A Flexible Framework for the Estimation of Coverage Metrics in Explicit State Software Model Checking ⋆

Edwin Rodríguez, Matthew B. Dwyer, John Hatcliff, and Robby

Department of Computing and Information Sciences
Kansas State University ⋆⋆

Abstract. Explicit-State Model Checking is a well-studied technique for the verification of concurrent programs. Due to exponential costs associated with model checking, researchers often focus on applying model checking to software units rather than whole programs. Recently, we have introduced a framework that allows developers to specify and model check rich properties of Java software units using the Java Modeling Language (JML). An often overlooked problem in research on model checking software units is the problem of *environment generation:* how does one develop code for a test harness (representing the behaviors of contexts in which a unit may eventually be deployed) for the purpose of driving the unit being checked along relevant execution paths?
In this paper, we build on previous work in the testing community and we focus on the use of coverage information to assess the appropriateness of environments and to guide the design/modification of environments for model checking software units. A novel aspect of our work is the inclusion of *specification coverage* of JML specifications in addition to code coverage in an approach for assessing the quality of both environments and specifications. To study these ideas, we have built a framework called **MAnTA** on top of the Bogor Software Model Checking Framework that allows the integration of a variety of coverage analysis with the model checking process. We show how we have used this framework to add two different types of coverage analysis to our model checker (Bogor) and how it helped us find coverage holes in several examples. We make an initial effort to describe a methodology for using code and specification to aid development of appropriate environments and JML specifications for model checking Java units.

1 Introduction

Building concurrent object-oriented software to high levels of assurance is often very challenging due to the difficulty of reasoning about possible concurrent task interleavings and interference of interactions with shared data structures. Over the past several years, several projects have demonstrated how model checking technology can be applied to detect flaws in several types of software systems [1, 10, 12] including concurrent object-oriented systems [6, 3, 19]. Model checking is very useful for the analysis

⋆ This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, and by Rockwell-Collins.

⋆⋆ 234 Nichols Hall, Manhattan, KS 66506, USA.
{edwin,dwyer,hatcliff,robby}@cis.ksu.edu

of concurrent systems because it can explore all the program states represented by possible execution interleavings of the threads in a system, spotting very intricate errors. However, the exhaustive nature of model checking makes it difficult to scale to large systems. This lack of scalability is due mainly to the state-space explosion problem: as systems become more complex, the state space grows exponentially, making it very difficult to analyze large systems without having the analysis exhaust available memory.

For these reasons, many reseachers believe that it is most natural to apply model checking to software units, or modules, instead of whole systems. Specifically, software model checking is often envisioned as part of a development and quality assurance methodology in which it is incorporated with *unit testing*. When model checking a software unit, one typically desires to specify/check as much of the unit's behavior as possible in the hope of detecting as many bugs as possible. In the past, model checkers have only supported checking of temporal property specifications with simple state predicates and assertions. To enhance the ability of developers to specify meaningful properties of software units, we have recently extended Bogor[20], our software model checking framework, to support checking of rich specifications [19], written in JML (Java Modeling Language, [16]). Using JML, developers can specify class invariants and method pre/post-conditions that contain detailed constraints on variables and data structures using various forms of universal and existential quantification over heap-allocated data.

To apply model checking to software units (with or without JML specifications), a developer needs to follow an approach that is similar in many respects to the steps involved in traditional unit testing. In unit testing, one develops a *test harness* that makes method calls into the unit for specific sets of parameter values and examines the results of the method calls for invalid results (indicating failed test). When applying model checking to software modules, one must similarly use a test harness (also termed a *closing environment* or an *environment*) to drive the unit through particular execution paths. The scale and complexity of a software unit's interface may vary greatly: a unit may consist of multiple classes and interfaces that expose fields and methods through a variety of mechanisms, such as, reference, method call, inheritance and interface implementation. Consequently, a general environment for a unit must be designed to accommodate all legal modes of external interaction. The environment will sequence those interactions to represent the behavior of program contexts in which the unit will be used in a larger piece of software.

Since model checking aims to exhaustively explore program execution paths, it is important for the environment used in model checking units to generate, to as large an extent as possible, the execution paths through the unit that will occur when the unit is actually deployed as part of a larger software system. Constructing such environments for the purpose of model checking is surprisingly difficult. For example, [18] note that it took several months to construct an environment that correctly modeled the context of the DEOS real-time operating system scheduler. Furthermore, in recent work, Engler and Musuvathi [7] describe proper environment construction as one of the main impediments to applying model checking as opposed to other techniques including static analysis for bug-finding in source code. Our work on the Bandera Environment Generation tools [22, 21] provides basic support that addresses many of the challenges en-

countered in the DEOS case study, but it does not treat the complexities that arise when model checking units with JML specifications. In this setting one encounters additional questions: how complete is the test harness when it comes to stressing all the behaviors referred to by the specification?, and dually, how complete are the specifications with respect to all the behaviors that are exercised by the test harness upon the module?

Since an environment determines the behaviors of the system that are explored during model checking it directly influences the cost of checking and the ability to find bugs. In this paper, we seek to build on extensive work done in the testing community and emphasize various types of *coverage* as a means of (a) determining the suitability of an environment, and (b) guiding the construction and refinement of environments. A variety of strategies for developing environments based on coverage are possible, for example, generalizing a given environment to increase coverage or adding additional environments that provide complementary coverage. Previous work on model checking has used code coverage information to guide heuristic search strategies [9] and to determine the completeness of analysis of large software systems including TCP implementations [17]. In this paper, we view the collection and use of coverage information as an integral part of a model checker's implementation and application, focusing specifically on its interaction with the checking of strong specifications written in JML. In particular, we use coverage information to not only determine adequacy of the environment, but we also distinguish the notions of *code coverage* and *specification coverage* for the purpose of addressing several interesting questions about the relationships between the code of the unit, the specification of the unit, and the environment(s) used to check a unit.

- In addition to determining the code coverage for a given unit with respect to an environment, what are appropriate notions of coverage for a specification?
- For effective bug-finding with respect to a specification, are both high specification coverage and code coverage required?
- If there is a mismatch between code and specification coverage, what revisions of the environment, code or specification might this suggest?
- How is the cost of checking and the ability to find bugs sensitive to the number and generality of environments used to achieve a given level of coverage?

To experiment with these ideas, and to begin to answer some of these questions, we have built an extensible framework in Bogor called **MAnTA** (**M**odel-checking **An**alysis of **T**est-harness **A**dequacy) for implementing the collection of a variety of forms of coverage information. We describe how we have used MAnTA to implement two coverage estimation procedures and show how we have used them to find coverage holes in some of our models, triggering the refinement of the test harnesses for these systems. We summarize lessons learned and attempt to lay out a methodology that uses coverage information to help developers build environments that are appropriate for model checking units with strong specifications such as can be written in JML.

The rest of the paper is organized as follows. Section 3 discusses the general architecture of the MAnTA framework, its major components, and how it can be configured for effective estimation of coverage metrics in Bogor. In section 4 we show the results of experiments conducted using MAnTA to implement two different types of coverage estimation analyses and briefly discuss the benefits and implications of these results to

the application of coverage metrics in model checking. Section 5 presents some preceding work in the area of coverage metrics estimation for model checking as well as some other related works. Finally, in section 6 we conclude giving some conclusions drawn from the experiences in this research.

2 The Need for Coverage Metrics in Model Checking

Slogans associated with model checking proclaim that it represents "*exhaustive* verification" and that it "explores all possible states" have misled many users to believe that if a model checker completes with no specification violations reported then the system is free of errors. We believe that it is commonplace for errors in specifications and environment encodings to lead to "successful" checks that are essentially meaningless, since they explore only a tiny portion of the behavior of the system.

Consider Figure 1 which shows excerpts of a concurrent implementation of a linked list from [15]. To analyze this code we must construct an environment. In general one strives to develop environments that:

- are concurrent in order to expose errors related to unanticipated interleavings
- are non-deterministic in ordering method calls in order to expose errors related to unanticipated calling sequences

Unfortunately, the degree of concurrency and non-determinism in an environment are precisely the factors that lead to exponential explosion in the cost of model checking. For these reasons, users of model checkers typically develop restricted environments that have a limited number of threads where each thread implements only a specific pattern of method calls. We took this approach when developing the environment shown in Figure 2.

When we fed the linked queue code with the driver to the model checker, the model checker reported no specification violations. Upon examining MAnTA's output, we found that the code coverage was rather low. The problem lies in the method `run()` of the class `Process`. All processing done by each thread reduces to inserting an item into the list and then removing an item from the list. After some assessment, it is not so hard to see that by doing this, every thread is guaranteed that prior to extracting an element from the list, on any execution trace, the list is never empty.

The linked queue class implements a blocking discipline when a client tries to extract an object and the list is empty: it will block the client thread and make it wait until there is something (see method `take()`). However, as shown in Figure 3, all the code that implements the blocking behavior of the list is never executed by our given test harness (the figure shows all the code that was not covered by the test harness in a shaded area). This code was never exercised because the method is never called in a context where the list is empty. Thus, when using a model checker to analyze code units which are closed with the addition of an environment , a "no violations" result from the model checker cannot by itself give confidence that the unit being analyzed satisfies its specification.

Using the results of MAnTA, we were able to redesign the environment to increase the level of coverage significantly. In this case, we generalized the environment to allow for more general calling sequences in the thread.

```
public class LinkedQueue {
 final /*@ non_null @*/ Object putLock;
 /*@ non_null @*/ LinkedNode head;
 /*@ non_null @*/ LinkedNode last;
 int waitingForTake = 0; ...
 //@ invariant waitingForTake >= 0;
 //@ invariant \reach(head).has(last);
 /*@ behavior
   @   assignable head, last, putLock,
   @              waitingForTake;
   @   ensures \fresh(head, putLock) &&
   @           head.next == null; @*/
 public LinkedQueue() {
  putLock = new Object();
  last = head = new LinkedNode(null);
 }
 /*@ behavior
   @   ensures \result <==>
   @           head.next == null; @*/
 public boolean isEmpty() {
  synchronized (head)
   return head.next == null;
 }
 /*@ behavior
   @    requires n != null;
   @    assignable last, last.next; @*/
 void insert2(LinkedNode n) {
  last.next = n;
  last = n;
 }
 /*@ behavior
   @  requires x != null;
   @  ensures true;
   @ also behavior
   @  requires x == null;
   @  signals (Exception e) e instanceof
   @   IllegalArgumentException; @*/
 public void put(Object x) {
  if (x == null)
   throw new IllegalArgumentException();
  insert(x);
 }
 synchronized Object extract() {
  synchronized (head) return extract2();
 }
 /*@ behavior
   @  assignable head, head.next.value;
   @  ensures \result == null
   @    || (\exists LinkedNode n;
   @        \old(\reach(head)).has(n);
   @        n.value == \result
   @        && !(\reach(head).has(n))); @*/
 Object extract2() {
  Object x = null;
  LinkedNode first = head.next;
  if (first != null) {
   x = first.value;
   first.value = null;
   head = first;
  }
  return x;
 }
 /*@ behavior
   @  requires x != null;
   @  ensures last.value == x
   @          && \fresh(last); @*/
 void insert(Object x) {
  synchronized (putLock) {
   LinkedNode p = new LinkedNode(x);
   synchronized (last) insert2(p);
   if (waitingForTake > 0)
    putLock.notify();
locSpec2:
   return;
  } }
}
class LinkedNode {
 public Object value;
 public LinkedNode next; ...
 /*@ behavior ensures value == x &&
   @                  next == null; @*/
 public LinkedNode(Object x) {
   value = x;
 }
}
```

Fig. 1. A Concurrent Linked-list-based Queue Example (excerpts)

With a model checker alone, there is no tool support to indicate potential problems with the environment. We believe that this problem is especially important in our context where we are model checking JML specifications which typically specify more details about a program's state than specification languages traditionally used in model checking. Accordingly, we seek to develop forms of automated tool support not only for environment generation (as in our earlier work [21]), but also for providing feedback on the quality of environments in terms of both code coverage and *specification coverage*.

3 MAnTA Architecture

With MAnTA, we seek to provide a flexible framework that will allow different coverage metrics to be included as an integral part of the model checking process. Users

```
package linkedqueue;

public class LinkedQueueDriver {

 /*@ behavior
   @    ensures true;
   @*/
 public static void main(String[] args) {
  LinkedQueue lq = new LinkedQueue();
  new Process(lq).start();
  new Process(lq).start();
 }
}
```

```
class Process extends Thread {
 /*@ instance invariant lq != null; @*/
 final LinkedQueue lq;

 /*@ behavior
   @    requires lq != null;
   @    assignable this.lq;
   @    ensures this.lq == lq;
   @*/
 Process(LinkedQueue lq) {
  this.lq = lq;
 }

 /*@also
   @ behavior
   @    diverges true;
   @    ensures false;
   @*/
 public void run() {
  Object data = new Object();
  while (true) {
   lq.put(data);
   data = lq.take();
  }
 }
}
```

Fig. 2. A test harness for the concurrent linked queue program.

```
public Object take() {
  Object x = extract();
  if (x != null)
    return x;

  else {
    synchronized (putLock) {
      try {
        ++waitingForTake;
        for (;;) {
          x = extract();
          if (x != null) {
            --waitingForTake;
            return x;
          } else {
            putLock.wait();
          }
        }
      } catch (InterruptedException ex) {
        --waitingForTake;
        putLock.notify();
        // throw ex;
        throw new RuntimeException();
      }
    }
  }

}
```

Fig. 3. Portion of method `take()` not executed by test harness.

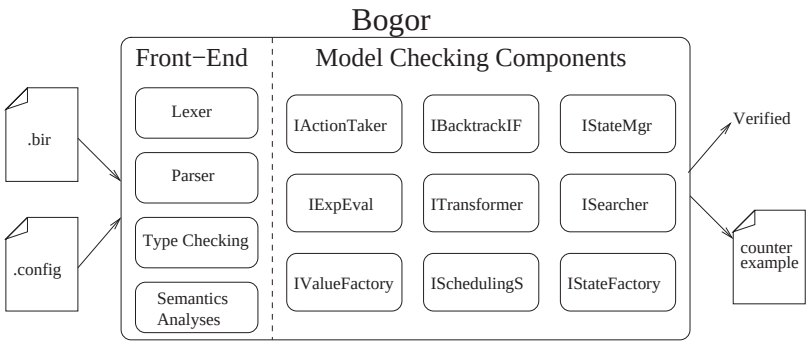

Fig. 4. Bogor Architecture.

can add an extra layer of confidence to the verification process simply by activating Bogor's default coverage analyses through MAnTA options. In addition, users can easily include their own implementation of coverage estimation algorithms to customize coverage information for specific reasoning goals or to experiment with new coverage algorithms. Indeed, one of our primary goals in developing the MAnTA framework is to support our continuing research on exploiting coverage metrics in model checking.

MAnTA is built on top of the Bogor software model checking framework. Figure 4 shows the architecture of Bogor, as presented in [19]. The framework is composed of loosely coupled modules, each one enclosing a different aspect of the model checking process. The *ISearcher* controls the exploration of the state space graph. The *IExpEvaluator* is the module that computes the value of side-effect free expressions by accessing the heap and all relevant local activation records. Each module can easily be extended to add extra functionality. For example, in the work by Dwyer et al. in [5], the *ISearcher* was extended to add partial order reductions to the model checker. Also, in [20], the *IActionTaker* was extended to allow the verification of methods' frame conditions.

Building on this flexibility, MAnTA accumulates coverage information by monitoring specific events in any of the Bogor modules. This is achieved with the use of an Observer pattern [8]: MAnTA modules can subscribe to relevant events of specific Bogor modules. For example, the module for structural coverage described in section 4.1 subscribes to the *ITransformer* module and observes each transition that is executed, using this information to perform the analysis. Similarly, the specification analysis module in section 4.2 observes the commands executed by the *IActionTaker* looking for the execution of assertions.

Figure 5 shows the simple architecture of the MAnTA framework. The *IMantaManager* module takes care of initialization and finalization of all the modules that implement specific coverage estimation analyses. The *IMantaCoverageAnalyzer* interface provides the point of implementation for different coverage estimation algorithms. To incorporate a particular coverage analysis into MAnTA, one simply specifies the implementing classes and the manager loads them during the initialization period.

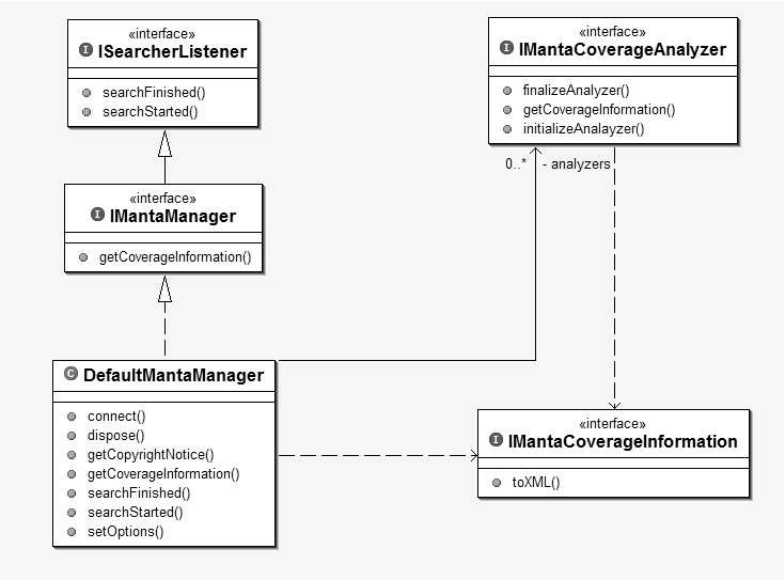

Fig. 5. MAnTA Architecture.

3.1 An Example

We illustrate the configuration aspects of the framework with the linked queue example from Figure 1. Suppose we wanted to model check this program and implement two types of coverage analysis – the structural and specification analyses that we will describe in detail in Sections 4.1 and 4.2. First we specify options in the Bogor configuration file that indicate the implementation of the analysis manager (we will choose the default manager) and indicate the number of analysis modules that will be executed.

```
IMantaManager=DefaultMantaManager
IMantaManager.analyzers.size=2
```

Now, we have to specify the implementation classes that MAnTA will use for each analysis. Notice that MAnTA does not need to know about the specifics of every analysis. All it needs to know is the name of the classes that implement the analyses to be able to load them:

```
IMantaManager.analyzers.0=MantaEdgeCoverageAnalyzer
IMantaManager.analyzers.1=MantaSimpleBooleanCoverageAnalyzer
```

Finally we specify the options for each of the analyzers. Structural coverage does not require an option. The specification coverage analyzer takes a few options from the configuration file that indicate particular locations to be monitored. For example, suppose we want to monitor the post-condition of the method `insert(Object)` in the linked queue program. We specify the following options:

```
MantaSimpleBooleanCoverageAnalyzer.analyzeAll=false
MantaSimpleBooleanCoverageAnalyzer.monitoredLocations.size=1
MantaSimpleBooleanCoverageAnalyzer.monitoredLocations.0=insert(),locSpec2
```

The first option tells the analyzer that we are interested in specific locations and not in all the specification formulas. The second and third options specify the size of the set of locations that we want to monitor and the method name and location label for the program point that we want to monitor. To be able to refer to a specific location we simply label the program point with a Java label as shown in Figure 1. However, it is not mandatory to refer to specific locations in order to monitor them. A common analysis will involve monitoring all the pre and post-conditions (and invariants). To do so, the three lines shown previously turn into:

```
MantaSimpleBooleanCoverageAnalyzer.analyzeAll=true
```

which tells the analyzer to monitor all pre-conditions, post-conditions and invariants checking.

At the end of the model checking run, the coverage manager invokes a finalization method on each module and collects the results. The coverage analyses results are reported from the analyzers to the manager in the form of an *IMantaCoverageInformation* object. This interface provides an uniform way to present the results, regardless of the type of analysis.

4 Case Studies

We have applied MAnTA to analyze coverage information during model checking of a collection of six Java programs with JML specifications. Model checking Java programs with Bogor is achieved by compiling Java byte codes into BIR (Bandera Intermediate Representation) [19]; BIR is a guarded-assignment language with primitives for defining all of the object-oriented features of Java. There is a direct correspondence between Java source lines, byte codes and sequences of BIR transitions. This facilitates the mapping of analysis results, including coverage information, calculated on the BIR model back to the input Java program. Figure 6 shows the BIR translation of the method `put(Object)` in the linked queue example. JML specifications are encoded as embedded assertions in the BIR model. Assertions are inserted during the translation process and correspond to JML pre and post-conditions, and invariants checking; only partial support for assertion generation from JML is implemented in the current toolset. For more details, we refer the reader to [20].

MAnTA, as a Bogor extension, operates on BIR models. In this section we describe our experiences with MAnTA while implementing two coverage analyses. The first example shows the implementation of structural coverage based on branch coverage. We report some interesting findings concerning some of our existing models and issues with their test harnesses. The second example is the implementation of a boolean satisfaction analysis. We report results for the same set of systems as the structural analysis case.

4.1 Structural Coverage Estimation with MAnTA

The first type of coverage analysis that we implemented on top of MAnTA was branch coverage. The classical definition of branch coverage is ensuring that each branch of a decision point (for example, an if statement) is executed at least once. For the BIR

```
function {|LinkedQueue.put(java.lang.Object)|}
 ((|LinkedQueue|) [|r0|],
  (|java.lang.Object|) [|r1|])
{
 (|java.lang.IllegalArgumentException|) [|$r2|];
 (|java.lang.Object|) [|$r3|];
 boolean spec1;
 boolean spec2;
 loc locSpec0: live {}
  do invisible
  {
  }
  goto locSpec1;
 loc locSpec1: live { spec1, spec2 }
  do
  {
   assert([|r0|]./|LinkedQueue.head|\ != null);
   assert([|r0|]./|LinkedQueue.last|\ != null);
   assert([|r0|]./|LinkedQueue.putLock|\ != null);
   assert([|r0|]./|LinkedQueue.waitingForTake|\ >= 0);
   assert(Set.contains<(|java.lang.Object|)>
   (State.reachSet<(|java.lang.Object|)>
    ([|r0|]./|LinkedQueue.head|\),
    [|r0|]./|LinkedQueue.last|\));
   spec1 := [|r1|] != null;
   spec2 := [|r1|] == null;
   assert(spec1 || spec2);
  }
  goto loc2;
 loc loc2: live { [|r1|], [|r0|], spec1, spec2 }
  when [|r1|] != null do
  {
  }
  goto loc6;
  when !([|r1|] != null) do
  {
  }
  goto loc3;
 loc loc3: live { [|$r2|], spec1, spec2 }
  do invisible
  {
   [|$r2|] := new (|java.lang.IllegalArgumentException|);
  }
  goto loc4;
 loc loc4: live { [|$r2|], spec1, spec2 }
  invoke
   {|java.lang.IllegalArgumentException.<init>()|}
    ([|$r2|]) goto loc5;
 loc loc5: live { spec1, spec2 }
  do
  {
   throw [|$r2|];
  }
  goto loc5;
 loc loc6: live { spec1, spec2 }
  invoke virtual
   +|LinkedQueue.insert(java.lang.Object)|+
    ([|r0|], [|r1|]) goto locSpec2b;
 loc locSpec2b: live { spec1, spec2 }
  do invisible
  {
  }
  goto locSpec2;
 loc locSpec2: live { spec1, spec2 }
  do
  {
   assert([|r0|]./|LinkedQueue.head|\ != null);
   assert([|r0|]./|LinkedQueue.last|\ != null);
   assert([|r0|]./|LinkedQueue.putLock|\ != null);
   assert([|r0|]./|LinkedQueue.waitingForTake|\ >= 0);
   assert(Set.contains<(|java.lang.Object|)>
    (State.reachSet<(|java.lang.Object|)>
     ([|r0|]./|LinkedQueue.head|\),
     [|r0|]./|LinkedQueue.last|\));
  }
  goto loc7;
 loc loc7: live {}
  do
  {
  }
  return;
 loc locSpec3: live {[|$r3|]}
  do invisible
  {
   assert ([|$r3|] instanceof
    (|java.lang.IllegalArgumentException|) && (spec2 && !spec1));
   throw [|$r3|];
  }
  goto locSpec3;
 catch (|java.lang.Object|) [|$r3|] at
  loc2, loc3, loc4, loc5, loc6, loc7 goto locSpec3;
}
```

Fig. 6. A Concurrent Linked-list-based Queue Example (excerpts)

program model this is achieved by determining the proportion of transitions that are exercised during model checking. The information returned by the analysis is the percentage of transitions executed and information about the actual transitions that were not executed (location within the model, etc.).

The implementation of this analysis was very straightforward. As with each MAnTA analysis, the module implementing the analysis must implement the coverage analyzer interface of Figure 5. During the initialization period, the module constructs a set `NotExecuted`, containing references to all the possible transitions in the system. This module subscribe to the *ITransformer*; every time a transition is executed, the tranformer notifies the analyzer, and the analyzer removes the transition reference from the set `NotExecuted`.

We tested this coverage analysis implementation in all the models used in our work for [20]. All these programs are annotated with JML [16], which is one of the specification languages that we used for verifying properties in Bogor. Table 1 summarizes the results we obtained for each program. Most of the model checks exhibited high structural coverage. The reason none of the numbers are over 90% is because there is some dead code in each model that is generated by our compiler. We estimate that those

Model	Total Number of Transitions	Number of Unexplored Transitions	Coverage Percentage
LinkedQueue	231	103	55.41%
BoundedBuffer	163	31	81.21%
RWVSN	238	34	85.71%
DiningPhilosophers	210	28	86.67%
ReplicatedWorkers	585	195	66.67%

Table 1. Percentage of structural coverage for all the systems analyzed.

models with coverage around 85% have around 95% coverage. They would not have 100% or close to it in part because our BIR models also model exceptional code, which is not exercised by our environments. It is possible, of course, to trigger exceptional code by defining appropriate environments. Two examples achieved significantly lower levels of coverage: *LinkedQueue*, from Figure 1, and *ReplicatedWorkers*.

When we model checked the *LinkedQueue* example with the structural coverage enabled we found a couple of surprises. First, we found an error in the model – specifically, in a fragment of the model derived from translating one of the JML specification annotations by hand to overcome a limitation in our compiler. An error in the hand translation yielded an assertion check that was short-circuited and, hence, never been checked. Second, as shown in Table 1, the coverage percentage for this program was very low (55.4%). It turns out that the environment for this program was inadequate for exercising certain implementation behaviors. As discussed in section 2, this was caused by failure to execute code in method `take()`.

The other program that also had low structural coverage is the *ReplicatedWorkers*. This time the low coverage was due to the fact that our compiler generates BIR code for abstract classes. Bogor only needs the concrete classes to perform the checking, therefore all the code generated from the abstract classes is never executed. Based on these observations, we are modifying the code generation of the compiler and the manner in which coverage information is accumulated to (a) avoid reporting spurious coverage problems (e.g., like those associated with abstract classes) and (b) incorporate options that allow the user to selectively mask non-coverage reports for different types of code such as exception handlers.

4.2 Boolean Coverage Estimation with MAnTA

Structural analysis was very useful in spotting environment limitations. However, as we mentioned in section 1, we are also interested in analyses that help extract information coverage at the specification level.

Coverage analysis at the specification level has been found useful in the past for specification refinement. For example, Hoskote et al. [13] show how they used a coverage analysis similar to the one described in [4] to refine the specification of several circuits from a microprocessor design. In our case, we are very interested in this type of analysis because of our goal of checking strong JML specifications. The type of specifications that can be expressed in JML are very rich in terms of expressing complex

Model	Total Number of Monitored Formulas	Number of Unsatisfied Formulas
LinkedQueue	20	2
BoundedBuffer	15	0
RWVSN	14	0
DiningPhilosophers	40	0
ReplicatedWorkers	25	0

Table 2. Number of formulas not satisfied during model checking.

```
/*@ behavior
  @  requires x != null;
  @  ensures true;
  @ also behavior
  @  requires x == null;
  @  signals (Exception e) e instanceof
  @   IllegalArgumentException; @*/
public void put(Object x) {
 if (x == null)
  throw new IllegalArgumentException();
 insert(x);
}
```

```
/*@ behavior
  @  assignable head, head.next.value;
  @  ensures \result == null
  @   || (\exists LinkedNode n;
  @       \old(\reach(head)).has(n);
  @       n.value == \result
  @       && !(\reach(head).has(n))); @*/
Object extract2() {
 Object x = null;
 LinkedNode first = head.next;
 if (first != null) {
  x = first.value;
  first.value = null;
  head = first;
 }
 return x;
}
```

Fig. 7. Methods in `LinkedQueue` that had unsatisfied formulas.

properties of heap allocated data, and in covering all the different behavioral cases of a method. Writing this type of specification is very hard and demands a lot of expertise. Even when written by an expert, the potential for overlooking or over-constraining aspects of method behavior is significant. We believe that coverage analysis can be successfully used for specification debugging.

In this section we describe an analysis we implemented for this purpose. This additional coverage estimation analysis is a simple boolean satisfaction analysis. Again, the implementation only involved developing the logic of the coverage analysis module and configuring the coverage framework to interact with the model checker. The coverage criterion was a simple monitoring policy: for each boolean expression, determine whether there are sub-formulas of the expression that are never satisfied. If there are such formulas, then report them as possible coverage problem, along with their position in the source code.

Table 2 summarizes the results of the analyses. The only program that showed lack of coverage under this criterion was the *LinkedQueue* example. Figure 7 shows the two methods that had a problem of low coverage.

This analysis uncovers problems in different methods than the structural coverage analysis. For method `put()` the tool reports that the condition **`x == null`** is never satisfied in the precondition. This is to be expected since this condition corresponds to exceptional behavior. In some sense, the structural coverage test also uncovered the

same issue because it reported that the `throw` instruction was never executed. But the boolean analysis gives a more direct indication of the conditions that led to the `throw` code not being covered.

For the `extract2()` method the tool reports that `\result == null` never holds in the postcondition. After looking carefully we note that this condition corresponds to the case when the list is empty. We already determined, with the information from the previous test, that the list was never made empty. However, the problem has now been exposed at a different point. In the first test, we spotted the problem *from the code*. Now, we do so *from the specification*. This phenomenon suggests that these two techniques are complementary in the types of coverage problems they can find. They uncovered the same problem from different perspectives.

4.3 Discussion

The basic forms of coverage analyses we implemented have demonstrated that individual coverage analyses can be effectively integrated with model checking to provide useful information about the quality of environments. We believe that multiple coverage analyses, when aggregated, can yield additional benefit on top of what can be obtained by using them independently.

To see this, let us consider together the results obtained in Sections 4.1 and 4.2. For the *LinkedQueue* example, both analyses showed that there was a behavior, namely the blocking behavior of the list, that was not being exposed. On one hand, the structural analysis spotted this problem by pointing directly at the portion of code that was not being executed. On the other, the boolean analysis uncovered the same problem, but this time pointing at a specification case that was never enabled. Independently, the two analysis would give the same benefit: find a gap in the coverage produced by an environment. However, when run together they provide two extra benefits and we discuss them in turn.

First, the obvious extra benefit is the increase in the reliability of the reports given by each analysis. Certainly, if both analysis point at the same problem, then the chances are higher that the problem is actually a concern and not just some superficial issue (as discussed previously, some analyses may report spurious or superficial coverage problems, as is the case for structural analysis when it pointed to transitions corresponding to abstract classes). The second benefit, not so apparent, but extremely useful, is in the way in which, when working together, these two analyses can help not only by providing estimation of coverage, guiding in that way the refinement of the test harness, but also in the verification process itself by uncovering potential errors, both in the implementation and in the specification.

To see this last point keep in mind the duality of these two approaches. For each portion of the model not covered, every analysis should report some information in the form of transitions not-executed and conditions not-enabled, for structural and boolean analysis, respectively. Now, consider Figure 8-a. Suppose further that it is model checked with an environment such that the method is always called in a state where `b == true`. Obviously, the boolean satisfaction analysis will report that the pre-condition `!b` is never true. However, the structural analysis will report complete coverage. This mismatch suggests that the implementation is missing code that represents the specification

```
/*@ behavior
  @  requires b;
  @  ensures a == 1;
   @also
  @ behavior
  @  requires !b;
  @  ensures a == 2;
  @*/
public void m(boolean b) {
 if (b)
  a = 1;
}
```

(a)

```
/*@ behavior
  @  requires b;
  @  ensures a == 1;
  @*/
public void m(boolean b) {
 if (b) {
  a = 1;
 } else {
  a = 2;
 }
```

(b)

Fig. 8. Example methods that show how structural and boolean analysis can be complemented for finding bugs and debugging the specification.

that is not covered, otherwise that code would be reported as not executed. Or if the code is not missing, then it must contain bugs because it does not match the specification. In fact, when we examine this code we can see that the implementation is incomplete. Results from analyses that separate would be disconnected information, together have helped to find a bug.

Notice that a model checker would not be able to find this bug because the environment is incomplete in the first place. And the boolean analysis by itself only suggests that the environment should be refined. Indeed, after refining the environment the model checker would spot the error, but this would require another model checking run. On the other hand, by analyzing the information from the coverage analyses, we can find the same bug with much less effort.

Similarly, if we look at Figure 8-b, and if we consider the same environment, we see that in this case we have the opposite situation. Now, the structural analysis would report that a portion of the method is never executed. However, the boolean satisfaction analysis would report that the specification is completely covered. Indeed, the implementation is correct with respect to the specification. However, the mismatch between the two analysis suggests the possibility that the specification is not complete. This must be true for otherwise there would be some portion of the specification that would not be covered because the state space that it represents is never explored. If the specification is already totally satisfied, it means that it is insensitive to the code that was not executed. In fact, when we look at the specification we see that, indeed, the specification is incomplete: it does not handle the case when `b == false`. All this would lead to a refinement of the specification to include the case that is not handled (if that is what we really want). In this case, the benefit is not in finding bugs in the implementation, but in *specification debugging*.

Coverage-enabled Model Checking All the concepts presented above suggest a general methodology for software verification using model checking extended with these coverage analyses:

- Run the model checker on the system to be verified with the two coverage analysis extensions described in this work.
- If the structural analysis reports complete coverage, but the boolean analysis reports incomplete coverage, then there is potential for an incomplete implementation (a bug caused by missing code for a specification case).
- If the structural analysis reports incomplete coverage, but the boolean analysis reports complete coverage, then there is potential for an incomplete specification (at the very least, the specification is insensitive to the code that was not covered).
- If both analyses report incomplete coverage, there should be a correspondence of non-covered cases from one analysis to the other (as in our example in sections 4.1 and 4.2). If such correspondence exists, then the coverage problems can be attributed to the test harness. Otherwise, there is potential for implementation/specification incompleteness, depending on which analysis has an unmatched non-coverage case.

Note that this extra benefit cannot be achieved if only the model checker is run, or if the analyses are included but done independently. For example, suppose we have the case where the specification is incomplete and it does not cover some code that, additionally, is not executed by the test harness. The model checker would not detect the problem because from the point of view of the model checker there is no problem: the implementation satisfies the specification. The boolean analysis alone would not report any problem, and the structural analysis would simply point out that the test harness must be refined to cover the code not-executed.

Also, notice that this methodology will only detect incomplete specification cases when it is accompanied by the same incompleteness in the test harness. For example, in figure 8-b, if the test harness does cover the case when `b == false`, then both analyses, structural and boolean, achieve 100% coverage and nothing can be inferred. Spotting these specification incompleteness cases is another interesting direction that we want to pursue.

5 Related Work

Although coverage estimation has not received as much attention as other areas in model checking, there is still some work that has been done to address this problem. Among the most significant works in this area is the work done by Chockler et al. [4]. However, their definition of the coverage problem is slightly different. The concern in their work is the following: given a model and a specification, how much of the system state space is relevant to the specification. For example, consider the simple transition system in Figure 9. The specification shown with the system holds in the initial state because any of the two outgoing paths lead to the state where `q` is always true. However, the third state, where `r` is true, is totally insensitive to this specification. Under this definition of coverage we say that the node is not covered by the specification, so we have to refine the properties. To find these states, this technique would change the value of specific variables (for example, `p`) in each state and run the model checker. Those nodes in which the property does not fail, are nodes insensitive to the specification, and thus not covered.

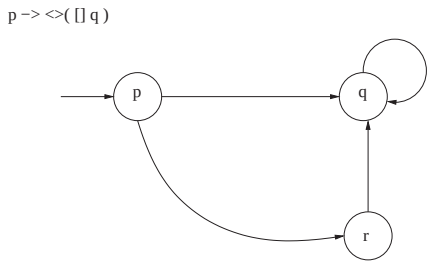

Fig. 9. Simple transition system and a specification to be verified.

The information obtained with this type of analysis gives a rough estimation of the proportion of behaviors covered by the specification. This approach, although useful, has several shortcomings. First of all, some specifications necessarily refer to only a portion of the states, due to the nature of the property they express. In this case, the specification does not need to be refined and the coverage information is not very useful. Second, the cost of the analysis is the same as that of model checking itself (two model checks must be run – the first "regular" check, and then a second in which the values of the basic propositions are negated). However, as we show in this paper, by simply adding some book-keeping procedures to the model checker algorithm, some very useful coverage information can be gathered to support model refinement, at a much lower cost. For example, the structural analysis we implement simply needs to keep track of the transitions that are executed.

The combination of specification and structural coverage analysis yields results that are similar to that obtained by [4], but with much less effort. In that work, the analysis detects portions of the state space that are not covered by any aspect of the specification, pointing directly at those points of the state space. However, to do this an execution separate from the verification run must be made, which is as expensive as the model checking itself. On the other hand, our approach can be run while model checking the system, and the analysis information is done by monitoring the checking process. This monitoring is very inexpensive (with respect to one analysis, although the cost will grow as more analyses are attached to the framework). Nevertheless, our system won't point directly at the regions of the state space that are not covered and requires insight from the user to find this regions, once the evidence of the potential problem is found.

Another idea, explored by the same researchers mentioned in the previous paragraph, is the concept of vacuity checking [14]. This is a boolean satisfaction analysis. The idea is to verify that there are no formulas in the system that are being trivially satisfied, that is, that are proven correct by the model checker, but their truth is independent of the model. They show, surprisingly enough, how vacuity checking is harder to perform than one might imagine and that it can be very common in several models. Again, the usefulness of this analysis is limited just to some cases. Although vacuous formulas in specifications might be more common than expected, the calculation tends to be fairly expensive, yielding information about just one specific type of errors. For example, the approach described in the previous paragraph would also detect this formulas because the whole system would be insensitive to the formula. To see this, just

consider what happens if p is made false in the initial state of the system in Figure 9. In this case, the formula is vacuously satisfied in all the states of the system, and would be detected because every state would be insensitive to this formula. Although it would be more expensive, the cost would be amortized by all the other type of errors that this approach would also find.

Musuvathi and Engler have used some coverage metrics in the verification of a TCP implementation using model checking [17]. Their coverage metric is based on simple line coverage, that is, whether a line of code is executed or not. However, their usage of the metric and motivation is quite different from ours. They use the coverage metrics to evaluate the effectiveness of the state space reduction techniques implemented in their model checker. However, they do note how identifying unexplored portions of the state space can help refine the test harness used to model check a system. Similarly, SPIN [11] has had for some time a feature through which it reports statements that are never reached.

In another similar line of work, Groce and Visser explore the use of coverage metrics to make heuristic-based model checking [9]. The coverage metric they use is branch coverage, which is exactly the one used in our structural coverage analysis. However, the purpose of this work is to use the information to both estimate the effectiveness of *partial checks* and use the branch coverage measure as a value function to guide the search in heuristic model checking.

Although the coverage metrics reported in the works mentioned in the previous two paragraphs are equivalent to our structural coverage, there are several differences in the usage of the metrics and in the motivations. The main difference is that, while those focus on using the metric to obtain a measure of how much of the state space is explored (useful in bounded search or when the model checker runs out of memory, i.e., partial checks), our focus is to use the metric to refine test harnesses used to model check open systems. Also, we have integrated the structural coverage with a specification coverage and proposed exploiting this integration to enhance the verification. This has to do with the main motivation for our work: we want to check *strong specifications* in software units and are deeply interested in verifying the depth to which the specifications are covered. Our interest in refining the test harnesses is focused towards increasing specification coverage rather than code coverage. This is why we have integrated the structural analysis with a boolean analysis for the specification.

The Bandera Environment Generator (BEG) project [21], takes another approach to providing tool support for deriving appropriate test harnesses and environments. It focuses on automatically generating test harnesses from high-level specifications of orderings of calls to the module being tested and from information gathered from automated static analysis (e.g., about side-effects and aliases) of the code being analyzed. It lets the user specify: (a) the components that are relevant for the verification, (b) constraints over these components, and (c) certain assumptions about the environment (such as the order in which methods in the unit being analyzed should be called), and then generates an environment made up of stubs and driving code that then is translated to the model checker language. Although the work in BEG is certainly relevant to the coverage issue, it is rather complementary to the work presented herein: BEG tries to generate adequate environments that achieve good coverage from user provided infor-

mation (such as a regular expression indicating a pattern of method calls to the unit under test), whereas MAnTA estimates the coverage achieved by a given environment (whether it is generated automatically or by hand). For example, even though the user provides a regular expression to specify the ordering of unit method calls in the test harness, this ordering may still not give good coverage (and MAnTA can be used to detect these situations).

The coverage problem has been studied extensively by the testing community [2]. In this case, all the studies made in test coverage, for example [23], are directly relevant to the work presented in this paper. Several notions of structural coverage have been developed by the testing community. For example, there is the concept of node coverage, branch coverage, path coverage, etc. All this work includes potential lessons to be applied in monitoring coverage in model checking.

6 Conlusions

This paper presented MAnTA – a coverage analysis framework built on top of the Bogor software model checking framework to guide effective construction of environments for model checking. MAnTA allows a variety of coverage analyses of both code (e.g., branch coverage) and specification (e.g., boolean satisfaction of method pre/post-conditions) to be integrated in the model checking process. Since model checking is exhaustive (i.e., all interleavings are considered), thus, the results of the coverage analyses can pinpoint deficiencies in the environments used in the system. For example, a branch analysis can be used to determine code regions that are not exercised due to the specificity of the environment. Another example, the exercised behaviors of a particular method may not be sufficient because it always satisfy *a specific* pre-condition of the method instead of *each* of the pre-condition of the method.

MAnTA eases the incorporation of these kinds of coverage analyses by providing a plugable API for adding new coverage analysis. We showed the effectiveness of the framework by easily incorporating widely used coverage analyses in the testing community as well as by implementing a novel specification coverage analysis. We showed how the analyses can uncover common deficiencies in test harnesses with respect to code and specification. Therefore, we believe that MAnTA can be used to guide analysts when constructing environments such as usually done in unit testing.

References

1. T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, 2001.
2. R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.
3. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
4. H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi. A practical approach to coverage in model checking. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 66–78. Springer-Verlag, 2001.

5. M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Designs*, 2004. (to appear).
6. M. B. Dwyer, J. Hatcliff, and D. Schmidt. Bandera: Tools for automated reasoning about software system behaviour. *ERCIM News*, 36, Jan. 1999.
7. D. R. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference*, volume 2937 of *Lecture Notes in Computer Science*, pages 191–210. Springer, 2004.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., Jan. 1995.
9. A. Groce and W. Visser. Model checking java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–21. ACM Press, 2002.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.
11. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
12. G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proceedings of the 21st international conference on Software engineering*, pages 597–607. IEEE Computer Society Press, 1999.
13. Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proceedings of the 36th ACM/IEEE Design automation conference*, pages 300–305. ACM Press, 1999.
14. O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1999.
15. D. Lea. *Concurrent Programming in Java: Second Edition*. Addison-Wesley, 2000.
16. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, Oct. 1998.
17. M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of The First Symposium on Networked Systems Design and Implementation*, pages 155–168. USENIX Association, 2004.
18. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the deos scheduler kernel. In *Proceedings of the 22nd international conference on Software engineering*, pages 488–497. ACM Press, 2000.
19. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
20. Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2004.
21. O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 18th International Conference on Automated Software Engineering*, Oct. 2003.

22. O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the Fourth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 2003.
23. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.