Bounded verification of voting software

Bounded Verification of Voting Software

Greg Dennis, Kuat Yessenov, and Daniel Jackson

Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
{gdennis, kuat, dnj}@mit.edu

Abstract. We present a case-study in which vote-tallying software is analyzed using a *bounded verification* technique, whereby all executions of a procedure are exhaustively examined within a finite space given by a bound on the size of the heap and the number of loop unrollings. The technique involves an encoding of the procedure in an intermediate relational programming language, a translation of that language to relational logic, and an analysis of the logic that exploits recent advances in finite model-finding. Our technique yields concrete counterexamples – traces of the procedure that violate the specification.

The vote-tallying software, used for public elections in the Netherlands, had previously been annotated with specifications in the Java Modeling Language and analyzed with ESC/Java2. Our analysis found counterexamples to the JML contracts, indicating bugs in the code and errors in the specifications that evaded prior analysis.

1 Introduction

First deployed for public elections in the Netherlands in 2004, the KOA Remote Voting System is an open-source, internet voting application written in Java. Intended to be used primarily by expatriots, KOA stands for the Dutch phrase "Kiezen op Afstand", which literally means "Remote Voting".

The KOA application contains a small vote-tallying subsystem that processes the ballot data and counts the votes. The vote-tallying module was developed independently of the rest of the application by the Security of Systems (SoS) research group at the University of Nijmegen, the developers of ESC/Java2 [1]. In building the module, SoS annotated their Java source code with specifications in the Java Modeling Language (JML) [2] and used their own ESC/Java2 tool to check the code against those specifications [3–6]. The code was also tested with the unit-testing tool *jmlunit* [7].

For our case-study, we used our own program analysis tool *Forge* [8] to check the vote-tallying code against the provided JML specifications. Our analysis found counterexamples to the JML contracts that evaded both the unit tests and ESC/Java2. Each counterexample is reported by Forge as a trace of the procedure under analysis, and each could indicate a bug in the code or an error

in the specification. The purpose of the study was to explore the benefits and limitations of our approach, compare it to existing techniques like ESC and testing, and look for opportunities for synergy.

Our work is part of a larger, ongoing effort to develop a new technique for automatically checking object-oriented code against functional specifications. The basic idea, originating with Vaziri [9] and furthered by Taghdiri [10], involves translating a procedure in object-oriented code into a formula in the relational, first-order logic of Alloy [11] and invoking a finite model-finder to search for solutions to the formula. The model finder translates the relational formula to a boolean satisfiability (SAT) problem and hands it to a SAT solver. If the solver finds a solution, the model finder translates it to an instance of the relational formula, which is then converted into a trace of the procedure.

The approach, one of a larger class of analyses we refer to as *bounded verification*, examines all executions of a procedure up to some finite bound on the size of the heap and number of loop unrollings and reports counterexamples as actual procedure traces. It finds a counterexample if one exists within the provided bounds without issuing false alarms (given provisos described in Section 2.2); however, it will always miss bugs that require larger bounds for their detection. The effectiveness of the technique rests on the "small-scope hypothesis," the claim that many defects have small counterexamples, an idea consistent with our own experience and empirical evaluation [12, 13].

Our prior work [13] introduced a more efficient translation of code to relational logic that exploited advances in the Kodkod model finder [14], the engine behind the latest version of the Alloy Analyzer. A proof-of-concept tool was demonstrated that worked for a small subset of Java and JML. This was a promising first step, but, as a tool applicable to non-trivial programs like KOA, it fell short. It was unable to handle common program features, including inheritance and arrays. Its more fundamental limitation, however, was a lack of support for fully modular analysis and datatype abstraction. Consequently, called procedures could not be summarized by specifications, and their code had to be inlined, causing the analysis to scale poorly.

To achieve full modularity, we turned our proof-of-concept into a more mature framework for bounded verification, a new component of which is an intermediate representation of code called the *Forge Intermediate Representation* (FIR). FIR is a simple relational programming language, capable of expressing imperative statements, declarative specifications, and relational abstractions, within a single small grammar. Our new approach is to translate both Java and JML into FIR and then apply our existing bounded verification technique to the resulting FIR program. The new framework is what made this case study practically feasible.

In the course of developing the bounded verification tool and technique, some questions were repeatedly raised, by ourselves and others. How useful is the technique? Is the "small-scope hypothesis" empirically justified? How does our technique compare to existing techniques? What are the key areas of future work on this project? The paper uses the results of the case study to help answer these questions.

2 Approach

Our early work introduced a new encoding of Java code in relational logic [13]. With the introduction of the new Forge Intermediate Representation, a full analysis of Java with Forge now involves a three-stage translation: from Java to FIR (Section 2.1); from FIR to relational logic (Section 2.2); and lastly, from relational logic to a boolean satisfiability problem (via the existing Kodkod tool).

2.1 From Java to FIR

The Forge Intermediate Representation (FIR), is an imperative, relational programming representation, capable of expressing both programs and their specifications. FIR is not a textual syntax; rather it consists of data structures assembled through an API. Like most intermediate representations, FIR was designed to be simple and uniform so as to be amenable to automatic analysis.

When we say FIR is "relational," we mean that every expression in the language evaluates to a *relation*, i.e. a set of *tuples*, where each tuple is a sequence of *atoms*. The arity of a relation (the length of its tuples) can be any positive integer. A set of atoms can be represented by a unary relation (a relation of arity 1), and a scalar by a singleton set.

The translation of object-oriented programs to FIR is based on a *relational view of the heap* [15], in which program data values are interpreted as relations. Specifically, types are viewed as sets; fields as functional relations; and local variables as singleton sets. To illustrate, we will refer to the Java example in Figure 1 and its resulting FIR translation in Figure 2. For simplicity, the example translation ignores complexities arising from exception handling (which our tool does support).

For each concrete class in the code, the translation creates a corresponding FIR *domain* to represent the set of objects whose runtime type is that class. For the code in Figure 1, our translation declares three domains: Birthday, Month, and Object (Figure 2, Line 00). Two domains – Boolean and Integer – are built-in.

The translation maps each Java static type to a FIR type, either a domain or some combination obtained by the union or cross product of domains. The static types `Birthday` and `Month` are mapped to the FIR domains Birthday and Month respectively. Because every Java class is a subclass of `Object`, the translation maps the static type `Object` to the FIR union type Birthday ∪ Month ∪ Object.

The translation encodes each Java field as a global variable that maps members of the enclosing class to members of the field's type. For example, the Java field `month` is encoded as a FIR global month whose type is Birthday → Month (Line 01). Similarly, the translation creates global variables day: Birthday → Integer and maxDay: Month → Integer (Lines 02-03). The translation adds side constraints (not shown) that these binary relations are functions.

For each Java parameter and local variable in the method under analysis, the translation declares a local variable of the corresponding type, whose value is constrained to be a scalar. For the `setDay` method, the translation creates four

```
class Birthday {
    /*@ non_null */ Month month;
    int day;

    //@ requires this.month.checkDay(d);
    //@ ensures this.day == d;
    void setDay(int d) {
        Month m = this.month;
        boolean dayOk = m.checkDay(d);
        if (dayOk) this.day = d;
    }
}

class Month {
    int maxDay;

    //@ ensures \result <==> (d > 0 && d <= maxDay);
    /*@ pure */ boolean checkDay(int d) { . . . }
}
```

Fig. 1. Birthday Example in Java with JML

```
00  domain Birthday, domain Month, domain Object
01  global month: Birthday → Month
02  global day: Birthday → Integer
03  global maxDay: Month → Integer
04  local this: Birthday, local d: Integer
05  local m: Month, local dayOk: Boolean
06
07  proc setDay (this, d) : ()
08    m = this.month;
09    dayOk = spec (dayOk ⇔ (d > 0 ∧ d ≤ m.maxDay));
10    if dayOk then day = day ⊕ (this → d) else exit;
```

Fig. 2. Translation of `Birthday.setDay` into FIR

FIR local variables: this of type Birthday, d of type Integer, m of type Month, and dayOk of type Boolean (Lines 04-05).

The FIR expression language is essentially the same as that offered by the Alloy modelling language [11]. Expressions can be built from any of the following: set operators (union, intersection, difference); relational operators (join, cross product, override, transitive closure); boolean, arithmetic, bitwise operators; set comprehension; and universal and existential quantification.

The result of translating the `setDay` Java method is the FIR setDay procedure, which has two inputs – this and d – and no outputs (Line 07). The procedure begins by assigning the FIR expression this.month to the local variable m (Line 08). Although the FIR statement looks nearly identical to its Java counterpart, the dot (.) operator in FIR stands for relational join, not field deref-

erence. When representing Java fields as functional relations and Java locals as singleton sets, field dereference can be encoded as relational join, because the join of a singleton set and a function will always yield a singleton.

The call to `checkDay` in the Java code has been encoded in FIR as a *specification statement* (Line 09) that embeds a declarative specification in imperative code [16, 17]. FIR uses specification statements to facilitate modular analysis: once a procedure is found to meet a specification, calls to that procedure can be replaced with instantiations of its specification.

The specification statement in setDay is a FIR encoding of the JML contract for the `checkDay` method. Any variables on the left-hand side of the specification statement, in our example only dayOk, may be modified by the statement. On the right-hand side is a condition that the analysis establishes with demonic nondeterminism [18]; that is, the analysis will check the procedure for all executions that satisfy the condition.

An assignment to a field in Java is encoded using a relational override ($\oplus$) expression in FIR. The value of the FIR expression day $\oplus$ (this $\rightarrow$ d) is the relation containing the tuple (this $\rightarrow$ d) and any tuples in day that do not begin with this. Thus, the assignment day = day $\oplus$ (this $\rightarrow$ d) (Line 10) encodes the Java statement `this.day = d`. Note that the FIR assignment statement updates the *entire* value of the day relation (so that the symbolic execution mentioned below can compute a simple expression – not involving quantifiers or comprehensions – for the value of day at the end of the statement).

Datatype Abstractions. The translation from Java to FIR employs some useful datatype abstractions for common library collections, including sets, maps, and lists. Using abstractions of these collections in place of their concrete representations reduces the complexity of the resulting FIR program to be analyzed, thereby improving the performance of the analysis.

`Sets` are modelled abstractly by a binary relation whose domain is the Java `Set` objects and whose range is the elements in the set. A map is abstracted with a ternary relation that contains (Map $\rightarrow$ key $\rightarrow$ value) tuples. Lists and arrays are abstracted by ternary relations containing (List $\rightarrow$ index $\rightarrow$ value) tuples.

2.2 From FIR to Relational Logic

From a FIR procedure, the tool automatically obtains a formula P(s, s') in relational logic that constrains the relationship between a pre-state s and a post-state s' that holds whenever an execution exists from s that terminates in s'. A second formula S(s, s') is obtained from a user-provided specification, and its negation is conjoined to the first to obtain:

$$\mathsf{P}(\mathsf{s}, \mathsf{s'}) \wedge \neg \mathsf{S}(\mathsf{s}, \mathsf{s'})$$

which is true exactly for those executions that are possible but that violate the specification.

The translation of procedural code to relational logic uses a symbolic execution technique that traverses each branch in the code, building a symbolic relational expression for each variable at each program point. Although our earlier work [13] presented the technique in the context of Java code, applying it to FIR is straightforward (in fact, much simpler due to FIR's relational structure).

In addition to the procedure and its specification, a client of Forge must provide a bound on the analysis consisting of:

- the number of times to unroll loops;
- the bitwidth limiting the range of FIR integers; and
- one *scope* for each domain, i.e. a limit on the number of its instances that may exist in any heap reached during execution.

Each of these limits results in under-approximation, eliminating possible behaviors but never adding behaviors. Thus, any counterexample generated will be valid – either demonstrating a defect in the code or a flaw in the specification. If a counterexample exists within the bound, one will be found, though defects that require a larger bound will be missed.

The chosen bound and the relational formula are handed to the Kodkod model finder. Kodkod translates the formula and the bound into a boolean satisfiability (SAT) problem, which it passes to an off-the-shelf SAT solver. If the solver finds a solution, Kodkod maps it to an instance of the relational logic formula, which Forge then maps to a counterexample trace of the original FIR procedure.

Soundness and Incompleteness. The only imprecision introduced by the translation from FIR to relational logic are the under-approximations given explicitly in the user-provided bound, so an analysis of FIR is sound in general and complete within the bounds. However, the translation from Java to FIR does not handle all of Java and also employs some optimizations, which introduces imprecision that may lead to spurious counterexamples and missed counterexamples.

One potential source of false alarms is the bounding of integers to a bitwidth less than that of Java integers. Consider an analysis in a bitwidth of 5 that produces a counterexample due to integer overflow. Because Java integers have a larger bitwidth, this counterexample does not represent an actual trace of the code. However, an overflow error in a small bitwidth, in our experience, usually indicate the presence of an analogous counterexample in a larger bitwidth. That is, if the code can overflow at a bit width of 5, it can probably overflow at 32.

Upon a method call, our analysis requires only the invariants of the receiver object hold. ESC/Java, in contrast, requires the invariants hold for every argument of a call. Exactly which invariants hold should upon method calls is an open area of research for modular analysis techniques generally.

The other sources of imprecision in the Java and JML translations are: lack of support for real number arithmetic, I/O, static initialization, reflection, `ArrayStoreException`s; incomplete support for String parsing; treatment of exceptions as singletons; unsound treatment of object equality; and unrolling of recursive specifications.

3 Analysis of KOA

At the time of the initial release of KOA in 2004, 47% of the core (non I/O) methods had been verified with ESC/Java2. The rest did not verify due to either non-termination of ESC's Simplify theorem prover, incompleteness in ESC, or unspecified "invariant issues." The code was also tested using *jmlunit*, which generated nearly 8,000 unit tests, all of which passed. Since that time, the code has been improved and further analyzed with ESC/Java2 [4].

Our analysis of the KOA vote-tallying software centered on eight classes, listed in Table 1, that form the core of its functionality. The `AuditLog` class logs the progress of the vote-tallying; `Candidate` records the tally of an individual candidate; `CandidateList` pairs a list of candidates in an election with a `CandidateListMetadata` that stores additional properties of the election; `District`, `KiesKring`, and `KiesLijst` are kinds of political district boundaries; and `VoteSet` records the cumulative tally for all candidates in the election. The *methods* column in Table 1 lists the number of methods analyzed in each class.

When Forge is applied to the methods of a class, the performance of the analysis depends not only upon the complexity of the code in the class but also upon the complexity of its specification, as well as the specifications of classes upon which that class depends. The *sloc* column in Table 1 gives the number of source lines of code in each class; *slocc* includes code and comment lines. Because JML is written inside Java comments, the slocc measures, albeit indirectly, the complexity of the class' code and specification. The *dslocc* is the *slocc* plus the number of lines of comment in classes upon which the class directly depends. The *dslocc/method* approximates the complexity of a modular analysis of a method within the class.

As shown in the table, we applied Forge to a total of 169 methods of varying complexity across the eight classes. The *violations* column lists the number of methods that were found to violate their specification. A total of 19 specification violations were found. The experiments were run on a Mac Pro with two 3GHz Dual-Core Intel Xeon processors and 4.5GB RAM running Mac OS X 10.4.11. (Forge is single-threaded and so it did not take advantage of the multiple cores.) The code on which these analyses were conducted is, as of the time of this writing, the latest version available in its Subversion repository.

We initially analyzed each method in a scope of 5 instances of each type, an integer bitwidth of 4 (integers -8 to 7), and 3 loop unrollings. Most analyses completed quickly, but a few of the more complex methods exceeded our timeout of four hours. For those that timed-out, we progressively lowered the scope until the analysis completed within the time limit. The *mean scope* column in the table lists the average maximum scope in which the analysis successfully completed, and *mean time (sec)* is the mean time in seconds for a successful analysis. Note that these means are calculated over the analyses of the methods within a class, not over successive analyses of the same class. As shown in the table, the average analysis time is roughly correlated with the dslocc/method measure.

class	methods	sloc	slocc	dslocc	dslocc/ method	violations	mean scope	mean time (sec)
AuditLog	90	286	1237	1617	18.0	1	5.0	2.3
CandidateListMetadata	10	72	246	643	64.3	1	5.0	33.6
Candidate	12	103	363	1013	84.4	1	5.0	59.3
KiesKring	15	119	482	1272	84.8	5	5.0	249.7
District	6	53	163	543	90.5	0	5.0	18.5
KiesLijst	12	111	367	1432	119.3	4	5.0	104.6
CandidateList	13	130	493	1746	134.3	3	4.5	1416.8
VoteSet	11	113	400	2688	244.4	4	3.7	1783.9
Sum or *Mean*	**169**	**987**	**3751**	**10954**	*64.8*	**19**	*4.9*	*262.7*

Table 1. Summary analysis statistics of each class. The means are calculated over the analyses of the methods within a class, not over successive analyses of the same class.

3.1 Specification Violations

Table 2 gives statistics on the 19 specification violations detected. The methods named `init` in the table are constructors. We inspected every violation and determined that none were false alarms (although theoretically possible, as noted at the end of Section 2.2). To evaluate the "small-scope hypothesis", for each violation found we progressively lowered the bound on the analysis (scope of each type, bitwidth of integers, number of loop unrollings) until the analysis no longer detected the counterexample. The minimum bound under which each counterexample is found is given in the last column of Table 2.

Every specification violation can be attributed to one of two causes: a bug in the code or an error in the specification. As outside observers, we can make educated guesses as to the cause, but classifying the violation with complete certainty requires knowing the programmer's intention. Does the specification accurately reflect the programmer's intention, in which case the violation indicates a bug in the code; or did the programmer err in transcribing his or her intention into the specification?

Specification errors themselves can be divided into two subcategories: *overspecification* and *underspecification*. A case of *overspecification* occurs when the specification of the method under analysis requires too much from the implementation – either its pre-condition is too weak or its post-condition is too strong. A case of *underspecification* occurs when the method under analysis calls a method whose specification provides too little to the caller – either the pre-condition of the called method is too strong or its post-condition is too weak. (Recall that our analysis, being fully modular, assumes that the specifications, not the implementations, of called methods define their behavior.)

Specification errors, while not "bugs" per se, are still important to address. For example, there may be methods whose correctness depends on the overspeci-

class	method	error	min bound
CandidateListMetadata	init	under	1 / 3 / 1
KiesKring	addDistrict	bug	1 / 3 / 1
VoteSet	addVote(String)	over	1 / 3 / 1
KiesLijst	clear	over	1 / 3 / 3
AuditLog	getCurrentTimeStamp	over	2 / 1 / 1
Candidate	init	under	2 / 3 / 1
CandidateList	addDistrict	under	2 / 3 / 1
CandidateList	addKiesLijst	over	2 / 3 / 1
CandidateList	init	over	2 / 3 / 1
KiesKring	addKiesLijst	bug	2 / 3 / 1
KiesKring	init	under	2 / 3 / 1
KiesKring	make	under	2 / 3 / 1
KiesLijst	addCandidate	over	2 / 3 / 1
KiesLijst	compareTo	bug	2 / 3 / 1
KiesLijst	make	over	2 / 3 / 1
VoteSet	addVote(int)	over	2 / 3 / 1
VoteSet	validateKiesKringNumber	over	2 / 3 / 1
VoteSet	validateRedundantInfo	over	2 / 3 / 1
KiesKring	clear	over	2 / 3 / 3

Table 2. Specification violations: error classification and minimum bound (scope/bitwidth/unrollings) necessary for the error's detection.

fication of a called method. Fixing the overspecification may, therefore, reveal latent bugs in dependent methods. In contrast, a case of underspecification doesn't pose an immediate problem, but it could allow a bug to be introduced in the future. That is, the underspecified method's implementation could be changed at a later date in a way that still satisfies its contract, but causes dependent methods to fail.

As shown in Table 2, of the 19 violations found by Forge, we believe 3 to be due to buggy code, 11 due to overspecification, and 5 due to underspecification. From our follow-up "smallest scope" analyses of each violating method, we found that every violation would have also been found in a scope of 2, a bitwidth of 3, and 3 loop unrollings.

In fact, all but two of the violations required only 1 loop unrolling, the exceptions being `KiesKring.clear` and `KiesLijst.clear`. Both `clear` methods contain loops with if-statements in the body of the loop, and 3 unrollings were necessary to cover all paths. Additionally, four violations needed only the minimal scope of 1 and one violation was found in the minimal bitwidth of 1.

A minimal bitwidth of 3 (integers from -4 to 3) was needed for nearly every analysis, because some static array fields in the code were required to be of at least length 2. Lowering the bitwidth to 2 would allow a maximum integer of only 1. These arrays were not required to be fully populated, however – they could contain null elements – so their minimal length requirements did not in turn affect the minimal scope necessary to detect violations.

3.2 Example Violations

In this section, we present and discuss a sample of four specification violations detected by our analysis, two of which we've classified as bugs, one overspecification, and one underspecification. For brevity, some of the code excerpts and JML specifications shown below have been simplified.

(a) KiesLijst.compareTo [bug] The code for the method is a correct implementation of the `compareTo` method in `Kies`*`Kring`*, not `Kies`*`Lijst`*. This is likely a copy-and-paste error:

```
class KiesLijst {
  public int compareTo(final Object an_object) {
    if (!(an_object instanceof KiesKring)) {
      throw new ClassCastException();
    }
    final KiesKring k = (KiesKring) an_object;
    return number() - k.number();
  }
}
```

The `instanceof` check and the initialization of local variable `k` should refer to `KiesLijst`, not `KiesKring`.

(b) KiesKring.addDistrict [bug] The `KiesKring` class stores an array of districts in the `my_districts` field and a count of the number of districts in the `my_district_count` field. The specification for `KiesKring` includes an invariant that the count is equal to the number of non-null entries in the array:

```
private final /*@ non_null @*/ District[] my_districts
private byte my_district_count;
//@ invariant my_district_count == (\sum int i; 0 <= i && i < my_districts.length;
//@                                             (my_districts[i] != null) ? 1 : 0);

boolean addDistrict(final /*@ non_null @*/ District a_district) {
  if (hasDistrict(a_district)) {
    return false;
  }
  my_districts[a_district.number()] = a_district;
  my_district_count++;
  return true;
}
```

Each district has a number that is used as its index in the array. The `hasDistrict` method returns true when the `my_districts` array contains a district with the same number and name as its argument. Thus, if the `a_district` argument has the same number but a *different* name than a district already in the array, the method will overwrite an existing district and increment the district count in violation of the invariant. The district count should only be updated only if there is no existing district at that index.

This violation might be classified as a specification error if the programmer forgot an invariant prohibiting two districts from having the same number but different names. The rest of the code does not appear to rely on such an invariant, however. Indeed, the `District.equals` method checks for equality by comparing not only the number but also the name.

(c) VoteSet.addVote [overspecification] This method suffers from overspecification in the form of a missing precondition. Note that it invokes the method `Candidate.incrementVoteCount`:

```
class VoteSet {

  final void addVote(final int a_candidate_code) throws IllegalArgumentException {
    if (!(my_vote_has_been_initialized && !my_vote_has_been_finalized)) {
      throw new IllegalArgumentException();
    }
    final Candidate candidate = my_candidate_list.getCandidate(a_candidate_code);
    candidate.incrementVoteCount();
    candidate.kiesLijst().incrementVoteCount();
  }
}

class Candidate {

  //@  requires my_vote_count < AuditLog.getDecryptNrOfVotes();
  //@  modifies my_vote_count;
  //@  ensures my_vote_count == \old(my_vote_count + 1); final
  int incrementVoteCount() { . . . }
}
```

As shown, `incrementVoteCount` has a precondition that the number of votes for the candidate be less than a preset number, but `addVote` does not ensure this condition. We believe the programmer erred in not including the inequality constraint in the precondition of `addVote`. It is also possible that the programmer intended `addVote` to be robust when the inequality is false, in which case we would re-classify this violation as a bug.

(d) KiesKring.init [underspecification] The post-condition of the `KiesKring` constructor invokes an underspecified `KiesKring.name` method:

```
//@ requires a_kieskring_name.length() <= KIESKRING_NAME_MAX_LENGTH;
//@ ensures number() == a_kieskring_number;
//@ ensures name().equals(a_kieskring_name);
private /*@ pure @*/ KiesKring(final byte a_kieskring_number,
                               final /*@ non_null @*/ String a_kieskring_name) {
  my_number = a_kieskring_number;
  my_name = a_kieskring_name;
}

//@ ensures \result.length() <= KIESKRING_NAME_MAX_LENGTH;
/*@ pure non_null @*/ String name() { return my_name; }
```

The specification of the constructor claims that calling `name()` in the post-state yields a string equal to the `a_kieskring_name` argument, and the constructor does indeed assign the argument to the `my_name` field. However, even though the implementation of `name` returns the `my_name` field, its specification says merely that it returns some string whose length is less than a fixed constant. Thus, the post-condition of `name` does not induce a strong enough axiom to establish the post-condition of `init`. Indeed, an implementation of `name` that always returns the empty string would satisfy its weak specification, but would clearly cause `init` to violate its own specification.

4 Discussion

Did bounded verification prove to be a useful technique? Prior to our case study, the KOA software had been the subject of rigorous development. The code, written according to a "verification-centric methodology" [5], had been checked with ESC/Java2 and unit-testing. Despite these prior efforts, our technique found that 19 of the 169 methods analyzed violate their specification.

The effort in conducting the study did not prove particularly burdensome. The KOA developers had already written the JML specifications, which was presumably a time-consuming task, but if full functional correctness is required, it seems unlikely that writing a full specification can be avoided. The remaining effort was choosing a bound on each analysis and inspecting the counterexample traces detected, both of which we found to be straightforward.

For most methods, we were pleased with the runtime and scalability of the analysis. For others, we were disappointed that their analysis in a scope of 5 did not complete within a reasonable amount of time. The poor performance of these methods demonstrates clear room for improvement in our technique and helps direct our future work on this project, as discussed below.

Does the case study lend support to the "small scope hypothesis"? That we found several errors in a small scope lends some support to the hypothesis that in practice many errors have small counterexamples. But it tells us nothing about proportions; it is possible that the software analyzed is riddled with bugs beyond the bounds we checked. Ideally, one would increase the scope step by step until all errors are revealed, and then determine their distribution. This is infeasible, but we were able to do this in miniature by increasing the scope until the analysis became intractable, and by noting (Table 3.1) the smallest scope in which a given error is revealed. That increasing the scope from 2 to 5 reveals no additional errors suggests that at least within that range the small scope hypothesis holds.

For many violations, detection required fields (array fields in particular) to be pre-populated. To detect the violation of `KiesKring.addDistrict` (Section 3.2b), for example, the array of districts needed to store a district that was distinct from the district given as an argument. Thus, finding the error requires a scope of at least 2 (in this case 2 districts).

We were somewhat surprised that the minimal bitwidth stayed as high as 3. After all, some methods didn't even use integers. As explained at the end of Section 3.1, the minimal bitwidth of 3 was due to length requirements on some array fields in the code. Even though a lower bound may not violate the precondition of the method under analysis, it still may prevent the existence of a valid pre-state heap configuration.

How does our technique compare to unit testing? On one level, it may seem surprising that our bounded verification technique revealed specification violations missed by unit testing. After all, bounded verification is conceptually a form of testing, in which all tests up to some small size are executed. However, two properties of our technique distinguish it from unit testing in key ways. One

important difference is that our technique is modular and, therefore, can detect problems due to underspecification of called methods, while unit testing cannot.

The major difference, however, is one of coverage. The voting code was subject to nearly 8,000 unit tests, which on its face sounds like a large number of tests to generate and run. Our technique, however, by leveraging SAT-solving technology, is capable of analyzing thousands, if not millions, of scenarios of every method individually.

Despite these differences, it still came as a surprise that unit testing did not detect the buggy `KiesLijst.compareTo` method discussed in Section 3.2a. Perhaps the static type of the `compareTo` parameter being `Object` (not `KiesLijst`) caused the testing tool to only feed the method arguments of *runtime* type `Object`. In these cases, the buggy version of the method would behave correctly by raising a `ClassCastException`.

To catch the bug in `hasDistrict`, discussed in Section 3.2b, a unit-test would need to first populate the pre-state with a non-empty array of districts, and then pair the pre-state with a district argument with the same number but a different name from an existing district in the array. Revealing such a bug requires a higher level of coverage than can be expected from traditional unit-testing.

How does our technique compare to ESC/Java2? It is difficult for us to determine why ESC/Java2 failed to detect the specification violations found in our study, and, unfortunately, the authors of the ESC study were unable to provide additional information in this regard. We know that 53% of the KOA methods did not verify successfully with ESC/Java2, so perhaps all 19 violations fell into this category. Or perhaps some of the 19 did verify but, due to unsoundness in ESC, were actually faulty. Kiniry, Morgan, and Denby [19] detail the sources of unsoundness and incompleteness in ESC/Java2.

Two examples where the unsoundness of ESC may have played a role are the analysis of KOA methods `KiesKring.clear` and `KiesLijst.clear`. ESC/Java2 examines only one unrolling of each loop, but our case study found that at least three unrollings were necessary to detect the overspecification of those methods.

What areas for future work appear worthwhile? There are a number of changes that we plan to make to Forge to improve its performance. One area for improvement highlighted by this study is the need to exploit generics in the Java source code. The presence of a Java `Map` in the code introduces a ternary relation whose second and third columns range over the universe of all objects, an expensive relation to encode in SAT. Generics were not used in KOA, but when provided, an improved translation to FIR could exploit the type arguments for the keys and values of a `Map`, significantly reducing the state space needed to represent the ternary relation and dramatically improving the scalability of the analysis.

Beyond technical enhancements, we see further opportunities for hybrid techniques that combine bounded verification and theorem proving in a way that leverages the advantages of each. For example, a tool could perform a bounded verification analysis in a progressively higher bound until some timeout is reached and then apply an automated prover like ESC. Or perhaps an automated prover

could use bounded verification to dispose of only the less tractable portions of the verification conditions, such as those involving alternating quantifiers.

Lastly, we will continue to support developers building translations from other high-level languages to FIR, to make them amenable to our analysis. An automatic translation from C is already in development by the System Engineering Lab at Toshiba, and we would greatly welcome similar efforts.

5 Related Work

In addition to the three tools discussed in this paper – ESC/Java2, jmlunit, and Forge – there are a number of other tools for checking Java code against JML specifications. Some, such as the KeY System [20], exploit theorem proving technology. Another promising tool is Kiasan [21], a new symbolic execution and test-case generation framework for Java and JML. It would be of great value to see these tools applied to the KOA code, learn what errors the find or fail to find, and understand the effort required to apply them.

This work extends our prior work [13], which itself builds on the work of Vaziri [9], whose Jalloy tool analyzed Java code via a translation to the Alloy modeling language. Dolby and Vaziri have since offered some optimizations to their technique [22] that might offer benefits to our own analysis.

The Forge Intermediate Representation is similar to other available programming languages and representations, first and foremost being the Alloy modeling language [11] and the subset of its logic that is accepted by the Kodkod model finder [14]. In a sense, one could view FIR as an "imperative Alloy."

DynAlloy [23] is an extension to Alloy which allows one to specify and check dynamic properties of relational models. Unlike FIR, Dynalloy is still a declarative representation, not an imperative programming language and offers a different approach to encoding Java in relational logic that Frias, et al, are pursuing.

FIR is not the first programming notation based on sets and relations. An early example is SETL [24], a high-level programming language founded on set theory and set operations. Another example is the Crocopat relational manipulation language [25]. Unlike these, FIR supports declarative constraints via specification statements in the code. Also, FIR supports object instantiation, whereas Crocopat presumes a finite, predefined universe of objects.

Less similar in semantics but more similar in purpose to FIR is BoogiePL [26], the intermediate programming language accepted by Boogie, a static verifier for Spec#. Like FIR, it offers a full specification language including quantifiers, but does not use relations as its fundamental datatype, so it would not be a convenient representation to encode in relational logic.

6 Acknowledgements

This research was funded in part by the National Science Foundation under grant 0541183 (Deep and Scalable Analysis of Software) and the Toshiba Corporate Research and Development Center. We would like to thank the Digital Security group at the Radboud University Nijmegen and KindSoftware at the University of Dublin for making KOA available.

References

1. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java". In: PLDI'02. (June 2002) 234–245
2. Gary Leavens, e.a.: Java Modeling Language http://www.jmlspecs.org.
3. Jacobs, B.: Counting votes with formal methods. In: 10th Int. Conf. on Algebraic Methodology and Software Tech. (AMAST'04), Stirling, Scotland (July 2004)
4. Fairmichael, F.: Full verification of the KOA tally system (March 2005) University College Dublin. Bachelor's Thesis.
5. Kiniry, J., Morkan, A., Cochran, D., Fairmichael, F., Chalin, P., Oostdijk, M., Hubbers, E.: The KOA remote voting system: A summary of work to date. In: Proceedings of Trustworthy Global Computing (TGC'06), Lucca, Italy (2006)
6. Kiniry, J.: Formally counting electronic votes (but still only trusting paper). In: ICECCS'07. (July 2007) 261–269
7. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: ECOOP'02, Malaga, Spain (June 2002) 231–255
8. Dennis, G., Yessenov, K.: Forge website http://sdg.csail.mit.edu/forge/.
9. Vaziri, M.: Finding Bugs in Software with a Constraint Solver. PhD thesis, MIT (February 2004)
10. Taghdiri, M.: Inferring Specifications to Detect Errors in Code. In: 19th Int. Conf. on Automated Software Engineering (ASE'04), Linz, Austria (September 2004)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge, MA (2006)
12. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the "Small Scope Hypothesis" (2003) MIT CSAIL Unpublished Manuscript.
13. Dennis, G., Chang, F.S.H., Jackson, D.: Modular verification of code with SAT. In: ISSTA'06, Portland, Maine (July 2006)
14. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS'07
15. Jackson, D.: Object models as heap invariants. Essays on Programming Methodology (2000) 247–268
16. Morgan, C.: The specification statement. In: ACM Trans. Program. Lang. Syst. Volume 10., New York, NY, USA, ACM Press (1988) 403–419
17. Morgan, C.: Programming from Specifications. Prentice Hall International (UK) Ltd., Hertfordshire, UK (1994)
18. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
19. Kiniry, J.R., Morkan, A.E., Denby, B.: Soundness and completeness warnings in ESC/Java2. In: SAVCBS'06, New York, NY, USA (2006) 19–24
20. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag (2007)
21. Deng, X., Lee, J., Robby: Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: ASE '06, Washington, DC, IEEE Computer Society (2006) 157–166
22. Dolby, J., Vaziri, M., Tip, F.: Finding bugs efficiently with a SAT solver. In: 15th Symp. on the Foundations of Software Engineering (FSE'07). (September 2007)
23. Frias, M., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: ICSE'05. (May 2005) 442–451
24. Dewer, R.: The SETL Programming Language. (1979)
25. Beyer, D.: Relational programming with Crocopat. In: Proceedings of the 28th International Conference on Software Engineering. (May 2006)
26. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft (2005)