure 2, a valid execution never reaches the *error* state and ends in the acceptance state *closed*. A systematic test of the code in Figure 1 should reveal a possible null pointer exception at the line

```
class ClientCode {
  String readFile(File file) {
    String output = "";
    String ln = null;
    FileReader r = null;
    try {
      r = new FileReader(file);
      BufferedReader in =
        new BufferedReader(r);
      try {
        while((ln=in.readLine()) != null) {
          output += ln + "\n";
        }
      } finally {
        in.close();
      }
    } catch (IOException e) {
      logException(e);
    }
    finally {
      try {
        r.close();                  // (*)
      } catch (IOException e) {
        logException(e);
      }
    }
    return output;
  }

  void logException(Exception e){
    File logFile = new File("error.log");
    ...
  }
}
```

Figure 1. Sample Java class that reads the contents of a file and logs exception data to another file

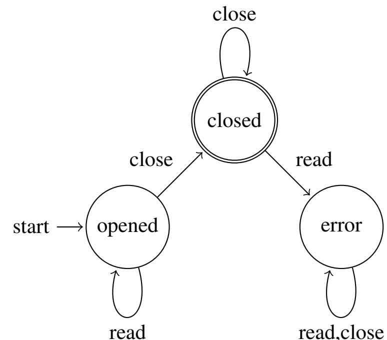

Figure 2. Automaton modelling the open/ close behavior of IO streams

```
HashSet union(HashSet set1, HashSet set2) {
  HashSet result = new HashSet();
  result.addAll(set1);
  result.addAll(set2);
  return result;
}

HashSet set1 = new HashSet();
HashSet set2 = new HashSet();
set1.add(1);
set2.add(2);

assert union(set1, set2).contains(1);
assert union(set1, set2).contains(2);
```

Figure 3. Toy example with set union implementation and some test cases

marked with (*), which can occur if `FileReader` causes a `FileNotFoundException` during construction. Because files can be arbitrarily large and are located on hard drive, we can test only some suspicious executions systematically. These include executions where the system file reader enters a failure state. It is, however, difficult to put the system file reader into all its possible failure states. Therefore, a replacement of the class `FileReader` should simulate this behavior and integrate the FSM from Figure 2. We assume that the code used in `logException` works correct. The challenge here is to test `readFile` independently, i.e., replacements for `File` or `FileReader` used in the context of `readFile` should not influence the concrete behavior of `logException`. We show different implementations of such replacements in Section III and discuss the challenge of using different versions of class `File` at runtime in Section IV.

B. Example: Finite Set Abstraction

The toy example in Figure 3 shows a set union implementation with sets in Java. The `union` method creates a new result set for each call locally. The example also contains a selection of functional test cases. A broader verification of behaviors and, hence, a reduction of the state space can be achieved by using a finite abstraction for sets containing $\{InSet, NotInSet\}$ regarding one specific item [12]. The partial functional specification can then be defined as:

$$union(InSet, NotInSet) = InSet$$

$$union(NotInSet, InSet) = InSet$$

$$union(NotInSet, NotInSet) = NotInSet$$

Such an abstraction could be implemented as a replacement of the concrete `HashSet` class.

III. Engineering Abstractions

Every phase in software development like design, implementation, or testing has corresponding engineering methods and tools. Currently, there exists no such method that focuses on the engineering of abstractions for software model checking. In this section, we analyze the requirements of such a method and make the case, to which degree current engineering methods fulfill these. For illustration, we use the IO example from Section II-A, since this example focuses on different important aspects like abstractions of different classes, different abstractions of one class, and modeling of system classes. In the following subsections, we discuss how to use AspectJ [13], Javassist [14], and the Java Model Interface (JMI) of Java PathFinder [15] for abstraction engineering, and thereafter we compare the methods. In the following, we define requirements and criteria for abstraction engineering that address important conceptual and technical aspects of engineering and design:

Expressiveness denotes to which degree concrete code can be refined or replaced by abstractions. We distinguish **behavioral replacement**, where only the internal behavior of methods can be modified, **structural refinement**, where a class can be refined with additional data, and **structural replacement**, where data can also be removed from a class or where a complete class can be exchanged.

Scoping defines the level of control that is given to specify when to use which abstraction or replacement. We distinguish between **global scoping**, where only one replacement can be used at runtime, and **local scoping**, where time and place of different usages can be controlled in more detail. In the example from Section II-A, local scoping would allow the execution of the client code with an abstraction of `File` to test the `readFile` method, but in the same time use the original `File` in the scope of the `logException` method. We discuss different variants of local scoping in Section IV.

Abstraction dependencies arise if abstractions of different classes access each other using a bigger interface than visible to the client code. In our example from Section II-A, e.g., a dependency would arise, if the abstraction of `File` would implement a new method which should be accessible within the abstraction of `FileReader`. For type safety reasons, such a dependency must not be visible to the class `ClientCode`. If the used engineering method does not support abstraction dependencies, the structural interfaces of `File` and `FileReader` in the abstraction must equal the concrete structural interfaces, i.e., all method signatures must be identical.

Static checking is a process that verifies the consistency between abstraction and program to analyze. The ab-

sence of static checking could for instance lead to incompatible abstractions or missing methods at runtime. In the example from Section II-A, an abstraction of `FileReader` needs to provide the methods `close` and `read` for the class `ClientCode` to be executable. An abstraction of `File`, however, is only accessed by the abstraction of `FileReader`. Hence, it only needs to provide a structural interface visible in the abstraction of `FileReader`.

Non-invasiveness states if a modification of the client code can be avoided. E.g., in the IO example it is not desirable to modify the code of the class `ClientCode` and, e.g., expose the local variable `r` to enable the introduction of different `FileReader` implementations.

Traceability denotes if the engineering method provides a mapping from executed code to source code, i.e., if a bytecode error trace can be mapped to a corresponding trace of source artifacts. This increases understandability of counter examples and enables further analyses of the source code as well as visualization of the error trace using source entities in an IDE.

System library support is a technical criterion that focuses on the Java programming language. It denotes if an abstraction technique can manipulate runtime 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. In the following, we call this approach the hand-written approach and compare it after an introduction to Java PathFinder with AspectJ, Javassist, and JMI.

A. Java PathFinder

Java PathFinder (JPF) is implemented as a virtual machine for Java and is able to execute analyzed programs with all language features of Java [5]. JPF has an API for modeling non-deterministic choice and execution pruning in the source code of the analyzed program. For example, rather than assigning a specific integer to a variable, one can specify a range of integers (say, 1 to 10), and all possible execution paths resulting from these non-deterministic choices are then systematically executed. Pruning of executions can be modeled with a special condition in the source code. Dependent on its evaluation at runtime, the execution path containing this condition is pruned. The main techniques in JPF for this purpose are backtracking and state matching. Backtracking enables resetting the state of the analyzed program to the next non-deterministic choice point. State matching prevents the repetition of already analyzed states. JPF can in fact be

```
class OpenCloseFSM extends FSM {

  OpenCloseFSM() {
    // Add transitions of the stream FSM
    ...
  }

  // Proceeds one transition with the given
  // name from the current state to the
  // next state. E.g. action "close"
  // proceeds from "opened" to "closed"
  void proceed(String action) {
    super.proceed(action);

    // Assert that the "error" state can
    // never be reached.
    assert !currentState.equals("error");
  }
}
```

Figure 4. Implementation of the stream FSM from Figure 2

seen as a generalization of traditional testing methods in the falsification setting [15]. Executing test cases with JPF has several benefits: Backtracking saves computation time compared to testing, since in testing each run executes from the beginning. State matching automatically rules out redundant test cases. Moreover, analyzing program executions with an explicit state model checker includes different thread interleavings of multi-threaded applications.

B. AspectJ

Our implementation of the IO example of Section II-A with AspectJ [13] explores three different variants of introducing abstractions into client code:

- adding state with inheritance,
- using inter-type declarations,
- mapping additional state to existing objects.

We assume some familiarity with AspectJ pointcut and advice — for further reading we refer to [16]. In our examples, we implement the stream FSM from Figure 2 with the class `OpenCloseFSM` as shown in Figure 4. The class `FSM` (not shown here) maintains the state of the FSM in the field `currentState` and stores the available states and transitions. These states and transitions are initialized in the `OpenCloseFSM` constructor. The method `proceed` of `FSM` performs one state transition on `currentState`. The implementation of `proceed` in `OpenCloseFSM` keeps track if the error state can be reached, e.g., if `read` is applied in the `closed` state.

Inheritance: The example in Figure 5 shows the introduction of state with inheritance. The around advice intercepts constructor calls of `FileReader` using the `call` pointcut and creates an instance of the new subclass `AbstractFileReader` instead. We model the possible non-existence of the file to read using the non-deterministic

```
class AbstractFileReader
    extends FileReader {

  FSM fsm;

  // Dummy constructor
  public AbstractFileReader(File in)
      throws FileNotFoundException {
    super(in);
  }

  public void close() throws IOException {

    // Models the occurrence of an
    // exception during closing
    if (Verify.getBoolean())
      throw new IOException();

    // Proceeds to the next state
    // of the stream FSM
    fsm.proceed("close");
  }

  // Similar for other FileReader methods
  ...
}

// Circumvents the "FileReader" constructor
FileReader around()
    throws FileNotFoundException :
    call(FileReader.new(..)) {
  AbstractFileReader result = null;
  try {
    result = AbstractFileReader
      .class.newInstance();
  } catch (Exception e) { ... }

  result.fsm = new OpenCloseFSM();

  // Models the existence of the file
  if (Verify.getBoolean())
    throw new FileNotFoundException();

  return result;
}
```

Figure 5. Abstraction engineering with AspectJ — adding state with inheritance

choice command of JPF, `Verify.getBoolean()`. This leads to different execution paths of the analysis, some paths with a `FileNotFoundException` thrown, others without. The new `close` method models the possibility of an `IOException` and implements the state transition `close` of the stream FSM. The construction of `AbstractFileReader`, however, suffers from the mandatory call to a super constructor, which should be circumvented in the first place. This is a technical and not a conceptual problem of the underlying virtual machine that executes the code. The Java Virtual Machine does not allow

```
aspect Abstraction {

  // Models the existence of the file to
  // read in a new field of class "File"
  boolean File.exists
    = Verify.getBoolean();

  FSM FileReader.fsm = new OpenCloseFSM();

  // Constructor modeling file existence
  FileReader.new(boolean exists)
      throws FileNotFoundException {
    if (!exists)
      throw new FileNotFoundException();
  }

  // Circumvents the original constructor
  // with the new constructor above
  FileReader around(File file)
      throws FileNotFoundException :
      call(FileReader.new(File))
      && args(file) {
    assert file != null;
    return new FileReader(file.exists);
  }

  // Circumvents the "close" method
  void around(FileReader reader)
      throws IOException :
      execution(* FileReader.close())
      && target(reader) {

    // Models the possible occurrence of
    // an exception during closing
    if (Verify.getBoolean())
      throw new IOException();

    // Proceeds to the next state
    // of the stream FSM
    reader.fsm.proceed("close");
  }

  // Around advice demonstrating local
  // scoping: the method "exists" is
  // only circumvented if the control
  // flow is not in "logException"
  boolean around(File file) :
      !cflow(call(* *.logException(..)))
      && call(* File.exists())
      && target(file) {
    return file.exists;
  }

  // Similarly for other "File"
  // and "FileReader" methods
  ...
}
```

Figure 6. Abstraction engineering with AspectJ — state introduced with inter-type declarations

the instantiation of `FileReader` here, since no null-ary constructor exists. The reflection implementation of JPF, however, allows such a call and uses a default initialization for all fields without calling one of the original constructors. The *inheritance* approach is yet restricted to Java classes that are not defined `final`. A dependency between different abstractions is not possible without using inter-type declarations. Therefore, the existence or non-existence of the file to read must be modeled directly in the around advice of the `FileReader` constructor and can not be maintained in an abstraction of class `File`, like done in our next example.

Inter-type declarations: The example in Figure 6 shows the aspect code of our implementation using inter-type declarations and around advice. The definition `FSM FileReader.fsm` inserts a new field `fsm` of type `FSM` into the `FileReader` class. The existence of the file to read is modeled with a new field `File.exists`. The abstractions depend on each other, since the `exists` field is read in the constructor around advice. The field `exists` is initialized using non-deterministic choice like described above. Calls to the other methods of `FileReader` are circumvented with around advice. Local scoping is needed in this example, since different instances of class `File` are referenced at runtime. The first instance (the formal parameter of `readFile` in Figure 1) should be part of the abstraction, while for the local variable in `logException`, the concrete behavior of `File` should be executed. We implement such a local scoping using the `cflow` pointcut to control the advice of the method `exists`. Within the control flow of `logException`, the normal behavior of the method `exists` will be executed, while in all other cases the around advice will take place. Note that inter-type declarations in Java library classes are not supported using the default configuration. The AspectJ compiler must be reconfigured to build a local copy of the java runtime library with the options "-inpath rt.jar -outjar modified_rt.jar".

State mapping: Mapping additional state to existing objects is conceptually not much different to the above version. Therefore, we do not show detailed code examples of this variant. Like in the former example, the behavior of the `FileReader` class is circumvented with around advice. State like the stream FSM is kept in a mapping from `FileReader` to state. Unfortunately, such a mapping requires a reference value of `FileReader`, i.e., one of the original constructors of `FileReader` has to be called. This is an obstacle to circumventing the undesired system call, similar to the object construction problem in the *inheritance* approach.

The usage of AspectJ could be eased by a design change that makes use of an auxiliary method for creating the object to abstract:

```
Reader r = createFileReader();
```

```
// Abstraction of class "File"
class MyFile {
  // Models the existence of the file
  boolean exists = Verify.getBoolean();

   ...
}

// Abstraction of class "FileReader"
class MyFileReader {
  FSM fsm;

  // The constructor references the type of
  // the abstraction class "MyFile" of
  // "File" in order to access the "exists"
  // field. This constructor replaces the
  // original "FileReader(File f)"
  MyFileReader(MyFile f)
      throws FileNotFoundException {
    assert f != null;

    // Models the existence of the file
    // dependent on the state of "f"
    if (!f.exists)
      throw new FileNotFoundException();

    fsm = new OpenCloseFSM();
  }

  void close() throws IOException {
    // Models the possible occurrence of
    // an exception during closing
    if (Verify.getBoolean())
      throw new IOException();

    // Proceeds to the next state
    // of the stream FSM
    fsm.proceed("close");
  }

   ...
}
```

Figure 7. Abstraction engineering with Javassist — abstraction of multiple classes with dependencies among the replacement classes

The client code has to be modified in this case, so that both the concrete class and the abstraction class implement the same interface. This would solve the object construction problem of the *inheritance* and *state mapping* approaches. It would also ease scoping, since only one around advice is necessary to instrument this auxiliary method.

C. Javassist

Javassist is a bytecode manipulation tool that can serve as program transformer or as custom class loader [14]. Unlike in AspectJ, a static checking of the manipulated constructs is not performed, i.e., it is possible to modify a class at load time in a way, such that the class gets incompatible at runtime. However, class members can be replaced, data

```
class MyTranslator implements Translator {
  public void onLoad(ClassPool p, String n)
      throws ... {
    if (n.equals("ClientCode")) {
      CtClass client = p.get("ClientCode");
      client.replaceClassName(
        "java/io/File",
        "MyFile");
      client.replaceClassName(
        "java/io/FileReader",
        "MyFileReader");
    }
  }
}
```

Figure 8. Abstraction engineering with Javassist — translator for substitution

```
class MyTranslator implements Translator {
  public void onLoad(ClassPool p, String n)
      throws ... {
    if (n.equals("java.io.FileReader")) {
      CtClass a = pool.get("MyFileReader");
      a.setName("java.io.FileReader");
      a.replaceClassName("MyFile",
        "java/io/File");
    }

    if (n.equals("java.io.File")) {
      CtClass a = pool.get("MyFile");
      a.setName("java.io.File");
    }
  }
}
```

Figure 9. Abstraction engineering with Javassist — translator for loading a replacement

fields and methods can be removed, complete classes can be exchanged. Our notion of abstraction with Javassist is to exchange a concrete class with an abstraction implementation. This can be done using two different methods:

- substituting the references to a class in client code,
- loading a different implementation of a class.

Substitution: The application of substitution suits our IO example best. The implementation shown in Figure 7 introduces two new classes `MyFile` and `MyFileReader` in order to abstract `File` and `FileReader`. A dependency between the abstraction types is introduced with the field `exists` in class `MyFile`. The structural interface of class `MyFileReader` declares a parameter of type `MyFile` instead of `File` in order to access the `exists` field in the constructor. The class `MyTranslator` shown in Figure 8 contains an excerpt of the class loading configuration. It can be used in a Javassist class loader, since it implements the interface `Translator`, which is part of the Javassist API. On load-time, the class `File` is substituted by `MyFile` in the class `ClientCode` from Figure 1, `FileReader` similarly. This substitution is local to class `ClientCode`, but could be extended to other classes as well. A local scoping that uses `MyFile` in `readFile` and `File` in `logException` at the same time cannot be achieved that way.

Loading a replacement: The *replacement* approach uses a translator that configures the class loader to load a different class file as soon as the class file containing the concrete implementation is requested. This method is not suitable for our example from Section II-A since the rebinding of system classes is required. However, we give a sketch of the corresponding translator class, since this approach can be used for non-system classes in general. The replacement classes are the same as in Figure 7. The translator is shown in Figure 9. The abstraction classes `MyFileReader` and `MyFile` replace their concrete counterparts globally. For the client code to work, their structural interfaces have to be compatible with the ones of `FileReader` and `File`, i.e., the references to `MyFile` in `MyFileReader` have to be substituted with `File`. Otherwise, calling the `FileReader` constructor in class `ClientCode` from Figure 1 would cause a runtime exception, since the static call site contains a reference to `File`.

The current implementation of Java PathFinder does not fully support reflection and class loading functionality of Java. Therefore, the load-time mechanism of Javassist is not compatible with analyses using Java PathFinder at present, so that the compile-time mechanism has to be used. The GluonJ project is a new aspect-oriented programming system that extends Javassist [17]. It includes a pointcut language that provides local scoping. However, GluonJ does only allow the refinement of existing classes and does not support the exchange of classes.

D. Java Model Interface

The Java Model Interface (JMI) of Java PathFinder is the intended mechanism of JPF for modeling Java core and library classes [15]. By configuring the class loader of JPF, it can be specified, which implementations should be used at runtime. In our analysis we implemented the IO example with a class that has the same interface as `FileReader` and that models the test goals shown in Section II-A. After configuring a Java PathFinder run with this class, we could successfully reveal the errors discussed in Section II-A.

Similar to Javassist, a static type safety of the modeling class is not guaranteed. Dependencies between abstractions that require a change of method signatures are not supported, since JPF does not provide an API for bytecode manipulation during class loading and we assume that the bytecode is compiled by a type-safe Java compiler, which would reject classes that have a different structural interface compared to their concrete counterparts.

	AspectJ	Javassist	JMI	h.w.
behav. replacement	✓	✓	✓	n/a
struct. refinement	✓	✓	✓	n/a
struct. replacement	no	✓	✓	n/a
global scoping		✗	✗	n/a
local scoping	✗			n/a
abs. dependencies	✓	✓	no	n/a
static checking	✓	no	no	✓
non-invasiveness	✓	✓	✓	no
traceability	no	no	no	✓
system libraries	✓	(*)	✓	no

Figure 10. Comparison of engineering methods

E. Comparison

We compare the advantages and disadvantages of the methods above using the requirements defined in Section III. Figure 10 provides an overview of our results, which we explain in the following. The abbreviation h.w. stands for hand-written solution; the field marked with (*) is explained later.

AspectJ enables static checking of the replacement behavior, so that all code is guaranteed to execute at runtime. AspectJ provides local scoping with `cflow` and `withincode` pointcuts. Pointcuts are encapsulated in aspects, which eases the application of different test cases with different abstractions, e.g., by compiling each test case separately with its corresponding abstraction aspect, or by refining the scoping in subaspects per test case. Abstraction dependencies are supported using inter-type declarations. We argue that the major disadvantage of AspectJ is its limited expressiveness. The object construction problem described in Section III-B is a conceptual problem of AspectJ that prevents the removal of data fields and the circumvention of their construction. The desired direct exchange of a class with an abstraction is not supported. Another disadvantage is that a source code mapping does not exist for aspects or for classes manipulated by the weaver.

Javassist outperforms AspectJ in expressiveness with support for exchanging classes. Structural replacement using an abstraction class with the same structural interface can easily be implemented. Dependencies between the abstractions are possible with the replacement strategies shown in Section III-C. Unlike in AspectJ, replacement classes are not guaranteed to be type-safe. The scoping of Javassist is rather global, since the usage of different abstractions at runtime is not impossible, but conceptually 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 aspects and has to be performed separately for each loaded class. How to manipulate library classes with Javassist at load-time is currently unclear, since the class loading mechanism that enables these manipulations does not work in our experimental setup due to the missing reflection support of JPF explained in Section III-C.

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

The characteristics of the hand-written solution are as expected opposed to the other methods in our experiment. A discussion of the requirements expressiveness, scoping and abstraction dependencies is irrelevant, since these requirements highly depend on the invasiveness of the approach. Enabling an analysis setup with abstractions, leads to design changes like the introduction of common interfaces. Local scoping leads to source modifications like the exposure of local variables. A manipulation of library code is not possible, but opposed to the other methods, the hand-written solution preserves traceability.

F. Automation

All presented approaches require substantial manual work. Several of the methods have room for a semi-automatic solution. In the following we give a sketch of automation possibilities, however, a detailed analysis of such approaches is future work. A skeleton of an abstraction class could be generated from the structural interface of a concrete class. Then, a programmer could chose between default configurations and user-defined configurations. Method defaults could be, e.g., exploring all possible exceptions or returning a subset of possible return-values using non-deterministic choice. The configuration could also be acquired from documentations that make use of the Java Modeling Language [18]. E.g., executable specifications maintained in so called model variables could be used to generate an abstraction class that explores these specifications.

IV. Discussion

None of the engineering methods analyzed in Section III is really satisfactory from the perspective of abstraction engineering. In this section we would like to discuss the main problems to which more engineering support is required. The issues discussed here can hence be seen as a challenge problem to the tool designers.

None of the approaches we discussed is well-suited to replace families of collaborating classes (such as `File` and `FileReader`) by an abstraction thereof. Ideally,

we would like to write classes `AbstractFile` and `AbstractFileReader` which are statically verified to have the same structural interface as their concrete counterparts. This would require some form of structural subtyping or a form of nominal subtyping where a subtype of a class can be created without inheriting its implementation, such as in the work by Ostermann et al [19], [20].

Furthermore, it would be desirable if the abstractions can have both wider and smaller interfaces than their concrete counterparts. `AbstractFile` could have methods not present in `File`, but accessing these additional methods induces a covariance problem which calls for a notion of class families that can be refined simultaneously [21]. It also frequently occurs that some of the methods are "private" within the class family, i.e., only called within the classes that are abstracted, and then it should not be necessary to provide these methods in the abstracted classes.

Approaches such as classboxes [22] or higher-order hierarchies [21] are not a solution to this problem, since they require that the simultaneous refinement of a family of classes must be anticipated in the design of the application.

These problems become even worse if one considers the possibility of sophisticated local scoping strategies, where multiple versions of the same class can co-exist in the same running program. If the interface of the abstracted classes is not exactly identical to the concrete classes, then typing problems arise if different versions of the same class are mixed. It would be desirable to either check that different families are never mixed (requiring a novel form of family polymorphism [23]), or to have exact control over what happens when different families meet, e.g., implicit conversions to the most specific common abstraction. This would require the abstractions to be organized in a lattice, where the "join" of two abstractions always exists.

Scoping itself can also take multiple forms: Lexical scoping, temporal scoping, thread-local scoping, scoping determined by the heap structure, etc. In principle, AspectJ provides a powerful scoping construct with its rich pointcut language, but pointcuts are only applicable to advice, but not to structural changes of classes. More research is needed on how to reconcile sophisticated scoping with the possibility to make structural changes to classes and types.

V. Related Work

In the following, we give an overview of current applications of AspectJ in testing [24], [25], [26] and describe the contributions of the Bandera toolkit to abstraction engineering [27].

AspectJ in Testing: AspectJ [13] has several applications in testing [24], [25], [26]. Laddad suggests the use of AspectJ to enable testing of private data with privileged aspects, monitor runtime behavior during testing, or use aspects for error reporting [26]. Lesiecki and Jeffries present several examples on the use of AspectJ to introduce mock objects into the system under test saving invasive code modifications and heavy design changes of client code [24], [25]. Unlike these approaches, our research focuses on abstraction engineering in general for 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. Using an explicit state model checker as test driver extends also the expressiveness of abstractions and specifications used in testing, e.g., by modeling with non-deterministic choice. We do not restrict our research to one tool, but analyze several program transformation methods and show their weaknesses regarding abstraction engineering.

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 [27]. It provides interfaces to several model checkers including Java PathFinder. Bandera focuses on verification and allows sound abstractions of primitive values like the abstraction of signs for integers [12]. In its latest release, it does not cope with all Java language features. Unlike the Bandera toolkit, our approach focuses on abstractions of classes in an object oriented language. With Bandera, only the abstraction of primitive types and primitive class members is possible. Moreover, Bandera builds on top of the Soot framework [28], which offers a complex API for bytecode manipulation. We believe, however, that the maintainability of an abstraction engineering tool relies highly on the complexity of the underlying back-end, for which tools like AspectJ or Javassist are better qualified. There are several approaches in literature that focus on the abstraction of a specific class of reference types, like linked lists or trees [8], [9]. However, these approaches focus on implications of the concepts and not on the usability in practice.

VI. Conclusion

We illustrated in our work how useful and feasible user-defined abstractions can be integrated into software systems in order to facilitate the application of model checking and to extend the modeling possibilities of common testing. We identified problems with engineering and design as the main obstacles for the application of such abstractions. Our analysis of current engineering methods showed to which degree these methods fulfill the requirements of abstraction engineering. Our comparison showed, however, that neither of the approaches is really satisfactory. We discussed how to solve the present weaknesses and addressed the tool developers with new challenges on extensions of the current tools. In our future work, these extensions will enable the design of a principled abstraction engineering approach, which will significantly increase the applicability of model checking in practice.

REFERENCES

[1] P. Cousot and R. 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*. Los Angeles, California: ACM, 1977, pp. 238–252.

[2] W. Visser, S. Park, and J. 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*, pp. 3—12, 2000.

[3] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. Lisbon, Portugal: ACM, 2005, pp. 263–272.

[5] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, 2000, pp. 3–11.

[6] P. Godefroid, "Model checking for programming languages using VeriSoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Paris, France: ACM, 1997, pp. 174–186.

[7] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM Press, 2004, pp. 246, 236.

[8] S. Khurshid, C. Pasareanu, and W. Visser, *Generalized symbolic execution for model checking and testing*, 2003.

[9] S. Anand, C. S. Psreanu, and W. Visser, "Symbolic execution with abstraction," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 53–67, 2009.

[10] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, Nov. 2002.

[11] T. Mackinnon, S. Freeman, and P. Craig, *Endo-testing: unit testing with mock objects*. Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 287–301.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Psreanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from java source code," in *Proceedings of the 22nd international conference on Software engineering*. Limerick, Ireland: ACM, 2000, pp. 439–448.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer-Verlag, 2001, pp. 327–353.

[14] S. Chiba, "Load-Time structural reflection in java," in *Proceedings of the 14th European Conference on Object-Oriented Programming*. Springer-Verlag, 2000, pp. 313–336.

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

[16] "Aspectj documentation," http://www.eclipse.org/aspectj/doc/next/, 2009.

[17] "Gluonj," http://www.csg.is.titech.ac.jp/projects/gluonj/, 2005.

[18] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards, "Model variables: cleanly supporting abstraction in design by contract: Research articles," *Softw. Pract. Exper.*, vol. 35, no. 6, pp. 583–599, 2005.

[19] K. Ostermann, "Nominal and structural subtyping in component-based programming," *Journal of Object Technology*, vol. 7, no. 1, pp. 121 – 145, 2008.

[20] K. Ostermann and M. Mezini, "Object-oriented composition untangled," in *Proceedings of ACM 16th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '01), Tampa, Sigplan Notices, Vol. 36, No. 10*, 2001.

[21] E. Ernst, "Higher-order hierarchies," in *Proceedings ECOOP 2003*, ser. LNCS 2743, L. Cardelli, Ed. Heidelberg, Germany: Springer-Verlag, Jul. 2003, pp. 303–329.

[22] A. Bergel, S. Ducasse, and O. Nierstrasz, "Classbox/J: controlling the scope of change in java," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. San Diego, CA, USA: ACM, 2005, pp. 177–189.

[23] E. Ernst, "Family polymorphism," in *Proceedings ECOOP 2001*, ser. LNCS 2072, J. L. Knudsen, Ed. Heidelberg, Germany: Springer-Verlag, 2001, pp. 303–326.

[24] N. Lesiecki, "Test flexibly with AspectJ and mock objects," http://www.ibm.com/developerworks/java/library/j-aspectj2/, 2002.

[25] R. Jeffries, "Virtual mock objects using AspectJ with JUNIT," http://www.xprogramming.com/xpmag/virtualMockObjects.htm, 2002.

[26] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, Jul. 2003.

[27] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Psreanu, and H. Zheng, "Tool-supported program abstraction for finite-state verification," *IN PROCEEDINGS OF THE 23RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, pp. 177—187, 2001.

[28] R. Valle-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. Mississauga, Ontario, Canada: IBM Press, 1999, p. 13.