Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 351 (2020) 51–73

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Safety of a Smart Classes-Used Regression Test Selection Algorithm

Susannah Mansky[1,2] Elsa L. Gunter[3]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA

Abstract

Regression Test Selection (RTS) algorithms select which tests to rerun on revised code, reducing the time required to check for newly introduced errors. An RTS algorithm is considered safe if and only if all deselected tests would have unchanged results. In this paper, we present a formal proof of safety of an RTS algorithm based on that used by Ekstazi [3], a Java library for regression testing. Ekstazi's algorithm adds print statements to JVM code in order to collect the names of classes used by a test during its execution on a program. When the program is changed, tests are only rerun if a class they used changed. The main insight in their algorithm is that not all uses of classes must be noted, as many necessarily require previous uses, such as when using an object previously created. The algorithm we formally define and prove safe here uses an instrumented semantics to collect touched classes in an even smaller set of locations. We identify problems with Ekstazi's current collection location set that make it not safe, then present a modified set that will make it equivalent to our safe set. The theorems given in this paper have been formalized in the theorem prover Isabelle over JinjaDCI [7], a semantics for a subset of Java and JVM including dynamic class initialization and static field and methods. We instrumented JinjaDCI's JVM semantics by giving a general definition for Collection Semantics, small-step semantics instrumented to collect information during execution. We also give a formal general definition of RTS algorithms, including a definition of safety.

Keywords: interactive theorem proving, regression test selection, small-step semantics, Java

1 Introduction

Testing is a crucial part of writing code. When writing programs it is important to run tests that demonstrate that the behavior of those programs is as expected and documented. When a program is modified, its tests are rerun to make sure changes have not introduced new bugs. This rerunning of tests is called regression testing.

[1] This material is based upon work supported in part by the National Science Foundation under Grants CCF-1439957 and CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We would like to thank Milos Gligoric and Darko Marinov for providing the motivating problem and for their insight into the importance of initialization timing. We are also grateful to the reviewers for their time, suggestions, and comments.

[2] Email: sjohnsn2@illinois.edu

[3] Email: egunter@illinois.edu

https://doi.org/10.1016/j.entcs.2020.08.004

In practice, a body of code can be large and have a huge number of tests written over it. Rerunning every single test can in some cases take hours or days, making it impractical to run them all after every small change. However, most changes will only even possibly affect a small number of tests. Thus, algorithms and methods have been developed to select which tests to rerun after code changes. The process of selecting which tests to run (alternatively, deselecting tests that should not be affected) is called Regression Test Selection (RTS). An RTS algorithm is *safe* if it only deselects tests whose previous results could be reproduced under the modified program.

In Java, all code is written inside methods in classes. Thus one approach to RTS is for Java test suites to determine which classes each test uses ("touches"), then only rerun a test if one of its touched classes has been changed. Ekstazi [3], a Java library for regression testing, uses this approach in its RTS algorithm. Touched classes can be derived in a number of different ways, but Ekstazi's algorithm collects this information dynamically via instrumentation of the code. The naive way to do this is to instrument every use of every class. However, Ekstazi uses a smarter approach, recognizing that many of these collections are likely redundant. For example, fetching a field of an object requires the object to have been created previously, meaning that a `new` has already been run on its class, so `getfield` does not need to trigger class collection. Their approach uses these insights to reduce the number of collection points.

In this paper, we present a formal proof of safety for an RTS algorithm similar to that used by Ekstazi. Our algorithm is designed for minimal collection. We then prove its safety and demonstrate how Ekstazi's algorithm can be modified to be equivalent and thus safe (and showing why the previous set was not safe). The definition of this algorithm is given via instrumentation of JinjaDCI's JVM semantics. These semantics are a model of a subset of the JVM written in the theorem prover Isabelle and are largely described in [7]. The instrumentation is created via a general definition we have given for what we call Collection Semantics that allows information-collection instrumentation of small-step semantics via the input of such semantics and a collection function.

We have also given a general definition of RTS algorithms that includes a definition of safety. Its axioms, including safety, further guarantee that repeated deslection of a test can safely be based on its output under a previous version of the program. [4]

Section 2 gives an overview of Regression Test Selection (RTS), Ekstazi, and JinjaDCI. Section 3 gives details of a few different class collection functions for use in an RTS algorithm using the classes-touched method to select tests. Section 4 describes Collection Semantics, our approach to instrumenting existing semantics with collection functions, and uses this approach to instrument JinjaDCI's JVM byte code with the class collection functions previously described. (These instrumentations are instantiations of the Collection Semantics definition.) Section 5

[4] This is an important feature for an RTS algorithm to have, as deselected tests are, by design, not rerun, so do not provide new outputs until they are reselected.

gives a general Isabelle definition for RTS algorithms, including a formal definition of safety, then combines this with the Collection Semantics definition to define collection-based RTS algorithms. The instantiations of Collection Semantics describing instrumented JinjaDCI JVM semantics are extended into collection-based RTS algorithm instantiations using this combined definition. Section 6 uses the general RTS definition and the instrumented semantics to give proofs of safety of using the defined class collection functions as a basis for classes-touched RTS algorithms. Finally, Sections 7 and 8 discuss some related work, recap, and suggest future directions.

The Isabelle development for the work presented here can be found online at `https://github.com/susannahej/rts`.

2 Background

We will first introduce the relevant topics and tools used in this paper.

2.1 RTS Algorithms and Ekstazi

In industry, many code bases are quite large, as are the test suites associated with them. These test suites can take on the order of days to run in full. Regression test selection (RTS) is the process of choosing which tests to rerun after changes have been made to a code base in order to decrease the time retesting takes. A test that is not run is said to be *deselected*. An RTS algorithm is called *safe* if it only deselects tests whose results would be unchanged.

Ekstazi [3] is a Java library for regression testing that employs an RTS algorithm at the level of JVM bytecode based on the classes that are used or referenced by each test. We call these the classes *touched* by a test. When a test is run, the names of the classes it touches during its execution are collected. Then, when changes are made to the code base, a test is only rerun if one or more of the classes it touched in its previous run have been modified. Thus if, for example, only a couple of modifications are made to non-core classes in a large code base, generally very few tests need to be rerun.

Ekstazi's algorithm attempts to collect the necessary classes as infrequently as possible (i.e., with as few collection triggers as possible), which leaves room for possibly under-collecting. To prove this algorithm's safety formally requires a semantics of the JVM that includes sufficient features to make the formal proof as convincing as possible. Static instructions and dynamic class initialization in particular are places where a class collection algorithm can be subtly incorrect. In Section 2.2 we will go into more detail about a semantics of the JVM that includes both of these features.

2.2 Isabelle and JinjaDCI

Isabelle allows us to model languages and programs written in them. It further allows the proof of many useful properties of those languages and programs, such

as how the information collected during the execution of a program with an instrumented semantics relates to that program.

Jinja [4] is a framework developed in the theorem prover Isabelle whose purpose is to give a formal semantics for a subset of both Java and JVM bytecode in a unified way. The authors wrote the semantics in Isabelle because this allowed them to write definitions and proofs based on their semantics. These proofs include type safety, equivalence of their big- and small-step Java semantics, and the correctness of a compiler from their Java semantics to their JVM semantics.

The two extensions of Jinja are JinjaThreads [6] and JinjaDCI [7]; the former extends Jinja with threads, while the latter extends it with static fields and methods and dynamic class initialization. The features of the latter are crucial uses of classes in Java, with behavior that differs from the other uses described by the original Jinja. For these reasons, these features are important to include in any proof of safety of Ekstazi-like touched-class-collection RTS algorithms like those discussed in this paper. On the other hand, threads should not create any complications with the safety of these algorithms as long as the synchronization of initialization is correct (as further justified in Section 6.4). Thus, we have chosen JinjaDCI as our semantic model for the proofs presented here.

The value of using a system like Isabelle to build a semantic framework is precisely the ability to prove these kinds of results, which is why we chose to use this framework: in order to prove that a test behaves the same on two different programs, we need both a definition of the language and a framework that allows meta-reasoning at this level.

In Jinja's JVM instruction names are capitalized and take appropriate arguments. For example, `new` is written as `New` and takes one argument: the name of the class to be instantiated, C; `getfield` is written as `Getfield` and takes two arguments: the name of the field whose value is being fetched, F, and the name of the class that defines that field in the object being passed, C.

2.2.1 Dynamic Class Initialization

In the JVM, class initialization methods are called dynamically. Rather than initializing classes up front, Java waits until the the class is actually used. When a class C is used, its initialization status is checked. If it is uninitialized, the class initialization procedure is run on it.

Initialization status is checked when a class or one of its subclasses is used. In particular, in the supported subset of JVM instructions, if a class C is not initialized, the class initialization procedure is called when:

- an instruction among `new`, `getstatic`, `putstatic`, or `invokestatic` references C,
- one of C's subclasses is being initialized, or
- at startup of the JVM, if C is the designated initial class.

In the full JVM, the procedure is also called at the invocation of certain bootstrap and reflective methods, and on certain interfaces when an instantiation is initialized.

The first step of the *class initialization procedure* [5] (called on a class C) is to check C's initialization status. If the status is `Error`, then an error is returned. Otherwise, if C needs to be initialized (and is not already being initialized), its superclass is checked and the procedure is run on it as necessary. Then, if that procedure returns normally, C's initialization method is run and its initialization status set to `Done`. If C's or any of its superclasses' methods throw an uncaught exception, then C's initialization status is set to `Error` and its initialization method is not run. [6] No class initialization methods are run until either a superclass is checked that is already initialized or the class `Object` (which has no superclass) is checked. At that point, the initialization methods for all the uninitialized superclasses found along the way are run from top to bottom, ending with C's.

Note how the class initialization *procedure* differs from a class's initialization method: the former is a general series of steps taken to determine the initialization status of a class and its superclasses, and which then calls the initialization methods of the appropriate classes. The latter is an actual method defined implicitly or explicitly by the definition of a class, and which is invoked in the course of the class initialization procedure.

Note that if a class has the initialization procedure called on it during the execution of a program (*initialization-called* classes), then that class was touched during that execution. (This set includes but is not necessarily limited to those classes actually initialized. A class may call the initialization procedure but fail to run its initialization method if one of its superclasses' initialization methods returns an error.)

Since the RTS algorithms described in this paper function by collecting the names of classes touched by a test (i.e., those classes whose definitions could affect the monitored run), the initialization-called classes are a good portion of the classes that will be collected. In particular, other than these, it turns out that the only other classes touched in a run are those classes that call static methods or fields defined by one of their superclasses. These latter can easily be collected at the time of such calls.

Collecting the initialization-called classes can be done either at the beginning of the initialization procedure or by collecting at points where this procedure might be called (i.e., at initialization checks). The former requires instrumenting the actual semantics of the JVM, because the initialization procedure is called dynamically rather than through instructions in the code. We have taken this approach in our algorithms. The latter is possible to do by adding instructions to the test code and is Ekstazi's approach.

[5] The procedure as described here is in brief and assumes a semantics without threads, as in Jinja and JinjaDCI. More precise details of this procedure are given in the JVM specification [5] and the JinjaDCI paper [7].

[6] The thrown exception is returned, and any further attempts to initialize this class will result in an error during the initialization status check as noted in the first step.

2.2.2 JinjaDCI's JVM

The class-collecting algorithms described in this paper are performed via an instrumentation of JinjaDCI's JVM semantics. The uninstrumented semantics are described in more detail in [7], but the following pieces are a brief summary of those most relevant to the instrumentation.

In JinjaDCI, as in Jinja, a program is a list of class definitions. A JinjaDCI JVM state σ is made up of a heap, a static heap (storing information about static fields and classes' initialization statuses), a frame stack (one frame per active method), and an optional exception. JinjaDCI JVM's execution function `exec` applied to a program P and a state σ will return a next state as long as σ has no exception and its frame stack is non-empty. A frame includes, among other things, an initialization call status flag indicating its current state relative to the initialization procedure. This flag can be built by one of the four constructors `Calling`, `Called`, `Throwing`, and `No_ics`. These are used as follows:

As described in Section 2.2.1, the procedure starts by creating a list of classes to be initialized including the triggering class C and its uninitialized superclasses. During this part of the procedure, the frame that called it has its initialization call status flag set to `Calling` C_l Cs, where Cs is the list uninitialized classes found so far (with C_l being the most recently added). Once this list is complete, the flag is set to `Called` $C_l \# Cs$ and the class initialization methods are run for the classes in $C_l \# Cs$ in order. If any of these methods throw an uncaught exception a, the flag is set to `Throwing` Cs' a, where Cs' is the list of classes whose methods were not run. If a frame is not in the middle of a class initialization procedure, the flag is `No_ics`.

3 Classes-Touched Collection Functions

In this section we describe the collection functions used by the RTS algorithms whose safety we consider in Section 6. All of these algorithms collect the names of classes touched over the course of a test's execution on a program. Then, when the program is changed, tests whose touched classes did not change are deselected. We have used two different general approaches to this kind of collection. The gist of each is as follows:

- *Naive approach*: At each step of execution, collect every class that might affect that step.
- *Smarter approach*: At each step of execution, collect only those classes that might affect that step that – by virtue of the step type – cannot be safely assumed to be collected during some other step. (That is, use knowledge of contextual guarantees to collect classes less frequently.)

An algorithm using a naive collection function is easier to prove safe, as it is safe over each step of execution. Once its safety is proved, it can be used to prove the safety of a correctly-defined smart collection function by showing it collects the same classes. For this reason, we will give instances of both, using proof of safety of the first to prove safety of the second.

3.1 A Naive Algorithm

The naive approach is to at each step collect every class that might change the behavior of the step. If an object is used, then the algorithm collects the name of the object's class, and all its superclasses. For example, if a static method is called, it collects the referenced class, and all its superclasses. If done sufficiently thoroughly, it is easy to see why this approach collects enough classes to form the basis of a safe deselection algorithm. Since for each step the classes that could have changed that step haven't changed, the behavior of each step is the same.

Below we give the details of the algorithm. When a class is collected, its superclasses are also collected. The collection goes as follows:

(i) At each step of execution, collect the error classes and the current class of each frame in the frame stack.

(ii) Additionally, collect based on the initialization call status of the current frame:
   (a) `Calling` C Cs: collect C
   (b) `Called` (C#Cs): collect C and the class defining C's `clinit` method
   (c) `Throwing` Cs a: collect the class of the object found at address a on the heap
   (d) `No_ics`: collect based on the current instruction:
      - `New` C, `Getstatic` C F D, `Putstatic` C F D, or `Invokestatic` C M n [7]: collect C
      - `Getfield` F C, `Putfield` F C, `Checkcast` C, `Invoke` M n, or `Throw` [8]: collect the class of the calling or thrown object, if properly provided; [9] otherwise, collect nothing
      - For any other instruction, collect nothing

Note that this algorithm does not depend directly on the behavior of the JVM. While its behavior runs in parallel to the semantics of the JVM described in Section 2.2.2, the collection at each step is only dependent on that step's initial state. This will be important in Section 4, which describes how this semantics-independent algorithm can be combined smoothly with a small step semantics in order to give an instrumented semantics.

However, while this approach collects the minimal set of touched classes, it also does so by collecting the same classes over and over again unnecessarily. This insight leads to a smarter approach.

[7] `New`'s argument is as described in Section 2.2. `Getstatic`'s and `Putstatic`'s arguments are the calling class, C, the field being referenced, F, and the class defining the field, D. `Invokestatic`'s arguments are the calling class, C, the method being invoked, M, and the number of arguments to the method, n.

[8] `Getfield`'s and `Putfield`'s arguments are as described for `Getstatic` in Section 2.2. `Checkcast`'s argument is the class being cast to, C. `Invoke`'s arguments are the method being invoked, M, and the number of arguments to the method, n.

[9] In the JVM, these instructions all expect an address to an appropriate object to exist in a designated location on the stack. If the expected object is not present, an error is thrown. Note that if the object is missing, then no classes are actually touched or affect the outcome of the instruction besides the error's class.

3.2 A Smarter Algorithm

The smarter approach recognizes a few things about correct programs:

(i) If an object is used, then that means that the object was created by a `new` instruction.

(ii) Instructions touching classes directly (`new` and the static instructions) will necessarily result in that class (and its superclasses) being initialized before the instruction is resolved.

(iii) The current class of any frame will be either the initial class or the defining class of that frame's current method; in the former case the class was initialized at the beginning of the current test's execution; [10] in the latter case it will have been touched during the method call that created the frame.

(iv) If the initialization call status of a frame is `Called` ($C\#Cs$), then at some point it was `Calling` C Cs.

From these observations it follows that many uses of a class by instructions can actually guarantee that the class has been used previously, and as such has already been collected. By not collecting at any point that can make this guarantee, the number of places where a class must actually be collected is significantly reduced. The result will be collecting the same set of classes, but each will be collected many fewer times, meaning less added overhead. The modified collection approach is as follows:

- Collect a class when the class initialization procedure is called on it (that is, a frame has initialization call status flag `Calling` C Cs for some Cs)
- On a `getstatic`, `putstatic`, or `invokestatic` instruction:
  - If the field/method does not exist, collect the referenced class and all its superclasses
  - If the field/method does exist, collect all classes between the referenced class and the declaring class (including the referenced class but not the declaring class)
- Collect the names of error classes and their superclasses

Note that when the class initialization procedure is called on a class C, C is collected at the beginning of the procedure, not C's class initialization method. (Recall the difference between the class initialization procedure and a class's initialization method from Section 2.2.1: the former is the steps followed by the JVM that leads to running a class's initialization method and that of its superclasses, whereas the latter is an actual method of a class.) This is necessary because during the procedure, C's superclass is initialized first (if it has not been already); if the superclass's initialization method fails, C's is never run. However, even in that case C must be collected because a change to C could include changing the name of its superclass.

[10] In Java, to run a program (or test) is actually to run a static method of a class. This class is the initial class, which is initialized before its static method is run.

This algorithm is constructed by not collecting anywhere a class is guaranteed to have been collected by a previous use. Furthermore, each of these previous uses is either still a collection point or is covered by its own previous use that is still a collection point. Thus it can be seen that the above collects the same classes as the naive algorithm does. Therefore, as long as the naive algorithm is safe, this smarter algorithm is as well. We will formally prove these observations in Section 6. However, some informal reasoning follows, touching on each place where the naive algorithm collected classes.

First, the collection of error classes only happens once in this approach instead of during every step. The naive algorithm only needed to collect these classes at every step because it was designed in a way that made every individual step clearly safe by itself. For this algorithm, proving safety necessarily involves confirming that classes were collected at some point during execution, so collecting once at the start is sufficient. In practice, these classes would not necessarily need to be collected even then, as they would be initialized. Collecting these classes up-front is only necessary here as an artifact of the way that Jinja handles the error classes (by instantiating them up front).

Second, as noted in the above list of observations, each frame's current class is the class that declared the method whose execution is being handled by that frame. Other than the initial frame, each frame is created by an `Invoke` or `Invokestatic` instruction. In either case, the class declaring the invoked method is collected by other cases. The initial frame's class is the initial class; this class is initialized before the initial method is executed, and so is collected at that point.

Third, the `Called` and `Throwing` initialization call statuses do not need to collect anything because the former is covered by collection at `Calling` (which is guaranteed to have been the init call status in a previous step of execution), and the latter is covered by its use of an existing object (created by a `new` instruction).

Finally, when a static instruction runs, the existence of the field or method being used is checked before the initialization status of the declaring class. Thus if the field or method cannot be found ("does not exist"), all the classes that might affect that existence must be collected, since no class initialization procedure will be called. If the field or method is found, then the initialization procedure is called on the class that declares it. The declaring class will be collected at the beginning of the procedure as per the previous collection rule, as will all its superclasses (either by being initialized during this procedure call, or during whatever procedure call initialized them previously). Thus, of the referenced class and its superclasses, only the classes between the referenced and declaring classes will not be collected via the procedure, and so must be collected at this point. `New` C always results in the initialization of C and its superclasses, so it does not need to collect anything.

No other instructions need to collect any classes at all. This follows from the observations given at the beginning of this section, as all the other collecting instructions use an object created by a `new` instruction (which in turn is preceded by initialization of the relevant classes).

To sum up, if the name of a class is collected at the beginning of its class

initialization procedure, then the only other times it needs to be collected are when it is able to change behavior but is not initialized or checked for initialization status. The only places where this can occur are those where a class is referred to in order to access a field or method declared by one of its superclasses.

3.2.1 Ekstazi's Algorithm

In Ekstazi's algorithm, as in the smarter algorithm, only certain uses trigger collection, relying on the fact that some uses are necessarily preceded by another use of the class. However, unlike in the algorithms described above, which are performed via instrumentation of the semantics of the JVM, Ekstazi's algorithm adds print statements to the code just before class uses. The smarter algorithm partially works because the semantics is being instrumented rather than the code, meaning it is possible to instrument the class initialization procedure to collect classes in a way code instrumentation does not allow (as calls to the class initialization procedure occur dynamically during runtime rather than through instructions in the code). Thus Ekstazi's collection function requires a few adjustments from ours in order to achieve the same effect. These adjustments amount to replacing collecting at the beginning of initialization with collecting just before any place where the initialization procedure might be run.

Even assuming instrumentation of the semantics rather than the code, in Ekstazi multiple tests may be run on the same JVM, meaning that some classes may already be initialized at the beginning of any particular test. In such cases, it would be necessary to collect in all places where initialization could be called regardless, as the actual calls would only occur on classes not already initialized. Running multiple tests in the same JVM does introduce the problem of state pollution: the values of static fields may be changed by one test and used by another, potentially affecting the latter's result. We do not consider the effect of state pollution in this paper.

Also worth noting are the two places Ekstazi collects that the Jinja approach cannot due to incompleteness of semantics: reflection invocations and interface invocation. (Neither are modeled in Jinja or its extensions.) According to the JVM specification, the class initialization procedure is called upon "invocation of certain reflective methods." Thus it is unnecessary to collect at these invocations if collection occurs at the beginning of the class initialization procedure (as in the smarter algorithm), and necessary to collect there otherwise (as in Ekstazi). As for interfaces, the `invokeinterface` instruction does not result in calling the initialization procedure, so it is necessary to instrument this instruction in a context with interfaces regardless of semantic or code instrumentation. We leave this addition to future work. [11]

In summary, this is what Ekstazi's algorithm should do:

(i) Collect the classes touched by the following instructions via print statements

[11] As interfaces act a great deal like classes that cannot be instantiated, including that they are initialized in the way classes are, we do not believe the addition of interfaces would create any problems with the proofs presented here if handled in the same way. That is, in the smarter algorithm's collection, an interface's name would be collected when the initialization procedure is called on it. In Ekstazi's algorithm, it would be collected before the invocation of interface methods and when a class extending it is collected.

```
//test class
class T{
  public static void main(...){
    //static method call; calls initialization procedure on declaring class C
    C.M();
  }
}

class D{
  static { throw e; } //class initialization method
}

class D'{
  //class initialization method is empty by default
}

//ORIGINAL DEFINITION FOR CLASS C
 //when initialization procedure is called on C, init procedure is called on D prior to
  running C's init method; D's init method throws an error, so C's init method never runs
class C extends D{
  static void M(){ }
}

//CHANGED DEFINITION FOR CLASS C
 //direct superclass is updated; init procedure will now be called on D' instead of D, so
  test completes without error
class C extends D'{
  static void M(){ }
}
```

Fig. 1. Example of code where Ekstazi does not collect enough

added prior to them:

- `new`, `getstatic`, `putstatic`, `invokestatic`, `invokeinterface` instructions
- Invocations of reflective methods

(ii) Collect all error classes (or rather, treat them all as touched classes for all tests)

In Section 6.3, we will demonstrate the safety of this approach.

Ekstazi's actual collection function (as described in [3]) differs mainly in that, instead of collecting before `new` and `invokestatic` instructions or the initialization procedure, it collects at the beginning of class initialization methods, constructors, and static methods. These collections are too late, making it not safe. For example, consider the code given in Figure 1. When the test is run with the original code for C, Ekstazi will collect class D, as its initialization method is run, but not C, as the error thrown by D's initialization method ends the program before C's initialization method or static method M are run. Our smart algorithm, on the other hand, will collect C when its initialization procedure is called. The test fails under the original code and succeeds under the changed code, so it should be run again. However, the only change is to class C, meaning Ekstazi would not rerun the test.

4 Collection Semantics

The addition of information collection like that of the class-collecting algorithms described in Section 3 can be modeled by instrumenting a semantics of the relevant language by adding the collection of relevant information on top of the step's normal behavior.

One approach to this is to directly modify an existing semantics of the relevant language, adding an extra piece to the state that keeps track of the information

being collected. However, this approach results in a semantics with no immediate proof of mathematical equivalence in behavior to the original semantics. Proof of this equivalence is required in order to use its consequences. As consequences include a guarantee that results proven over the instrumented semantics hold over the original, proven equivalence is essential to the usefulness of the new semantics. Further, the instrumented semantics would need to be kept equivalent manually.

A better approach is to create a function that takes a semantics and a collection function and produces an instrumented semantics, which we will refer to as a Collection Semantics. This allows a general theorem about the function showing behavioral equivalence between the original and instrumented semantics. Then on any input, this equivalence would be immediate, and any results proved on the former would be instantly applicable to the latter. Also, any changes to the original semantics would be reflected in the instrumented semantics without any extra effort.

Such a function is best defined in Isabelle by using a locale, a way to define a collection of components with a set of axioms on those components. This definition can then be instantiated, giving instances access to any theory developed from the axioms. Further details about locales are given in Section 4.3.

Once instantiated with the naive and smart algorithms given in Section 3, the behavior of the semantics produced can be evaluated and compared, allowing us to prove their safety as a mechanism of test deselection.

We define the `CollectionSemantics` locale by first defining the `Semantics` locale as a base.

4.1 `Semantics` Locale

We first give a general definition for a semantics.

Definition 4.1 A ***Semantics*** is a pair:

- a small-step function `small` that takes a program and a state and returns a set of next states, and
- a set of end states, `endset`,

which fulfills the axiom `endset_final`: $\forall \sigma \in \texttt{endset}.\ \forall P.\ \texttt{small}\ P\ \sigma = \{\}$.

Note that this definition specifies small-step style semantics. This allows the collection function to be more semantics-agnostic: it will only need to collect based on a state assuming a single step.

Given a `Semantics`, a big-step semantics function `big` is derived from `small` using `endset`: `big` just applies `small` to the input until a state in `endset` is reached, then returns that end state.

4.1.1 Running Example: `Semantics` Instance

The semantics locale can be instantiated with the JVM `exec` function as `small`, with `endset` as the set of states that have empty frame stacks or an exception flag.

Recall from Section 2.2.2 that `exec` returns no next state on any of the states in this set, satisfying the `Semantics` axiom `endset_final`.

4.2 `CollectionSemantics` Locale

Given the `Semantics` definition, it is then possible to extend to a `CollectionSemantics`.

Definition 4.2 A ***CollectionSemantics*** is a Semantics paired with a three-tuple:

- a collection function `collect` that takes a program and two states (before and after), and returns a collection,
- a function `combine` for combining collections that takes two collections and returns another, and
- an identity for the combining function, `collect_id`,

where `combine` is associative and `collect_id` acts as both a left- and right-identity under `combine`.

In the above, a "collection" can by anything from a set to an integer to a file.

The pieces `small` and `collect` of a `CollectionSemantics` are used to define a small-step instrumented semantics `csmall`, then extended with `endset` to an instrumented big-step semantics `cbig`. The former simply returns a set of pairs of results returned by applying `small` to the input, then applying `collect` to the input and output. The latter returns the result of applying `csmall` to the input as many times as it takes to reach an end state, using `combine` to combine the information collected across the steps. Note that the resulting collection is the identity `collect_id` if no steps are taken. As the states returned by `csmall` are the same as those returned by `small`, the states returned by `cbig` are also the same as those returned by `big`. Then any proven instance of the definition will immediately be able to use both the derived `cbig` and the result that its output is the same as the derived `big`.

4.2.1 Running Example: `CollectionSemantics` Instances

The instance of `Semantics` given in Section 4.1.1 can be extended to instances of `CollectionSemantics` with the naive and smart class collection functions described in Section 3. Since these functions return sets of classes, the components `combine` and `collect_id` are the set union operator and the empty set, respectively. It is easy to see that the axioms of associativity and left- and right-identity hold.

4.3 Using Locales

In Isabelle, definitions comprised of a collection of fixed items ("components") together with a set of axioms can be given using a `locale`. Once these components and axioms are given, theory can be developed that relies on them, including definitions and lemmas. One might write a definition depending on the components and then prove things about that definition given the axioms. This approach was

used to turn the above definitions of `Semantics` and `CollectionSemantics` into a locale, as well as those definitions given in Section 5.

Locales can be instantiated by giving concrete definitions of the correct types to match its components, followed by a proof that this instance of the components meets the requirements mandated by the axioms. Once this has been done, the locale's theory can be used, including any derived definitions it contains and any theory developed about them, in addition to any lemmas proved to follow from the axioms.

5 Formally Defining Regression Test Selection

We will now give a formal, general definition for RTS algorithms. We will then combine this definition with `CollectionSemantics` from Section 4 to get a definition for a collection-based RTS algorithm.

5.1 `RTS_safe` Locale

The following defines a general regression test selection algorithm that is safe.

Definition 5.1 An ***RTS_safe*** is a five-tuple:

- set of valid programs `progs`,
- set of valid tests `tests`,
- output function `out` that takes a program and test and returns a set of program outputs,
- equivalence relation `equiv_out` over pairs of program outputs, and
- deselection relation `deselect` taking an initial program, program output, and altered program,

which fulfills the following axioms:

- `existence_safe`: for all $P, P', t, o1$, if $P, P' \in$ `progs`, $t \in$ `tests`, $o1 \in$ `out` $P\ t$, and `deselect` $P\ o1\ P'$, then $\exists o2 \in$ `out` $P'\ t$. `equiv_out` $o1\ o2$,
- `equiv_out_equiv`: `equiv_out` is an equivalence relation, and
- `equiv_out_deselect`: if `equiv_out` $o1\ o2$ and `deselect` $P\ o1\ P'$, then `deselect` $P\ o2\ P'$.

The sets of valid programs and tests give a scope to the safety axiom: the algorithm is only required to be safe over these given sets.[12] The output function provides some sort of output given a program and test, such as the output of a semantics for the language, as used to run tests with programs. The equivalence relation over outputs gives a way to directly compare outputs to determine whether they count as sufficiently similar results in the context of safety. The deselection

[12] Note that this is desirable rather than scope-reducing because the only programs and tests that are relevant are those that can be run – those that meet well-formedness conditions that would generally be enforced by compilers.

relation is the meat of the algorithm, choosing which tests not to run based on a pair of programs, plus an output. As given in the safety axiom, these would be instantiated with the original program, the new program, and the output of running a test over the original program. Then deselection would be applied to the test that produced the given output. This function takes a test output instead of a test because deselection will be based on the achieved output, as there may be more than one.

The safety property we use here is one we call *existence safety*. This version of safety is designed with non-deterministic semantics in mind: if a test may produce more than one outcome, it only guarantees that if the original output of a test resulted in its deselection, then there is at least one equivalent outcome under the changed program. Under this definition, if a flaky test (i.e., one that can produce both a passing and failing outcome under the same program) is deselected, this axiom guarantees it will remain flaky under the changed program. We have chosen this definition of safety because the algorithms we describe here are not designed to identify flaky tests to rerun. Thus, this is the kind of safety promise that is expected and desired here.

The two axioms other than safety require that `equiv_out` is in fact an equivalence relation over outputs and is fine-grained enough that equivalent outputs are indistinguishable to the `deselect` function. When combined, these axioms are sufficient to prove the following:

Lemma 5.2 Safety Transitivity: *If a non-empty sequence of programs Ps are all in `progs`, test t is in `tests`, o_0 is an output under the first program in Ps and t, and o_0 is deselected under each sequential pair of programs in the sequence, then there is an output under the final program in Ps and t that is equivalent to o_0.*

This lemma is a guarantee that after a deselection based on a given output, it is safe to continue using that output for future deselection decisions until the test is selected again. This is important because the intention of an RTS algorithm is to not run a deselected test again until it is selected, meaning that there will be no updated output to use until this occurs.

As described in Section 4.3, `RTS_safe` can be turned into a locale.

5.2 `CollectionBasedRTS` Locale

Having defined `CollectionSemantics` (Section 4.2) and `RTS_safe` (Section 5.1), we are able to define the combination of the two, `CollectionBasedRTS`. This gives the general form of an RTS algorithm that uses information collected during execution to make selection decisions.

Definition 5.3 A ***CollectionBasedRTS*** is a triple of a CollectionSemantics; an RTS_safe whose `out` returns a set of state-collection pairs; and the pair:

- a function `make_test_prog` that takes a program and a test and returns a modified program that includes the ability to run the test, and
- a function `collect_start` that takes a program and returns a starting collection,

which fulfills the axiom `out_cbig`: $\forall P\ t.\ \exists \sigma.$ `out` $P\ t = \{(\sigma', coll') \mid \exists coll.\ (\sigma', coll) \in$ `cbig` (`make_test_prog` $P\ t$) $\sigma\ \wedge\ coll' =$ `combine` $coll$ (`collect_start` P)$\}$.

While `CollectionSemantics`'s `cbig` takes a program and a state, `RTS_safe`'s output function takes a program and a test as inputs. The former is a general execution function allowed to start at any point in execution. The latter just runs tests, deriving a start state for execution from the given test and program. Therefore the latter can be seen as a specific instance of the former. `CollectionBasedRTS`'s components and axioms are defined around formalizing this idea.

The function `make_test_prog` takes a program and a test as might be given to `out` and returns a program for input into `cbig`. The function `collect_start` returns a collection for each program representing the information that should be collected about it up-front. This represents any information that the RTS algorithm takes into account on the basis of the program itself, and which the `out` function will include automatically.

The axiom formalizes the above expectations of the relationships between `out`, `cbig`, `make_start_prog`, and `collect_start`. For every program-test pairing, there exists a state σ such that the outputs of `out` and `cbig` are equal if the program's starting collection is added to the latter's collection outputs. The state σ is functionally the state that the `out` function runs from on the given program and test.

5.2.1 Running Example: `CollectionBasedRTS` Instances

The instances of `CollectionSemantics` given in Section 4.2.1 can be extended to instances of `CollectionBasedRTS` with the following additional instanstiations:

`make_test_prog` is a function that takes a program (a list of class definitions) and a test (a class definition) and adds the test class to the beginning of the program's list to make a new program.[13]

`collect_start` always returns the empty set in the naive case; since each step collects everything for that step, nothing is collected up-front. In the smart case, `collect_start` collects the exception classes and their superclasses.

`out` takes a program and a test, and applies `cbig` (derived from the `CollectionSemantcs` being extended) to the program returned by `make_test_prog` on the given program and test and the starting state dictated by the same.[14] By design, this function meets `CollectionBasedRTS`'s required relationship with `cbig`.

`deselect` takes a JVM program; a (JVM state)-(class collection) pair; and a second program, and returns `True` if the classes in the collection have not changed from the first program to the second,[15] `False` otherwise.

[13] It also creates a class definition for a `Start` class whose superclass is `Object` and has two methods: a class initialization method that does nothing, and a `main` method that calls the test class's `main` method. This class simplifies modeling the calling of the class initialization procedure on the test class, which is the true initial class, by creating what is essentially a "nothing" frame from which the procedure can be called and to which it can return for the completion of the call to the test's `main`.

[14] Given a program, the start state starts with a starting heap (which has starting instances of the error classes), a starting frame stack (with a single frame whose class and method are `Start` and `main`, with program counter 0 and initialization call status `No_ics`), and the starting static heap (that simply sets `Start`'s initialization state flag to `Done`).

[15] A class is considered changed is anything inside it is different, or if it exists in one program and not the

`equiv_out` is defined as equality between (JVM state)-(class collection) pairs. This definition clearly meets `RTS_safe`'s axioms as an equivalence relation where equivalence outputs are indistinguishable by `deselect`.

`progs` is the set of JVM programs that are well-formed, do not already contain a `Test` or `Start` class,[16] and whose `Object` class's `main` method - if it exists - is static, takes no arguments, and returns type `void`.

`tests` is the set of class definitions whose name is `Test`, create well-formed programs when combined with any of the programs in the above described set, and have a `main` method that is static, takes no arguments, and returns type `void`.

For both instantiations, `RTS_safe`'s existence safety is the only axiom that is not immediate. Proofs of this axiom for both the naive and the smart instance will be presented in Section 6.

6 RTS Safety Proofs

Below we will describe the steps taken in formally proving safety of the naive and small collection-based RTS algorithms. All lemmas and theorems stated in this section are in the context of JinjaDCI's JVM semantics.

6.1 Safety of Naive Algorithm

Proving the safety of the naive collection-based deselection algorithm boils down to showing that for each kind of step of execution, the classes collected by that step are the only classes that could affect its behavior. In other words, if those classes are unchanged in a changed program, the behavior of the step remains the same.

Lemma 6.1 Naive Single-Step Safety*: If the classes collected by the naive collection function over the single step of JVM execution under program P from valid state σ do not differ between programs P and P', then the single step of execution under P' from state σ yields the same state and collection.*

Lemma 6.1 can then be extended from one step to many. From this and the validity of the start state, it is straightforward to show that the end state reached from the start state will be the same under any two programs that agree on the classes collected over the full execution.

Theorem 6.2 Naive Safety*: If P and P' are well-formed JVM programs, t is a valid test class, (σ, Cs) is an output of the naive collection JVM semantics under P and t, and the classes in Cs do not change from P to P', then (σ, Cs) is an output of the naive collection JVM semantics under P' and t. Thus deselecting t on this basis is safe.*

Well-formed JVM programs and valid test classes are as defined for `progs` and `tests` in the naive instantiation of the `CollectionBasedRTS` locale. Note that when

other.

[16] This requirement is an artifact of Jinja's JVM requiring class names to be unique; in practice this is not restricting.

the classes in the collection Cs are unchanged from P to P', that is when the naive algorithm will deselect t. Thus the stated safety of t's deselection in this case is also safety of the naive collection as a method for deselection.

6.2 Safety of Smarter Collection

The approach to proving the safety of the smarter collection-based deselection algorithm is necessarily less direct than that for the naive approach. By design, most of the classes that could affect a given step of execution are not collected at that step, relying instead on being collected by either an earlier or later step in execution. For each step of execution, represented by the current state at that step, the classes collected at earlier or later steps can be grouped into "backward promises" - classes collected prior to the step - and "forward promises" - classes that will be collected in future steps, if they have not already been collected. Which classes are in each promised set are determined by the state in that step of execution.

The *backward-promised* classes relative to a state are the initialized classes (as marked on the static heap), classes of objects in the heap, current classes from the frame stack (i.e., those declaring the methods that are currently mid-execution), classes of system exceptions, and superclasses of all the above. The *forward-promised* classes depend on the current initialization call status. If it is `Calling` C Cs, then C (i.e. the class whose initialization is actively being called by the current frame) and its superclasses are promised. If it is `No_ics` or `Called` [], then the class whose initialization is checked by the current instruction's execution (if the instruction is a `new` or static instruction) and its superclasses are promised.

The backward-promised classes are designed to cover those classes that are known to have been collected based on information currently present in the state. These classes are most of those we had previously observed could be counted on having been previously collected: classes that have already been initialized, classes that have been instantiated, and so on. These promises, once proved, allow proof that instructions that, for example, use an initialized class or an existing object on the heap, do not have to collect those classes.

The forward-promised classes are those that are about to be collected during an initialization procedure, if they have not been already. This promise, once proved, allows proof that steps in the middle of the initialization procedure do not have to already have collected the superclasses of the class currently being initialized.

Together, the promises are designed so that in order to show that smart collection collects at least the classes collected by naive collection, it is sufficient to show that these promises are kept. First, we show that the forward-promised classes not covered by backward promises are collected:

Lemma 6.3 Forward Promises Kept*: If* `Object` *is a superclass of* C *in program* P*,* σ *is a non-end state whose top frame has initialization call status* `ics`*, and:*

- `ics` *is* `Calling` C Cs*, or*
- `ics` *is* `No_ics` *and the current instruction of the top frame of* σ *is* `New` C*,* `Getstatic` D F C*,* `Putstatic` D F C*, or* `Invokestatic` D M n*, where* F

or M (as applicable) exists, is static, and is seen by D in C

then all classes of C and its superclasses that are uninitialized on σ's static heap are collected by the smart collection algorithm by the end of complete execution from σ.

Note that Lemma 6.3 only promises collection of classes that are uninitialized on the static heap. This is because all initialized classes are guaranteed collected by the backward promise about classes on the static heap. Additionally, the case where the current initialization call status is `Called` [] is not covered because state conformity [17] guarantees that the class whose initialization was called is initialized on the heap, meaning that the backward promise for classes on the static heap is sufficient for the promise to be kept. The requirements for existence of the relevant static method or field are there because initialization will only occur if these checks pass. (Naive collection also skips these classes if the exists or static checks fail.) Finally, note that the classes promised collected by this lemma can only be assumed collected when execution terminates, as it only guarantees the classes collected by the end of execution.

The two pieces of Lemma 6.3 are proved separately: first the `Calling` case is proved by induction over the steps of execution. The other case is then proved for each relevant instruction type, using the first case and that the next execution step after each will be to set the current initialization call status to `Calling` C [].

Unlike the forward promises, the backward promises are a preservation property. That is, for each step of execution, if the backward promises have been kept up to that point, then that step will preserve those promises. More precisely, since backward promises are entirely relative to the current state, if the backward promises are assumed kept at a state σ via collection Cs, then if execution of that state yields state σ' and some collection Cs', then the backward promises are kept for state σ' by the combined collection $Cs \cup Cs'$. So if, for example, a step of execution adds an object to the heap - increasing the scope of the promise that heap object's classes are collected - the class of that object is either collected by that step (and is in Cs') or is already covered by σ's backwards promises (and is in Cs). Either way, it will be in $Cs \cup Cs'$.

This preservation of the backward promises is formally stated in the first half of Lemma 6.4 below. That fact, together with Lemma 6.3's guarantees about forward promises being kept, can be used to prove that the smart collection algorithm collects at least those classes collected by the naive collection algorithm (the second half of Lemma 6.4):

Lemma 6.4 *If P is a well-formed JVM program under the typing Φ,* [18] *the state σ is fully conforming under P and Φ, and the set of classes Cs contains all classes*

[17] Our proof of equivalence between smart and naive collection, and thus safety of smart collection, assumes a well-formed program. We have further shown that well-formed programs produce conforming states (and execution preserves state conformance, so it can be safely assumed here). For a complete definition of state conformance, `correct_state`, see JinjaDCI's `BVConform` theory.

[18] Φ is a function that returns the expected types for the stack and local variables for each instruction of each method of each class in a program. It is used by Jinja and its extensions to encode expected types in a way that allows proof of type safety of their JVM byte code semantics.

described by the backward and forward promises over σ plus the classes collected by the smart algorithm on the single execution step under P from σ to σ', then:

(i) Backward Promise Preservation*: The classes described by the backward promises over σ' are in Cs*

(ii) Naive $\subseteq$ Smart*: Since backward and forward promises are kept, all the classes collected by the naive algorithm are in Cs*

Further, by definition inspection, the naive algorithm collects at least all the classes collected by the smart algorithm (Smart $\subseteq$ Naive). Thus, together with Lemma 6.4 ii, the smart and naive algorithms collect the exact same set of classes during the execution-to-termination of a well-formed JVM program starting from a state whose backward promises are met by the starting collection set. (The start state's backward promises must be met up-front so that backward promise preservation can kick in.) Since the smart algorithm's starting collection set is designed to meet the backward promises of its start states, we get the following:

Lemma 6.5 Naive = Smart*: If P is a well-formed JVM program and t is a valid test class, then the set of classes collected by running the naive-instrumented JVM semantics over P and t is equal to the set of classes collected by running the smart-instrumented JVM semantics over P and t.*

Therefore, since the naive approach is safe (Theorem 6.2), the smart approach is as well.

Theorem 6.6 Smart Algorithm Safe*: If P and P' are well-formed JVM programs, t is a valid test class, (σ, Cs) is an output of the smart collection JVM semantics under P and t, and the classes in Cs do not change from P to P', then (σ, Cs) is an output of the naive collection JVM semantics under P' and t. Thus deselecting t on this basis is safe.*

6.3 Making Ekstazi Safe

In Section 3.2.1, we described the differences between the smart collection algorithm and that used by Ekstazi and presented a modified set of collection points for the latter. In order to achieve the safety guaranteed by the above proofs, Ekstazi must at least collect in places that cover what we have outlined here. In particular, since Ekstazi cannot collect directly during the class initialization procedure, it instead collects at each instance where the procedure will be called - in advance of the call. After the call would be too late, as class initialization does not return to the calling instruction if it fails. Since this is what our modified set does, an algorithm using this modified collection function is safe.

6.4 A Note About Threads

The proofs of safety of the algorithms described here are over a semantics that does not include threads. However, class initialization is key to class collection in the smart algorithm and the class initialization procedure uses locks to ensure that

classes are not used until they are fully initialized. Thus even though a semantics including threads would be more complicated, if the locking mechanism correctly prevents class use prior to initialization, proof that the collection algorithms given would be safe would follow in a very similar fashion.

7 Related Work

Over the years there have been many approaches and algorithm proposed and implemented for the purposes of regression test selection.(A recent survey of techniques is given in [1].) A number of these techniques are regarded as safe, but with only informal arguments. Formal proof is quite often skipped because it requires a formal model of the language in addition to efforts like presented here. For some cases this can be good enough, as a large number of uses eventually uncover most errors in reasoning. Further, while safety can be desirable, the time trade-off of near-safety can be good enough if the full test suite is occasionally run. Other important features of an RTS technique are precision (minimal tests run with unchanged results; safe but imprecise algorithms mean more time running tests) and inclusivity (the percentage of modification-revealing test cases selected; a safe RTS algorithm is 100% inclusive).

The approach used by Ekstazi was proposed by Skoglund and Runeson [11] as a modification to the *class firewall* technique in order to make it safe. (The class firewall technique involves statically determining the relationship between modules or classes in a program and uses these relationships to determine which tests to run, but can miss tests that run code inside of a firewall when the methods used to determine the initial structure are not reliable [10].) However, the proof of that modification's safety is informal. In their paper [3] on Ekstazi, Gligoric et al. present explanations for their chosen instrumentation points, but they do not present formal proof, which is what we have sought to rectify here.

Collection Semantics as presented here can be thought of as a labeled semantics with a built-in interpretation function over the label trace of an execution. Labeled semantics are generally used for collecting information during execution (as we do here) and have seen many uses (those given in [8,9,6,2] are just some examples). Labeled semantics itself is an instance of a labeled transition system, a construct formalized in Isabelle in [6]. Connecting our Collection Semantics locale to this work (such as by proving it to be an instance of the LTS locale) would be straightforward, but the result would not have advanced any of the goals of this paper. It could, however, prove useful in allowing simulation of one Collection Semantics by another, especially in attempts to prove that the labels could be correctly replaced by instructions (such as the print instructions used by Ekstazi). We leave this connection and proof to future work.

8 Conclusion and Future Work

In this paper we presented a proof of safety for class-collection based RTS algorithms for JVM programs based on that used by the Java testing library Ekstazi. These proofs were given in the theorem prover Isabelle over the partial Java and JVM semantics JinjaDCI. The first of the algorithms collects classes exhaustively everywhere they were used. The second collects classes when the initialization procedure is called on them and when they are between the referenced and defining classes of static fields and methods called via static instructions. Both differ from Ekstazi's by instrumenting the semantics of the JVM rather than the code run in it. As a code instrumentation, Ekstazi's algorithm cannot collect at actual initialization calls (as they occur at runtime), and replaces this part of our second algorithm with collecting at each instruction that may call the initialization procedure. Thus the safety of our modification of Ekstazi's algorithm can be derived from the safety of our algorithm by seeing that it collects the same set of classes in corresponding places, just slightly earlier when necessary. We also pointed out why this modified set is necessary, thus fixing a bug in their algorithm.

The formalization of the two algorithms' instrumentations is given via defining the Collection Semantics locale. This locale allows the combination of a small-step semantics with a collection function, allowing the latter to be somewhat semantics-agnostic, and the derived semantics to have automatic lemmas of behavioral equivalence with the original.

We leave formal proof of the actual code of a modified Ekstazi to future work, along with the formalization of further aspects of Java's semantics. However, we do not anticipate any of these additions to impact the results presented here. The Collection Semantics locale can be further used to give definitions of various labeled semantics such as those uses mentioned in Section 7. The RTS locale we defined is also sufficiently general to be usable to formally define other RTS algorithms in the context of existence safety.

References

[1] Biswas, S., R. Mall, M. Satpathy and S. Sukumaran, *Regression test selection techniques: A survey*, Informatica (Slovenia) **35** (2011), pp. 289–321.
URL http://www.informatica.si/index.php/informatica/article/view/355

[2] Gadducci, F., F. Santini, L. F. Pino and F. D. Valencia, *A labelled semantics for soft concurrent constraint programming*, in: T. Holvoet and M. Viroli, editors, *Coordination Models and Languages* (2015), pp. 133–149.

[3] Gligoric, M., L. Eloussi and D. Marinov, *Practical regression test selection with dynamic file dependencies*, in: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015 (2015), p. 211–222.
URL https://doi.org/10.1145/2771783.2771784

[4] Klein, G. and T. Nipkow, *A machine-checked model for a java-like language, virtual machine, and compiler*, ACM Trans. Program. Lang. Syst. **28** (2006), p. 619–695.
URL https://doi.org/10.1145/1146809.1146811

[5] Lindholm, T., F. Yellin, G. Bracha and A. Buckley, *The Java Virtual Machine Specification: Java SE 8 Edition* (2015).
URL https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

[6] Lochbihler, A., *Jinja with threads*, The Archive of Formal Proofs. http://afp.sf.net/entries/JinjaThreads.shtml (2007).

[7] Mansky, S. and E. L. Gunter, *Dynamic class initialization semantics: A jinja extension*, in: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019 (2019), p. 209–221.
URL https://doi.org/10.1145/3293880.3294104

[8] Mansky, W., Y. Peng, S. Zdancewic and J. Devietti, *Verifying dynamic race detection*, in: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017 (2017), p. 151–163.
URL https://doi.org/10.1145/3018610.3018611

[9] Nagarakatte, S., J. Zhao, M. M. Martin and S. Zdancewic, *Softbound: Highly compatible and complete spatial memory safety for c*, in: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09 (2009), p. 245–258.
URL https://doi.org/10.1145/1542476.1542504

[10] Rothermel, G. and M. J. Harrold, *Analyzing regression test selection techniques*, IEEE Transactions on software engineering **22** (1996), pp. 529–551.

[11] Skoglund, M. and P. Runeson, *Improving class firewall regression test selection by removing the class firewall*, International Journal of Software Engineering and Knowledge Engineering **17** (2007), pp. 359–378.
URL https://doi.org/10.1142/S0218194007003306