Program Equivalence through Trace Equivalence

Tim Wood
Imperial College London
tim@lexicalscope.com

Sophia Drossopoulou
Imperial College London
s.drossopoulou@imperial.ac.uk

Abstract

Often when programmers modify source code they intend to preserve some parts of the program behaviour. We propose a formal criterion by which to characterise the preserved part of the program behaviour. Two program versions are equivalent up to a set of affected objects $\mathcal{A}$, if executions of the two versions correspond at each execution step when we do not consider the objects in $\mathcal{A}$.

We propose a sufficient condition for this criterion. It suffices to establish that traces of method calls and returns between $\mathcal{A}$ objects and other objects are equivalent. Examining the stack and heap at each execution step is not necessary.

We give a proof of the sufficiency of this condition, much of which we have automatically verified using Dafny. We also discuss our experiences with Dafny and detail how some interesting parts of our Dafny proof work.

1. Introduction

Program maintenance dominates the program lifecycle; fixing bugs and adding features is recurring, expensive, and error-prone: a study of operating system bugs [1] found that at least 14.8%-24.4% of patches were incorrect; a study of application bugs that took more than one attempt to fix [2] found that for 15% of patches there is no direct relationship between the location of the original patch and the location of subsequent supplementary patches. Typically, several programmers over many years maintain overlapping parts of a program, making it difficult to keep a consistent mental view of the whole program.

Our long-term objective is to produce automated tools that help developers to modify programs in a safer, faster and cheaper manner. We are developing a Differential Static Analysis [3] tool, which will automatically check characterisations of the differences between two versions of a program $(\mathrm{V1}, \mathrm{V2})$. By using a previous version of the program as a behavioural base-line, these techniques can reduce the need for extensive programmer-generated specifications.

In this paper we propose a novel formal criterion for equivalence between parts of the behaviour of $\mathrm{V1}$ and $\mathrm{V2}$. We then give a sufficient condition for establishing such an equivalence. This condition depends only on some of the method calls from execution traces. Thus establishing this condition does not require the whole heap to be precisely tracked, some parts of the heap can be approximated, making checking more practical.

We are building a tool which uses approximation and symbolic execution to automatically check our condition, but we do not describe this tool further here.

We give a formal description of the criterion, and a proof of the sufficiency of the condition. We used the Dafny [4] program verifier to automatically prove much of this work, so we discuss our automated proof and experiences with Dafny in some detail.

In section 2 we give a motivating example where our criterion is used to detect a problem in a modification. In section 3 we illustrate with an example how we infer our criterion of equivalence between states from our condition of equivalence between observations taken from execution traces. To reduce the size of the trace that we have to consider, we use knowledge of which part of the state the programmer expects to be affected by a modification. In section 4 we illustrate the abstract model of execution that we will use to prove the soundness of our inference. We give a concrete operational semantics and concrete equivalences, and construct a refinement of the abstract model using the concrete model in section 5. We detail the automated Dafny proof of our theorem and important lemmas in section 6. And we describe our experiences using Dafny to construct this proof, the techniques we used for the more interesting parts, and what we learnt from the process. Finally we discuss related and future work in section 7.

This paper builds on our previous work published in a short paper at the Imperial College Computing Student Workshop 2013 [15]. That paper was primarily concerned with refactorings, contained only some parts of the definitions presented here, and did not contain a proof, automated or otherwise.

2. Motivating Example

Listing 1 contains a small example program, and Listing 2 contains a modification to it. The example illustrates how our technique detects a problem with the modification. The problem is detected without the programmer having to understand the part of the program which exhibits the problem.

```
class Main {
 StudentDb db = new StudentDb();
 Logger logger = new Logger();
 Prizes prizes = new Prizes();
 void main() {
  List<Student> students =
       db.orderedByGrade();
  logger.considered(students);
  prizes.awardTo(students); }}

class Prizes {
 void awardTo(List<Student> students) {
  award(students.get(0));
  award(students.get(1));
  award(students.get(2));}
 void award(final Student student) { /*
      ... */ }}

class Student {
 String name() { /* ... */ }}

class StudentDb {
 List<Student> orderedByGrade() { /* ...
      */ }}

class Logger {
 void considered(List<Student> students) {



  for (Student s : students) {
   println("considered: " + s.name()); }}}
```

Listing 1. Version 1. This Java program awards prizes to top students. It also logs which students were considered for a prize.

```
class Main {
 StudentDb db = new StudentDb();
 Logger logger = new Logger();
 Prizes prizes = new Prizes();
 void main() {
  List<Student> students =
       db.orderedByGrade();
  logger.considered(students);
  prizes.awardTo(students); }}

class Prizes {
 void awardTo(List<Student> students) {
  award(students.get(0));
  award(students.get(1));
  award(students.get(2));}
 void award(final Student student) { /*
      ... */ }}

class Student {
 String name() { /* ... */ }}

class StudentDb {
 List<Student> orderedByGrade() { /* ...
      */ }}

class Logger {
 void considered(List<Student> students) {


  sort(students, compareStudentsByName());
  for (Student s : students) {
   println("considered: " + s.name()); }}}
```

Listing 2. Version 2. This Java program is a modification of the program in 1. Students are now logged in name order. However, this version awards prizes to the wrong students because the sort in Logger unintentionally mutates the list. The modified parts are highlighted.

```
class Main {
 StudentDb db = new StudentDb();
 Logger logger = new Logger();
 Prizes prizes = new Prizes();
 void main() {
  List<Student> students =
       db.orderedByGrade();
  logger.considered(students);
  prizes.awardTo(students); }}

class Prizes {
 void awardTo(List<Student> students) {
  award(students.get(0));
  award(students.get(1));
  award(students.get(2));}
 void award(final Student student) { /*
      ... */ }}

class Student {
 String name() { /* ... */ }}

class StudentDb {
 List<Student> orderedByGrade() { /* ...
      */ }}

class Logger {
 void considered(List<Student> students) {
  List<Student> studentsCopy =
   new ArrayList<>(students);
  sort(studentsCopy, compareStudentsByName());
  for (Student s : studentsCopy) {
   println("considered: " + s.name()); }}}
```

Listing 3. Version 3. This Java program is a modification of the program in 1. Students are now logged in name order. This version awards prizes to the right students by copying the list of students before sorting it. The modified parts are highlighted.

The program awards prizes to the top three eligible students ordered by grade, and also logs which students were considered for prizes. Users complain that it is hard to see who was and was not considered for a prize in a list ordered by grade. The modified program logs students in name order instead. Unfortunately the programmer makes a mistake and the modified program award prizes to the wrong students.

Our technique can avoid problems like this. It allows the programmer to establish that the modification only affects parts of the program concerned with logging.

According to our criterion the programmer will describe the set of objects $\mathcal{A}$ she *expects to be affected* by the modification. Our criterion requires that both versions of the program reach equivalent states at every execution step, *when the objects in $\mathcal{A}$ are not considered*.

For the rest of the paper we often use the term *affected* as an abbreviation for *expected to be affected*, and *unaffected* to mean not expected to be affected. The objective of our approach is to check our criterion — that the allegedly unaffected objects really are unaffected by the modification.

The developer specifies the affected set as objects of class `Logger`, `PrintStream`, `List`[1] and `Student`. This is a reasonable expectation as she expects to impact: the `Logger` and `PrintStream` due to changing the order in which the students are logged; the `List` due to sorting the list of students by name; and the `Students` due to reading their names in a different order for output to the logger. Note that the affected objects are a superset of the objects with modified code.

Contrary to the developer's expectations, the modification in Listing 2 does impact some allegedly unaffected objects. The call to `sort` on line 27 mutates the list of students. Thus the values returned by calls to `List.get` from the `Prizes` object will sometimes be different in executions of Listing 1 and Listing 2, which may cause the prizes to be awarded incorrectly on lines 12-14. Our criterion does not hold due to the allegedly unaffected `Prizes` object sometimes having references to different students in executions of the two versions.

Note that the problem is detected without the programmer mentioning, or even needing to be aware of, the existence of the `Prizes` object.

The modification in Listing 3 corrects the problem. This version copies the list before sorting it, so both prints the students in the desired order and still awards the prizes in the intended manner. Our criterion is satisfied because there is always an equivalence between the unaffected objects in executions of Listing 1 and Listing 3.

3. Equivalences and Traces

We will informally describe our program equivalence criterion, and our condition. We will use an illustrative example, small enough to draw the whole stack and heap at interesting points in the execution. The example used in this section is different from the example in the previous section.

3.1 Equivalence between Unaffected Objects

We consider states to be equivalent if they have the same shape when only the unaffected objects are considered.

Listing 5 shows a version v1 of a fifo queue consisting of two classes `FiFo` and `Node`. Listing 4 shows a test program which uses the fifo queue. Listing 6 shows a modification of v1, to produce a new version v2. In v2 the implementation and representation of the queue has changed. We use the same test program for both v1 and v2.

The allegedly affected objects are those of class `FiFo` or `Node`.

Figure 1 shows the stack and heap when execution of the test program is just about to return from the second call to `add` — on line 2 of Listing 4. States are shown for executions of the test program with both versions of the fifo queue.

In figure 1 the states do not have the same shape when all objects are considered, due to the change in representation of the queue. The states are equivalent when only the unaffected objects are considered, as shown in the third diagram.

Our criterion for program equivalence is that executions of both versions have equivalent states at each execution step.

Generally, *if an interesting property of the program is encapsulated by an object or set of objects that are actually unaffected, we know that the interesting property will be unaffected by the modification.*

3.2 Establishing Equivalence

States may be large, and executions may reach a large number of states. Traversing the whole stack and heap at each execution step is computationally expensive, and requires the whole heap to be tracked. We have discovered a condition for our criterion that does not rely on the whole heap and may totally ignore many execution steps.

Our condition depends only on method calls and returns between unaffected objects and affected objects. It simply requires that method call *traces* are equivalent in executions of the two versions.

Traces comprise only the sequence of constructor calls, method calls and method returns that occur between the affected objects (here objects of class `FiFo` or `Node`) and the unaffected objects (everything else).

Figure 2 shows traces from executing the test program in Listing 4. A trace from running the test program against the fifo queue from Listing 5 is compared to a trace from running the test program against the modified fifo queue from Listing 6. A mapping exists between addresses such that all parameters which are aliases in one trace are aliases in the other, so we consider the traces to be equivalent.

The traces are equivalent so our condition holds, and thus we conclude that our criterion also holds. States are always equivalent when only unaffected objects are consid-

[1] Also internal classes used in the representation of `List`s

```
class Test { FiFo fi = new FiFo(); Object o1 = new Object(), o2 = new Object();
 void test() { fi.add(o1); fi.add(o2); fi.remove(); fi.remove();}}
```

Listing 4. Test program for the FiFo queue.

```
class FiFo {
 Node f;
 void add(final Object o) {
  if(f == null) { f=new Node(o); }
  else { f.add(o); }}
 Object remove() {
  Object r=f.value(); f=f.next(); return r;}}

class Node {
 Object o; Node n;
 Node(Object o) { this.o=o; }
 Node next() { return n; }
 Object value() { return o; }
 void add(final Object o) {
  if(n == null) { n=new Node(o); }
  else { n.add(o); }}}
```

Listing 5. Queue version v1 — a fifo queue with a recursive implementation of `add`.

```
class FiFo {
 Node f, l;
 void add(final Object o) {
  if(f == null) { l=f=new Node(o); }
  else { l=l.add(o); }}
 Object remove() {
  Object r=f.value(); f=f.next(); return r;}}

class Node {
 Object o; Node n;
 Node(Object o) { this.o=o; }
 Node next() { return n; }
 Object value() { return o; }
 Node add(Object o) {
 return n=new Node(o); }}
```

Listing 6. Queue version v2 — fifo queue with last element pointer. A modification of the Queue in Listing 5. The modified parts are highlighted.

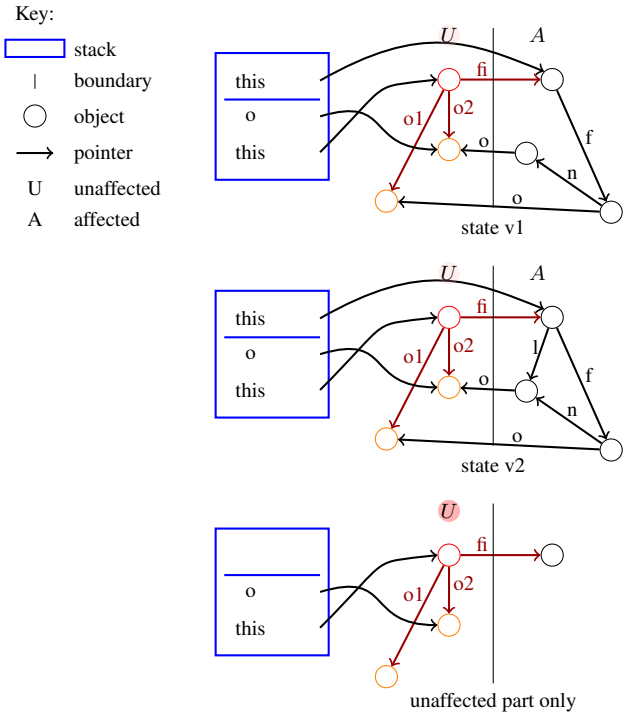

Figure 1. Two states of the test (Listing 4) are shown. The top state, v1, is from execution of the test with the code from Listing 5. The middle state, v2, is from execution of the test with the code from Listing 6.

States v1 and v2 differ when all of the state is considered. The pointer *l* only appears in state v2.

When only the unaffected objects are considered, then the states are equivalent. The bottom diagram shows the states with the affected objects elided.

ered. Note that we do not inspect the unaffected part of the heap to establish the criterion, nor do we need to inspect every execution step.

We prove the sufficiency of the condition in the following sections. But an intuition is that the traces capture the complete interaction between the unaffected and affected objects, and that the behaviour of the unaffected objects is not sensitive to differences that only relate to the actual addresses of affected objects.

4. Labelled Transition System Equivalences

We want to prove that our condition is sufficient to establish our program equivalence criterion. We prove this theorem (4.3) with an abstraction of program execution, considering abstract equivalences between abstract states and abstract labels in a labelled transition system (LTS). Then in section 5 we refine the abstract model to a concrete model of execution, with concrete equivalences, and thus can use theorem 4.3 to reason about concrete executions.

Large parts of this work have been checked with the Dafny verifier. To make our proof more amenable to Dafny we have structured it into several abstract modules which we then refine. We will follow the same structure here. More discussion of our use of refinement is in section 6.2.

4.1 Abstract Semantics

We abstract execution as a deterministic labelled transition system over abstract states and abstract labels (definition 4.1). Given a single step transition relation we construct a multi-step relation and define traces as sequences of labels encountered during multi-step execution.

t_1		t_2		t_3	
Method	**Params**	**Method**	**Params**	**Method**	**Params**
Fifo	this: 3	Fifo	this: 10	Fifo	this: 7
return	3	*return*	10	*return*	7
add	this: 3 o: 1	add	this: 10 o: 8	add	this: 7 o: 4
return		*return*		*return*	
add	this: 3 o: 2	add	this: 10 o: 9	add	this: 7 o: 6
return		*return*		*return*	
remove	this: 3	remove	this: 10	remove	this: 7
return	1	*return*	8	*return*	6
remove	this: 3	remove	this: 10	remove	this: 7
return	2	*return*	9	*return*	4

$t_1 \cong t_2$ using mapping (1,8),(3,10),(2,9) $\quad t_2 \ncong t_3$

Figure 2. Three traces. Trace t_1 is from an execution of the test in Listing 4 with the queue from Listing 5. Trace t_2 is from an execution of the same test, but run against the modified queue from Listing 6. Trace t_3 is for illustration, it is fictional and does not come from either version. Trace t_1 is equivalent to trace t_2 under the address bijection shown. Trace t_3 is not equivalent to t_1 or t_2, because there exists no bijection between the addresses of t_1 and t_3 that preserves the aliasing structure of the traces.

Definition 4.1 *A tuple* $\mathcal{LTS} = (S, L, \rightsquigarrow)$ *is an* abstract labelled transition system *if*

- S *is the set of states, ranged over by* s
- L *is the set of labels, ranged over by* l
- $\rightsquigarrow \subseteq S \times L \times S$ *is the transition relation written* $s_1 \rightsquigarrow {}^{l} s_2$
- $\rightsquigarrow$ *is deterministic:*

$$\forall s, l_1, l_2, s_3, s_4 : \\ s \overset{l_1}{\rightsquigarrow} s_3 \ \wedge \ s \overset{l_2}{\rightsquigarrow} s_4 \implies l_1 = l_2 \ \wedge \ s_3 = s_4$$

NOTATION

- $\twoheadrightarrow$ *is the transitive reflexive closure of* $\rightsquigarrow$
- *We say* trace *to mean the sequence of labels* ls *encountered in a multi-step execution, and write* $\twoheadrightarrow^{ls}$.

4.2 Abstract Equivalence

Here we give an abstract definition of equivalence between states and between traces for an LTS. We require that equivalence between traces is preserved by execution. If from equivalent states which produce equivalent traces, we take a single execution step from each, then the reached states must also be equivalent and produce equivalent traces.

Definition 4.2 *A tuple* $\mathcal{C} = (\simeq, \cong, A)$ *is an* abstract labelled transition system equivalence *for some abstract labelled transition system* $\mathcal{LTS} = (S, L, \rightsquigarrow)$ *if*

(i) A *is a set, ranged over by* α

(ii) $\simeq \subseteq S \times A \times S$ *is an equivalence relation over states, which we write* $s_1 \simeq_\alpha s_2$.

(iii) $\cong \subseteq L^* \times A \times L^*$ *is an equivalence relation over traces, and is written* $ls_1 \cong_\alpha ls_2$

(iv) If $\cong$ *holds the traces are the same length:*

$$\forall ls_1, ls_2, \alpha : ls_1 \cong_\alpha ls_2 \implies |ls_1| = |ls_2|$$

(v) Execution preserves $\simeq$ *and* $\cong$

$$\forall s_1, s_2, s_3, s_4, l_1, l_2, ls_3, ls_4, \alpha : \\ \left(s_1 \overset{l_1}{\rightsquigarrow} s_3 \ \wedge \ s_2 \overset{l_2}{\rightsquigarrow} s_4 \ \wedge \ s_1 \simeq_\alpha s_2 \ \wedge \ l_1 \cdot ls_3 \cong_\alpha l_2 \cdot ls_4\right) \\ \implies \\ (\exists \alpha' : s_3 \simeq_{\alpha'} s_4 \ \wedge \ ls_3 \cong_{\alpha'} ls_4)$$

4.3 Theorem

We show that execution from equivalent states, via equivalent traces, will reach equivalent states.

Theorem 4.3 *Given an abstract labelled transition system equivalence* $\mathcal{C} = (\simeq, \cong, A)$ *for an abstract labelled transition system* $\mathcal{LTS} = (S, L, \rightsquigarrow)$*:*

$$\forall s_1, ls_1, s_3, s_2, ls_2, s_4, \alpha : \\ s_1 \overset{ls_1}{\twoheadrightarrow} s_3 \ \wedge \ s_2 \overset{ls_2}{\twoheadrightarrow} s_4 \ \wedge \ s_1 \simeq_\alpha s_2 \ \wedge \ ls_1 \cong_\alpha ls_2 \\ \implies \\ \exists \alpha' : s_3 \simeq_{\alpha'} s_4$$

Proof. Automatically checked by Dafny. Sketch: Since $\simeq$ and $\cong$ are preserved by execution, and by induction on $\twoheadrightarrow$. □

5. Refinement and Concrete Semantics

We give semantics for a simple object-oriented language we call $\mathcal{L}_{\text{ver}}$. Then we consider how its objects can be partitioned into unaffected and affected objects. We give concrete equivalences for its traces and states, considering only the unaffected objects. Finally, we will show that a refined abstract labelled transition system and abstract labelled transition system equivalence can be constructed for this language and equivalences. This refinement allows us to use theorem 4.3, with programs written in $\mathcal{L}_{\text{ver}}$, i.e. to establish our criterion by establishing our condition.

Of particular interest will be: picking a useful equivalence between the unaffected objects; and dealing with differences in the number of executions steps due to the modification.

5.1 Language

Our language, $\mathcal{L}_{\text{ver}}$, is similar to but much smaller than Java bytecode. States have a stack of stack frames and a heap of objects. Stack frames have local variables for the receiver and parameter as well as an operand stack, and a continuation in lieu of the program counter. Our language does not have loops but does have recursion, so loops could be encoded.

We represent two versions $(\text{V1}, \text{V2})$ of the program directly in the semantics of $\mathcal{L}_{\text{ver}}$. States contain the version of the program they are from, and method lookup is indexed by version as well as class and method name.

The transition relation of $\mathcal{L}_{\text{ver}}$ is labelled. Transitions that add or remove a stack frame (i.e. *NEW*, *CALL* and *RET*) have a label containing information about the transition. All other transitions have the empty label τ.

Only fields of the current method receiver can be read or written, i.e. all fields are private. The only values in our language are addresses, although an extension of our approach to a language with other types of values, such as `int`s, is straightforward.

Definition 5.1 *Our language $\mathcal{L}_{\text{ver}}$ is*

- *The set of versions $Version \stackrel{def}{=} \{\text{V1}, \text{V2}\}$. Ranged over by v.*
- *Enumerable sets, $Field$ ranged over by f, $Method$ ranged over by m, $Class$ ranged over by c.*
- *Wellordered[2] enumerable set $Address$, ranged over by a.*
- *The set of local variable names $Var \stackrel{def}{=} \{\texttt{this}, \texttt{x}\}$. Ranged over by x.*
- *Expressions $Expr$ are ranged over by e and have the grammar:*

  $e \quad := \quad e; e \mid \texttt{write}\ f \mid \texttt{read}\ f \mid \texttt{push}\ x \mid \texttt{call}\ m \mid \texttt{new}\ c \mid \texttt{if}\{e\}\texttt{else}\{e\} \mid \texttt{nop} \mid \texttt{except}$
- *A stack, ranged over by $\tilde{\sigma}$, is a sequence of stack frames $Stack \stackrel{def}{=} StackF^*$ where*
  - $StackF \stackrel{def}{=} Method \times Expr \times Vars \times Operands$
  - $Vars \stackrel{def}{=} Var \rightharpoonup Address$
  - $Operands \stackrel{def}{=} Address^*$
- *A heap maps addresses to objects. Ranged over by h. $Heap \stackrel{def}{=} Address \rightharpoonup Object$ where:*
  - $Object \stackrel{def}{=} Class \times (Field \rightharpoonup Address)$
- *A state is the program version, a stack, and a heap. Ranged over by s. $S \stackrel{def}{=} Version \times Stack \times Heap$*
- *Labels are either empty or associated with a constructor call, a method call or a return. $L \stackrel{def}{=} \tau \cup (Class \times Address \times Address) \cup (Class \times Method \times Address \times Address) \cup (Address)$*
- *A total function $\mathcal{M} : Version \times Class \times Method \to Expr$ for looking up method bodies for version* V1 *or* V2*. We model missing methods as methods having a body consisting of the expression `except`, which is a stuck expression.*
- *A total function $next : Heap \to Address$, where $next(h)$ gives the least element of $Address$ that is greater than all elements in the domain of h, or the least element of $Address$ if h is empty.*
- *The transition relation $\to$ is defined by the rules in Figure 3.*

NOTATION

- *$StackF$ ranged over by σ*
- *$\sigma(meth)$ for the method $\sigma\downarrow_1$*
- *$\sigma(exp)$ for the expression $\sigma\downarrow_2$*
- *$\sigma(x)$ for the local variable $\sigma\downarrow_3(x)$*
- *$\sigma(ops)$ for the operands $\sigma\downarrow_3$*
- *$head(\sigma(ops))$ for the 0^{th} operand $(\sigma\downarrow_3)_0$*
- *$tail(\sigma(ops))$ for the remaining operands $(\sigma\downarrow_3)_{1\ldots n}$*
- *$h(a, class)$ to mean the class of the object at address a in heap h*
- *$h(a, f)$ to mean the value of the field f of the object at address a in heap h*
- *$sf(s)$ to mean the topmost stackframe of s*
- *$depth(s)$ to mean the number of stack frames in s*
- *$heap(s)$ to mean the heap of s*
- *$\twoheadrightarrow$ to mean the transitive reflexive closure of $\to$*

5.2 Partitioning into Affected and Unaffected Objects

We assume a predicate that determines for every heap address whether the object is allegedly unaffected[3]. This predicate must be preserved by execution. The methods of unaffected objects must be identical in both versions of the program. The methods, and even the classes of affected objects, can be arbitrarily different between the versions.

[2] Totally ordered, and each non-empty subset has a least element. Allows us deterministic allocation.

[3] We leave this predicate uninterpreted here, but in our tool it is implemented by a programmer-provided side-effect free Java expression over the whole program state. For example: a predicate may be based on the class of objects, or where in the program the objects were constructed.

$$\frac{\sigma(exp) = \texttt{write}\ f; e \qquad h' = h[\sigma(\texttt{this}), f \mapsto head(\sigma(ops))]}{v, \sigma \cdot \tilde{\sigma}, h \to^{\tau} v, \sigma[exp \mapsto e, ops \mapsto tail(\sigma(ops))] \cdot \tilde{\sigma}, h'}\ \text{WRITEF} \qquad \frac{\sigma(exp) = \texttt{nop}; e}{v, \sigma \cdot \tilde{\sigma}, h \to^{\tau} v, \sigma[exp \mapsto e] \cdot \tilde{\sigma}, h}\ \text{NOP}$$

$$\frac{\sigma(exp) = \texttt{read}\ f; e \qquad a = h(\sigma(\texttt{this}), f)}{v, \sigma \cdot \tilde{\sigma}, h \to^{\tau} v, \sigma[exp \mapsto e, ops \mapsto a \cdot \sigma(ops)] \cdot \tilde{\sigma}, h}\ \text{READF} \qquad \frac{\sigma(exp) = \epsilon \qquad a = head(\sigma(ops)) \qquad l = (a)}{v, \sigma \cdot \sigma' \cdot \tilde{\sigma}, h \to^{l} v, \sigma'[ops \mapsto a \cdot \sigma'(ops)] \cdot \tilde{\sigma}, h}\ \text{RET}$$

$$\frac{\sigma(exp) = \texttt{push}\ x; e}{v, \sigma \cdot \tilde{\sigma}, h \to^{\tau} v, \sigma[exp \mapsto e, ops \mapsto \sigma(x) \cdot \sigma(ops)] \cdot \tilde{\sigma}, h}\ \text{PUSH}$$

$$\frac{\begin{array}{lll}\sigma(exp) = \texttt{call}\ m; e & a = head(tail(\sigma(ops))) & l = (c, m, a, a') \\ c = h(\sigma(\texttt{this}), class) & a' = head(\sigma(ops)) & e' = \mathcal{M}(v, c, m)\end{array}}{v, \sigma \cdot \tilde{\sigma}, h \to^{l} v, [exp \mapsto e', \texttt{this} \mapsto a, \texttt{x} \mapsto a'] \cdot \sigma[exp \mapsto e, ops \mapsto tail(tail(\sigma(ops)))] \cdot \tilde{\sigma}, h}\ \text{CALL}$$

$$\frac{\sigma(exp) = \texttt{new}\ c; e \qquad h' = h[a, class \mapsto c] \qquad a = nexth \qquad l = (c, a, a')}{v, \sigma \cdot \tilde{\sigma}, h \to^{l} v, [\texttt{this} \mapsto a, \texttt{x} \mapsto a] \cdot \sigma[exp \mapsto e] \cdot \tilde{\sigma}, h'}\ \text{NEW}$$

$$\frac{\sigma(exp) = \texttt{if}\{e\}\texttt{else}\{e'\} \qquad e'' = \begin{cases} e \text{ if } head(\sigma(ops)) = head(tail(\sigma(ops))) \\ e' \text{ otherwise} \end{cases}}{v, \sigma \cdot \tilde{\sigma}, h \to^{\tau} v, \sigma[exp \mapsto e'', ops \mapsto tail(tail(\sigma(ops))] \cdot \tilde{\sigma}, h}\ \text{COND}$$

Figure 3. Operational Semantics

Definition 5.2 *For language $\mathcal{L}_{\text{ver}}$, U is a partitioning predicate if*

- *U is a predicate on addresses and heaps*

$$U \subseteq Address \times Heap$$

- *U is preserved by execution*

$$\begin{array}{l} \forall s_1, l_1, s_3, a: \\ a \in heap(s_1) \ \wedge\ a \in heap(s_3) \ \wedge\ s_1 \xrightarrow{l_1} s_3 \\ \quad \Longrightarrow \\ (U(a, heap(s_1)) \iff U(a, heap(s_3))) \end{array}$$

- *Unaffected objects don't have modified code*

$$\begin{array}{l} \forall s, a, m: \\ U(a, heap(s)) \\ \quad \Longrightarrow \\ code(\text{V1}, s, a, m) = code(\text{V2}, s, a, m) \\ \textit{where} \\ \qquad code(v, s, a, m) \stackrel{def}{=} \mathcal{M}(v, heap(s)(a, class), m) \end{array}$$

NOTATION

- *$U(s, \sigma)$ is short for the receiver of the stack frame σ is unaffected*

$$U(s, \sigma) \stackrel{def}{=} U(\sigma(\texttt{this}), heap(s))$$

- *$U(s)$ means the receiver of the top stack frame is unaffected. We call such an s an unaffected state*

$$U(s) \iff U(sf(s), s)$$

- *$\mathcal{U}heap$ is short for the part of the heap containing only the unaffected objects*

$$\mathcal{U}heap(s)(a) \stackrel{def}{=} \begin{cases} heap(s)(a) & \text{if } U(heap(s), a) \\ \textit{undef} & \text{otherwise} \end{cases}$$

5.3 Abstracting differences in the number of affected execution steps

The modification may cause executions of version V1 to differ from executions of V2 in the number of execution steps from affected states. Such executions may however still be considered equivalent when only the unaffected states are considered. We abstract steps between affected states by *squashing* series of consecutive steps from affected states into a single step, and by replacing some labels with τ. Consequently no two adjacent states in the abstract labelled transition system are both affected. And only transitions between affected and unaffected states have labels.

Figure 4 shows an example execution and its abstraction. States s_1 and s_3 are unaffected, so the transition is preserved but the label is replaced by τ. State s_5 is affected, so the transition and label from s_3 to s_5 is preserved. States s_7 and s_9 are affected, but state s_{11} is unaffected, so the transitions are squashed into a single step and the label of the last transition is used.

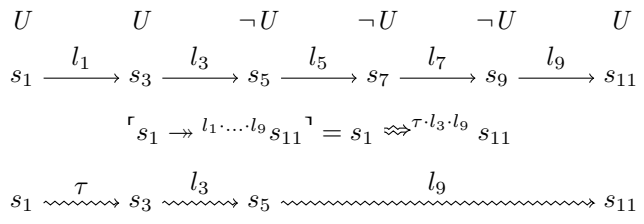

Figure 4. Squashing an execution from $\mathcal{L}_{\text{ver}}$ into an abstract labelled transition system execution. Consecutive transitions from $\neg U$ states are combined. And labels on transitions between U and U states are replaced with τ.

We can think of this squashing abstraction as analogous to a havoc command [5]. The abstract transition from an affected state represents arbitrary work that the affected objects do on the affected part of the state.

Lemma 5.3 *By taking S, L and $\rightarrow$ from $\mathcal{L}_{\text{ver}}$ and a partitioning predicate U we can construct an abstract labelled transition system $\mathcal{LTS} = (S, L, \rightsquigarrow)$, where:*

$$
\begin{aligned}
& s_1 \overset{l}{\rightsquigarrow} s_2 \overset{def}{\iff} \\
& \quad \Big(U(s_1) \ \wedge \ U(s_2) \ \wedge \ l = \tau \ \wedge \ \exists l' : s_1 \overset{l'}{\rightarrow} s_2 \Big) \\
& \vee \ \Big(U(s_1) \ \wedge \ \neg U(s_2) \ \wedge \ s_1 \overset{l}{\rightarrow} s_2 \Big) \\
& \vee \ \Big(\neg U(s_1) \ \wedge \ U(s_2) \ \wedge \ s_1 \overset{l}{\rightarrow} s_2 \Big) \\
& \vee \ \begin{pmatrix} \neg U(s_1) \ \wedge \ U(s_2) \ \wedge \\ \exists l1, s3 : \neg U(s3) \ \wedge \ s_1 \overset{l1}{\rightarrow} s_3 \ \wedge \ s_3 \overset{l}{\rightsquigarrow} s_2 \end{pmatrix}
\end{aligned}
$$

Definition 5.4 *Squashed Execution*
Given $\mathcal{LTS} = (S, L, \rightsquigarrow)$, for $\mathcal{L}_{\text{ver}}$ and U. Then a squashed execution is

$$\ulcorner s_1 \overset{ls_1}{\twoheadrightarrow} s_3 \urcorner \overset{def}{=} s_1 \overset{ls'_1}{\rightsquigarrow} s'_3$$

whenever $s_1 \twoheadrightarrow^{ls_1} s_3 \ \wedge \ s_1 \overset{ls'_1}{\rightsquigarrow} s'_3$ and either of

- $U(s_3) \ \wedge \ s_3 = s'_3$
- $\neg U(s_3) \ \wedge \ \forall s : s'_3 \twoheadrightarrow s \ \wedge \ s \twoheadrightarrow s_3 \implies \neg U(s)$

Corollary 5.5 *Every concrete execution $s_1 \twoheadrightarrow^{ls_1} s_3$ has a* squashed execution.

5.4 Equivalences

To complete our refinement, and thus show that theorem 4.3 applies to $\mathcal{L}_{\text{ver}}$, it is necessary to provide concrete equivalences that are preserved by execution. An equivalence between states for our criterion, and an equivalence between traces for our condition.

The equivalence definitions must be carefully constructed with reference to the desired properties and the operational semantics of the language. It was not immediately obvious to us exactly how the equivalences should be defined. We started by trying to work out the equivalence definition by looking at the definition of states. But we did not produce a useful equivalence between stacks. By careful thought about how the proof would work, we realised that by considering each rule in the operational semantics we could systematically create a sufficient equivalence definition.

The critical aspect of selecting the equivalence between states is to consider exactly the structure of the unaffected part of each state.

The critical aspect of selecting the equivalence between traces is that the labels on transitions between unaffected and affected states must uniquely determine the effect that the transition has on the unaffected part of the state.

Equivalence between heaps is discussed in section 3.1, and illustrated in figure 1. We establish a bijection on the addresses of objects that preserves aliasing between the fields of the unaffected objects.

We also need to consider when the stacks are equivalent. Figure 5 illustrates which stacks we consider to have equivalent structure. We ignore differences in the number of affected stack frames while requiring that the number of changes from affected to unaffected remains the same. This is very similar to the squashing process we used to abstract execution in section 5.3.

Once we determine that stacks have equivalent structure, we require that structurally related stack frames have the same continuation and that the locals and operands are related by the same bijection on addresses as the heap.

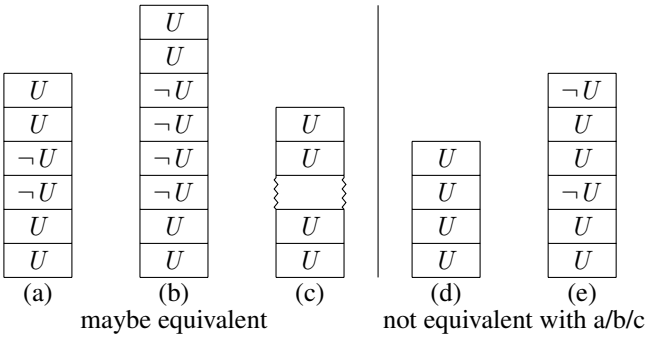

Figure 5. Four stacks, labelled with U if the receiver of the frame is unaffected, and $\neg U$ if the receiver of the frame is affected. Stacks (a) and (b) have equivalent structure, even though they differ in the number of affected frames. We consider both (a) and (b) to have the same structure as diagram (c). However stack (d) does not have equivalent structure with any of the others, because it has no affected frames between the unaffected frames. Stack (e) also does not have equivalent structure with any of the others, because of the additional unaffected frame at the top of the stack.

Definition 5.6 *For language $\mathcal{L}_{\text{ver}}$ and a partitioning predicate U we define a tuple $\mathcal{CLA} = (\simeq, \cong, A)$ of equivalences where:*
The set of mappings A is the set of injective mappings between addresses.

We write $\alpha(a)$ to mean the address that α maps a to, but also when as is a set of addresses we write $\alpha(as)$ to mean the set obtained by applying alpha to every element of as, and similarly when $\tilde{a}$ is a sequence of address $\alpha(\tilde{a})$ is the sequence obtained by applying alpha to every element of $\tilde{a}$.

$$A \stackrel{def}{=} \{\alpha \mid \alpha : Address \to Address \ \wedge \ \alpha \text{ is injective}\}$$

STATE EQUIVALENCE

States s_1, s_2 are equivalent under α, written $s_1 \simeq_\alpha s_2$, iff

HEAP

α preserves the structure of the unaffected part of the heap:

- *α preserves U*

$$\alpha(dom(\mathcal{U}heap(s_1))) = dom(\mathcal{U}heap(s_2))$$

- *α preserves field aliasing for the unaffected objects*

$$\forall a, f : \alpha(\mathcal{U}heap(s_1)(a, f)) = \mathcal{U}heap(s_2)(\alpha(a), f)$$

- *α preserves class of unaffected objects*

$$\forall a : \mathcal{U}heap(s_1, a, class) = \mathcal{U}heap(s_2, \alpha(a), class)$$

STACK

α preserves the structure of the stacks, $stack(s_1) \simeq_\alpha stack(s_2)$
where $\sigma_1 \cdot \tilde{\sigma_1} \simeq_\alpha \sigma_2 \cdot \tilde{\sigma_2}$ $\stackrel{def}{\iff}$

- *The top frames σ_1, σ_2 are unaffected, and equivalent, and the remaining frames are equivalent:*
  - *$\sigma_1 \simeq_\alpha \sigma_2$*
  - *$\tilde{\sigma_1} \simeq_\alpha \tilde{\sigma_2} \ \vee \ \tilde{\sigma_1} = \tilde{\sigma_2} = \epsilon$*
- *Or the top frames σ_1, σ_2 are affected and if we skip down each stack until the next unaffected frames then the stacks from there are equivalent. $\exists \tilde{\sigma_3}, \tilde{\sigma_4}, \sigma_3, \sigma_4, \sigma_5, \sigma_6$:*
  - *$\tilde{\sigma_1} = \tilde{\sigma_3} \cdot \sigma_3 \cdot \tilde{\sigma_5}$*
  - *$\tilde{\sigma_2} = \tilde{\sigma_4} \cdot \sigma_4 \cdot \tilde{\sigma_6}$*
  - *all frames in $\tilde{\sigma_3}$ and $\tilde{\sigma_4}$ are affected.*
  - *σ_3 and σ_4 are unaffected*
  - *$\sigma_3 \cdot \tilde{\sigma_5} \cong_\alpha \sigma_4 \cdot \tilde{\sigma_6}$*
- *Or the top frames σ_1, σ_2 are affected and all the remaining frames $\tilde{\sigma_1}, \tilde{\sigma_2}$ are affected*

Where individual stack frames are equivalent $\sigma_1 \simeq_\alpha \sigma_2$ $\stackrel{def}{\iff}$

- *they have the same continuation:* $\sigma_1(exp) = \sigma_2(exp)$
- *local variables are mapped by α:* $\forall x : \alpha(\sigma_1(x)) = \sigma_1(x)$
- *operands are mapped by α:* $\alpha(\sigma(opers)) = \sigma(opers)$

TRACE EQUIVALENCE

Traces ls_1 and ls_2 are equivalent under α when alpha preserves the aliasing structure of the traces
$ls_1 \cong_\alpha ls_2$ $\stackrel{def}{\iff}$

- $|ls_1| == |ls_2|$
- $\forall i : \begin{cases} ls_2(i) = (c, \alpha(a), \alpha(a')) & \text{if } ls_1(i) = (c, a, a') \\ ls_2(i) = (c, m, \alpha(a), \alpha(a')) & \text{if } ls_1(i) = (c, m, a, a') \\ ls_2(i) = (\alpha(a)) & \text{if } ls_1(i) = (a) \\ ls_2(i) = \tau & \text{if } ls_1(i) = \tau \end{cases}$

5.5 Equivalences preserved by Execution

We must now show that the defined equivalences are preserved by execution, as defined in definition 4.2(v).

Lemma 5.7 *$\mathcal{CLA}$ is an abstract labelled transition system equivalence for $\mathcal{L}_{\text{ver}}$ and U*

Proof. Checked by Dafny. To prove: equivalence is preserved by execution. Sketch:

First, we show by case analysis, for any $s_1 \to {}^{l_1} s_3$ if $U(s_1) \neq U(s_3)$ then the transition must be a method call, method return or new object construction.

Then, for arbitrary $s_1, s_2, s_3, s_4, l_1, l_2, ls_1, ls_2, \alpha$ assume $s_1 \rightsquigarrow {}^{l_1} s_3 \ \wedge \ s_2 \rightsquigarrow {}^{l_2} s_4$, and $l_1 \cdot ls_1 \cong_\alpha l_2 \cdot ls_2$ and $s_1 \simeq_\alpha s_2$. To Show $\exists \alpha' : ls_1 \cong_{\alpha'} ls_2 \wedge s_1 \simeq_{\alpha'} s_2$ By cases:

- $U(s_1)$ — $\rightsquigarrow$ is a single $\to$ step. And $\to$ from unaffected states preserves $\cong$, by case analysis on the $\to$ rewrite rules we show that all rules are agnostic to the actual values of addresses.
- $\neg U(s_1)$ — $\rightsquigarrow$ is a $\twoheadrightarrow$ step within the affected part, followed by a single $\to$ step from the affected part to the unaffected part.
  - Evaluation steps in the affected part do not affect the U part of the heap, so preserve $\cong$ by case analysis on $\to$ and induction on $\twoheadrightarrow$.
  - The evaluation step from the affected part to the unaffected part changes the U part of the heap, but it does so in a way uniquely determined by the labels, l_1 and l_2. By case analysis on the kind of label, we use the fact that α is also a mapping between the labels, and the rules for method call, object construction and method return. We are able to show that α is still a mapping after the transition.

□

5.6 Observing labels on calls between unaffected and affected is sufficient

The following corollary guarantees for equivalent states s_1 and s_2 whose executions lead to s_3 and s_4, with traces ls_1 and ls_2 respectively, that equivalence of the squashed

(definition 5.4) traces is sufficient to establish equivalence of s_3 with s_4.

Corollary 5.8 *Given some* U *and* $\mathcal{CLA} = (\simeq, \cong, A)$ *for* $\mathcal{L}_{\text{ver}}$. *For any states* s_1, s_2, s_3, s_3', s_4, s_4', *and traces* ls_1, ls_1', ls_2, ls_2. *If*

- $\ulcorner s_1 \twoheadrightarrow^{ls_1} s_3 \urcorner = s_1 \Rrightarrow^{ls_1'} s_3'$
- $\ulcorner s_2 \twoheadrightarrow^{ls_2} s_4 \urcorner = s_2 \Rrightarrow^{ls_2'} s_4'$
- $\exists \alpha : s_1 \simeq_\alpha s_2 \ \wedge \ ls_1' \cong_\alpha ls_2'$

then $\exists \alpha' : s_3 \simeq_{\alpha'} s_4$

Consequently, in order to establish our criterion, it is sufficient to establish trace equivalence. Specifically, it must be established that for the initial states, for every execution with some trace in the first version there exists an equivalent trace in the second version. And also the other way around.

6. Using Dafny

We have checked much of our formalism automatically using the Dafny verification system. Our experience using Dafny gave us a lot of feedback on the design of our formalism. Often the feedback caused us to think about how to structure the definitions to make the proofs simpler, leading to an overall increase in clarity. However, considerable effort was taken up by trying to understand how to write the formalism so that the verifier could check it reasonably quickly.

In this section we report on how we structured our proof. Many parts of our formalism could be translated quite directly into Dafny. Other parts required more inventiveness. So we also look in detail at some particularly interesting parts of the proof.

6.1 Verifier Timeouts and Debugging

When implementing our proof in Dafny a critical issue for us, and others [6], was how quickly Dafny gave feedback on failed verification. Verifiable lemmas often go through very fast. But when developing and debugging lemmas, timeouts or long delays occur often. Thus, the need to reduce feedback delays influenced the design of many parts of our proof.

We encountered timeouts and long delays when:

- proof insufficiently detailed
- proof contained an error
- lemma large or using many definitions, see section 6.9.

Verification can fail by a timeout or by counter example. The feedback that Dafny gives after a verifier timeout is poor, at best only indicating what line it was working on when the timeout occurred. Even this is not so helpful, because sometimes some earlier part of the lemma is very slow to verify but the solver timeout happens to occur whilst the verifier is working on some later part. Or sometimes the proof is incorrect or insufficiently detailed. But the Dafny feedback after timeout does not distinguish these cases.

Our proof contains two large definitions, the semantics of $\mathcal{L}_{\text{ver}}$ and the concrete definition of state equivalence. These definitions are core to the proof and widely used throughout, sometimes in the same lemma. We often experienced poor performance when working with these definitions.

To mitigate this problem we split the definitions into multiple smaller predicates, and marked them with the annotation `{:opaque true}`. This annotation instructs Dafny not to automatically reveal the definition of the predicate when the predicate is used. We selectively revealed the definitions when needed using the `reveal_MyPredicate()` construct.

We discuss further specific strategies for improving how quickly we got feedback in context in sections 6.2, 6.3, 6.4, 6.7 and 6.9.

6.2 Modules and Refinement In Dafny

We used modules and refinement [6] to split our formalism across files. Our original motivation for doing so was performance. However, interestingly, using refinement in this way allowed us to prove a more general result than we may otherwise have attempted.

Dafny supports concrete and `abstract` modules. Abstract modules can contain lemmas and functions without bodies. Dafny does not attempt to prove bodyless lemmas, so these lemmas function as axioms within that module. Abstract modules can also contain opaque type definitions, e.g. `type T`.

Modules can be refined by other modules: by providing bodies for any bodyless lemmas and functions; by providing concrete datatype definitions for opaque type definitions; or by adding additional definitions. Concrete modules must not contain any bodyless lemmas or opaque type definitions. Thus making the bottom-most module in a series of refinements concrete ensures that no bodies are missing.

Within a module, Dafny normally reveals function bodies some number of times at the call site [16][4]. Dafny does not reveal functions bodies across module boundaries. Thus modules are a useful abstraction boundary, where we can be sure that a proof does not depend on some implementation detail of a function. Modules also seems to offer some performance advantages: hiding irrelevant definitions sometimes seems to help the verifier work faster.

In an abstract module, we often choose to split the meaning of a bodyless predicate or function from its definition by defining it with no postcondition and giving the postconditions as separate axioms (bodyless lemmas) that we expect to follow from the concrete implementation of the predicate. This split allows us to use the predicate in contracts or in assertions without making the whole postcondition of the predicate available. We can selectively reveal, what is in effect, the exact part of the postcondition that we need by calling the

[4] although this automation can be restricted by marking functions with the annotation `:opaque true`, and selectively revealing the definitions by writing `reveal_functionname()`

relevant axiom. In a refining module we are obliged to provide bodies and prove that the lemmas hold for the concrete implementation, but Dafny can often do this automatically.

Listing 7 shows such a bodyless predicate and bodyless lemma. We use the predicate `Wf` throughout our proof to require that execution produces states that are well-formed in some sense. In the listing one of the axioms for `Wf` is shown. The axiom says that if a state is `Wf` then each stack frame has a `this` pointer.

```
predicate Wf(s:SC)

lemma wfImpliesThisDefined(s:SC)
  requires Wf(s);
  ensures ∀d :: inRangeStackDepth(s, d)⟹
    local(s,d,This).Some?
```

Listing 7. Abstract well-formedness predicate on states, illustrating separating of the predicate postcondition into axioms

6.3 Modules and Refinement Structure of our Proof

Our strategy is to create abstract modules to act as interfaces between the various parts of our system. This strategy has four main effects. Firstly, it splits the proof so that we do not have to always wait for the verifier to verify everything all at once. Secondly, it prevents knowledge of irrelevant implementation details affecting the verifier performance. Thirdly, it creates a clean separation of concerns helping us to understand more exactly which behaviour of the implementation is relevant to other parts of the proof. Fourthly, it potentially allows us to reuse parts of our proofs for different implementations. For example, should we be interested in a different language semantics we should be able to reuse some parts of our proof.

Figure 6 shows how our proof is structured into modules. The module `Auxiliary` is not shown as it is imported by all other modules. It contains some auxiliary lemmas about the built in Dafny datatypes set, map and sequence.

Module `TheoremProof` contains a proof of theorem 4.3 using the definitions from `AbstractLts` and `AbstractEquiv`. The module is 80 lines long and proceeds by induction over traces from the abstract labelled transition system transition relation. Module `AbstractLts` is around 90 lines long and contains the definition of abstract labelled transition system, very similar to how it is given in definition 4.1. Module `AbstractEquiv` is 30 lines long and contains the definition of our abstract labelled transition system equivalence, similar to how it is given in definition 4.2

Module `AbstractLang` contains an abstract definition of the language $\mathcal{L}_{\text{ver}}$ and its transition relation, and is around 1200 lines long. It mostly contains predicates that relate consecutive execution states. For example, Listing 8 shows a predicate that relates state `s1` to state `s2`, designed to capture the effect of `s1` executing a *return* to produce state `s2`. We use module `AbstractLang` to abstract the concrete representation of $\mathcal{L}_{\text{ver}}$ states, and to hide the concrete operational semantics which are contained in module `Lang`. Section 6.4 covers this in more detail.

```
predicate
    stacksEqualApartFromOperandsAndRemovedFrame
      (s1:SC, s2:SC)
  requires Wf(s1);
  ensures
   stacksEqualApartFromOperandsAndRemovedFrame(s1,s2)
   ⟹topCont(s1).Some?
   ∧topCont(s1).o = []
   ∧depth(s1)-1 = depth(s2)
   ∧equalLocals(s2, s1)
   ∧(∀d :: inRangeStackDepth(s2, d)⟹
        d ≠ depth(s2)-1⟹equalOperandsD(s1,s2,d))
   ∧equalMeth(s2, s1)
   ∧equalCont(s2, s1)
   ∧operands(s1, depth(s2)-1).Some?
   ∧operands(s2, depth(s2)-1).Some?
   ∧operands(s2, depth(s2)-1).o ≠ []
   ∧topOperands(s1).Some?
   ∧topOperands(s1).o ≠ []
   ∧operands(s1, depth(s2)-1).o = =
        [topOperands(s1).o[0]] +
        operands(s2, depth(s2)-1).o;
```

Listing 8. Dafny predicate describing the relation between a state `s1` that executes a return to produce state `s2`

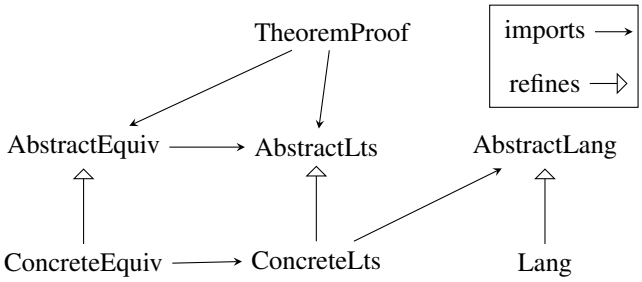

Figure 6. Module structure of our automatically checked proof.

Module `ConcreteLts` is a refinement of module `AbstractLts`. It contains the definition of U and is around 1200 lines long. Its primary role is to prove lemma 5.3, and squash multiple sequential steps from affected states into a single step. Specifically, it proves by refinement that an abstract labelled transition system transition relation can be constructed from the $\mathcal{L}_{\text{ver}}$ transition relation given in module `AbstractLang`.

Module `ConcreteEquiv` gives concrete equivalences for $\mathcal{L}_{\text{ver}}$ and is around 2500 lines long. Its primary role is to prove lemma 5.7, that the equivalences are preserved by execution. To do this, given some pair of equivalent states, we use the predicates from module `AbstractLang` to pick a relation α such that the successor states are also equivalent.

6.4 Partial Functions

We use Dafny's built in datatypes to represent the concrete states of our language $\mathcal{L}_{\text{ver}}$, but we do most of the proof using an abstraction of this representation. The abstraction is expressed using functions and predicates. To illustrate

we will describe our representation of heaps, but we did similarly for the stack.

Listing 9 shows how maps represent the heap and objects in a natural way. It was difficult to prove some of our complex lemmas using this representation directly. In particular we needed to prove that given two heaps which differ in the unaffected objects, that after arbitrary changes to the affected objects there remains the same difference in the unaffected objects. When using a map representation directly we continually experienced verification timeouts.

We created accessor functions around the maps. These functions are defined in module `AbstractLang` and used by other modules. Function definitions are not revealed over module boundaries so this hides the concrete representation of the heap whilst proving the lemmas in those modules. The accessor functions are implemented in module `Lang` where we discharge the obligation to prove their properties.

Listing 10 shows the functions we use to abstract our heap representation. Since Dafny maps are partial, we use a datatype `Option` to encode partial functions as total functions. The value `Some` is used for parameters that are in the domain, and `None` otherwise. For example, function `KlassOf(s,a)` returns the class of the object at address `a` in state `s` wrapped in `Some` iff address `a` is in the heap of `s`, and will return `None` otherwise.

This design makes it easy to assert certain properties that are important for our proof. The reason for this simplicity is that `None` is equal to itself. The code in Listing 11 asserts that two states have equal heaps. Note that we do not have to quantify over existent addresses and fields.

```
  datatype Heap =
    createHeap(objs:map<Addr, Obj>, next:int)
  datatype Obj =
    createObj(klass:Klass, fields:map<Field, Addr>)
```

Listing 9. Concrete representation of $\mathcal{L}_{\text{ver}}$heaps in Dafny, taken from module `Lang`

```
datatype Option<O> = Some(o:O) | None
function heap(s:SC, a:Addr, f:Field)
   : Option<Addr>
function KlassOf(s:SC, a:Addr)
   : Option<Klass>
predicate inHeap(a:Addr, s:SC)
  ensures inHeap(a,s) ⟺ KlassOf(s,a).Some?;
```

Listing 10. Abstracted representation of $\mathcal{L}_{\text{ver}}$heaps in Dafny, taken from module `AbstractLang`

```
assert (∀a,f :: heap(s1,a,f) = heap(s2,a,f)) ∧
       (∀a :: KlassOf(s1,a) = KlassOf(s2,a));
```

Listing 11. Asserting that two $\mathcal{L}_{\text{ver}}$ states have equal heaps in Dafny

6.5 Termination Preproofs

It is possible to write non-terminating programs in $\mathcal{L}_{\text{ver}}$. So when squashing a series of affected steps as in definition 5.4, we have to consider the case where execution continues forever, and each subsequently reached state is an affected state. We cannot construct an abstract transition from such an affected state to a successor unaffected state, because there is no such successor.

Our abstract labelled transition system must only squash finite series of affected steps which reach an unaffected state. However, Dafny insists that we prove termination for all parts of our proof. So we cannot simply write a function that executes from an affected state in the hope of reaching an unaffected one. Instead we must first produce a witness, or *preproof*, that executing from the state we are interested in does reach an unaffected state or terminate in a finite number of steps.

We use a sequence of the states reached during execution as our preproof. Because Dafny's built in sequences are always finite, if such a sequence of states exists then we know it is finite. Listing 12 shows the predicate `CheckReachesUOrStuck` over states and sequences of states that we use to identify a preproof. Our predicate holds if execution from the state proceeds through the steps in the sequence eventually terminating (i.e. stuck) or reaching an unaffected state. We pick out terminating states using the predicate `ReachesUOrStuck` which holds only if there exists such a preproof.

```
predicate CheckReachesUOrStuck(s:S, preproof:seq<S>)
  decreases preproof;
  requires Lang.Wf(s.sc);
{
  if preproof = [] then
    U(s)
  else
    preproof[0] = s ∧ !U(s) ∧
    var t := ExpandedStep(s);
    var tail := preproof[1..];
    if t.Stuck? then tail = []
    else
      t.GoesTo? ∧
      CheckReachesUOrStuck(t.s, tail)
}

predicate ReachesUOrStuck(s:S)
  requires Lang.Wf(s.sc);
{
   ∃preproof ::
     CheckReachesUOrStuck(s, preproof)
}

function Preproof(s:S) : Lang.Option<seq<S>>
  requires Lang.Wf(s.sc);
  ensures Preproof(s).Some?⟹
    CheckReachesUOrStuck(s, Preproof(s).o);
{
  if ReachesUOrStuck(s) then
    var preproof :| CheckReachesUOrStuck(s, preproof);
    Lang.Some(preproof)
  else Lang.None
}
```

Listing 12. Preproof predicate picks out terminiating executions

Finally we also use a function `Preproof` which returns an arbitrary preproof if one exists, or `None` otherwise. We use this function to help us neatly write conditions that distinguish terminating and non-terminating states.

We also found that we experienced fewer verifier timeouts when producing preproofs using a function, rather than directly via the let-such-that `:|` operator[5]. We hypothesise that this is because using the function we get the same, arbitrary, preproof for each state rather than a potentially different preproof each time we use the `:|` operator. Hence, once we have proved a property using the preproof we can later easily assert that property.

To prove determinism of execution in the unaffected part we also prove that at most one such preproof exists for any affected state.

Whenever we want to write a recursive function on such a series of steps from affected states we have to produce a preproof to show termination. Listing 13 shows such a pattern. Function `MyFn` uses the function `Preproof` to obtain a preproof if one exists. If it obtains one then it calls the recursive function `MyFn'`, otherwise it returns the value `None`.

```
function MyFn(s:S) : Option<S>
{
  var preproof := Preproof(s);
  if preproof.None? then None
  else Some(MyFn'(s, preproof.o))
}

function MyFn'(s:S, preproof:seq<S>) : S
  decreases preproof;
  requires CheckReachesUOrStuck(s, preproof);
{ ... }
```

Listing 13. Using a preproof

6.6 Termination Proofs and Finite Datatypes

We create a multi-step abstract labelled transition relation from the transitive reflexive closure of the abstract single step relation. We insist that multi-step execution between two states produces a sequence of labels, as shown in listing 14. Since Dafny sequences are finite, we can prove termination of inductive predicates and recursive functions over multi-step executions without any additional effort.

```
predicate Steps(s1:S, ls:seq<L>, s2:S)
```

Listing 14. Multi-step abstract labeled transitition relation

6.7 Flat Stack Equivalence

In definition 5.6 we gave a recursive definition of equivalence between stacks. We tried to use the same definition in our Dafny proof, but we found that our proofs became complex to write. We also experienced many verification timeouts. So instead we used a modified definition which instead of treating the stack as an inductively defined sequence, represents it directly as a partial function from integers to stack frames. We found that it was often easier for us to prove quantified assertions about such functions than to prove inductively defined assertions about recursive structures. And that we were less likely to run into verifier timeouts during development.

Given this definition of equivalence between stacks, we are able to represent the equivalence as maps between integers, which represent the depth of a stack frame. An additional benefit of this approach is that equivalence between stacks and equivalence between heaps have very similar definitions. Stacks are equivalent when there exists a bijective map between stack frame depths that preserves the structure of the stack. And two heaps are equivalent when there exists a bijective map between addresses that preserves the structure of the heap. Thus we are able to use lemmas that we prove about generic maps to deal with both parts of the state equivalence.

Listing 15 shows an example that asserts that pairs of mapped stack frames have the same continuation.

In parts of the proof dealing with method call and method return we have to reason about the top two stack frames. It seems that usually Dafny only reveals recursively defined predicates once [16]. When we used such recursively defined predicates for the stack equivalence we had to manually unroll the definitions to be able to reason about the second stack frame. Using our flattened stack equivalence avoided this.

```
var isos:map<int,int>; var s1,s2:S;
assert ∀d :: d ∈ isos ⟹
   cont(s1.sc,d) = cont(s2.sc,isos[d])

function cont(s:SC, d:int)
        : Option<seq<Expr>>
```

Listing 15. Asserting that each pair of stack frames mapped by the map `isos` have the same continuation. Also shown is the `cont(s,d)` function which returns the continuation for the `d`th stack frame of state `s`

6.8 Interesting Auxiliary Lemmas

In the process of proving our theorem we had to prove some properties about the built-in Dafny datatypes set, sequence and map. Most of these lemmas were straightforward to prove. Some of them were quite trivial properties of the datatypes that Dafny does not already know. Dafny offers some neat approaches to writing such proofs. Dafny lemmas are ghost methods, so can contain loops and other imperative control structures. Lemmas being methods enables us to prove a property of maps by: writing a method with a map parameter and the property as its post condition; implementing the method to calculate the property; and then proving that the implementation does indeed entail the postcondition.

[5] deterministic ghost operator, gives an arbitrary value satisfying a predicate

An illustrative example of this approach is shown in listing 16, where we prove that injective maps have an inverse. We have only shown the bodies of the most important lemmas. Lemma `injectiveMapHasInverse` proves that the inverse exists by calling lemma `canInvertMap`. We also provide a function `invert` which can be used to obtain the inverse in other lemmas and contracts: it first proves that the inverse exists, and then uses the "let such that" operator `:|` to return the inverse map.

The main body of the proof is in the lemma `canInvertMap` . The lemma returns a value `m'` which should be read as existentially quantified, whereas the parameter of the lemma should be read as universally quantified, i.e. $\forall m \exists m'$. Dafny maps are immutable. The lemma is implemented by: first copy the input map and call it `R`; then loop, at each iteration remove one key↦value from `R` and put it into a map called `S`, also put the inverse value↦key into a map called I. We maintain the loop invariant that `I` is the inverse of `S`, and at the end of the loop all elements of `m'` have been copied to `S` so `I` is the inverse of `m'`.

6.9 Editing and IDE

We did most of our proof development using the Dafny IDE plugin for the Microsoft Visual Studio IDE. We found the IDE very useful. The incremental verification and relatively rapid feedback it provides make it fun and relatively responsive to use. But delays waiting for the verifier, still make the experience frustrating at times, see section 6.1 for more.

We encountered two further difficulties. Firstly, the IDE would become unresponsive after some hours of use and need to be restarted in order to continue. Secondly, when opening a Dafny file for editing in the IDE the whole file has to be verified. For our large files this takes around 10 minutes per file.

To counter the problem we used the Dafny `:verify false` annotation. This annotation tells Dafny to skip verification of the particular lemma or function. We then only enable verification on the lemmas we are currently changing. After completing the proof we remove all annotations and run the Dafny command line verifier to ensure that using the annotation did not allow any errors through.

7. Conclusion

We have described a method for establishing the existence of equivalences between some parts of the behaviour of two versions of a program. Our method does not require us to examine the stack and heap at each execution step. Instead we are able to establish if the versions are equivalent by examining only some of the method calls from execution traces. We have automatically checked a proof that our method is sound using the Dafny program verifier. We detailed here some of our experiences with and techniques for using Dafny.

Related: Regression verification uses bounded model checking to verify the equivalence of some loop and re-

```
predicate mapInjective<U,V>(m: map<U,V>)
  ensures mapInjective(m) ⟺ ∀a,b ::
    a ∈ m ∧ b ∈ m⟹a ≠ b⟹m[a] ≠ m[b];
  ensures mapInjective(m)⟹∀a,b ::
    a ∈ m ∧ b ∈ m⟹m[a] = m[b]⟹a = b;

predicate mapsAreInverse<U,V>
      (m: map<U,V>, m': map<V,U>)
   ensures mapsAreInverse(m,m')⟹∀a ::
    a ∈ m⟹m[a] ∈ m' ∧ m'[m[a]] = a;
   ensures mapsAreInverse(m,m')⟹∀a ::
    a ∈ m'⟹m'[a] ∈ m ∧ m[m'[a]] = a;

lemma canInvertMap<U,V>(m: map<U,V>)
    returns (m': map<V,U>)
  requires mapInjective(m);
  ensures mapsAreInverse(m,m');
{
  var R,S,I := m,map[],map[];
  while R ≠ map[]
   decreases R;
   invariant mapSmaller(R, m);
   invariant mapSmaller(S, m);
   invariant R !! S; // disjoint
   invariant m = union(R, S);
   invariant mapsAreInverse(S,I);
  {
    var a :| a ∈ R;
    var v := R[a];
    var r := map i |
       i ∈ R ∧ i ≠ a :: R[i];
    R,S,I := r,S[a:=v],I[v:=a];
  }
  m' := I;
}

lemma injectiveMapHasInverse<U,V>(m: map<U,V>)
  requires mapInjective(m);
  ensures ∃m' :: mapsAreInverse(m, m');
{
  var m' := canInvertMap(m);
}

function invert<U,V>(m:map<U,V>) : map<V,U>
  requires mapInjective(m);
  ensures mapsAreInverse(m,invert(m));
{
  injectiveMapHasInverse(m);
  var m' :| mapsAreInverse(m,m');
  m'
}

predicate mapSmaller<U,V>(m: map<U,V>, m': map<U,V>)
  ensures mapSmaller(m,m')⟹
    (∀u :: u ∈ domain(m)⟹u ∈ domain(m'));
  ensures mapSmaller(m,m')⟺
    (∀a :: a ∈ m⟹a ∈ m' ∧ m[a] = m'[a]);
function domain<U,V>(m:map<U,V>) : set<U>
  ensures domain(m) = set u : U | u ∈ m :: u;
  ensures ∀u :: u ∈ domain(m)⟹u ∈ m;
function union<U, V>(m: map<U,V>, m': map<U,V>):
    map<U,V>
  requires m !! m';
  ensures ∀i ::
    i ∈ union(m, m') ⟺ i ∈ m ∨ i ∈ m';
  ensures ∀i ::
    i ∈ m⟹union(m, m')[i] = m[i];
  ensures ∀i ::
    i ∈ m'⟹union(m, m')[i] = m'[i];
  ensures |m| + |m'| = |union(m, m')|;
lemma sizeOfDomainIsSizeOfMap<U,V>
    (m:map<U,V>, s:set<U>)
  requires s = domain(m);
  ensures |s| = |m|;
```

Listing 16. Proof that injective maps have an inverse

cursion free programs [7]. Program verification tools in conjunction with automated theorem provers can be used to check some programs for equivalence [8, 9]. These tools are modular and fast but currently do not work well with modifications that result in different heaps states across many method calls. Provers can also be used to check library versions for backward compatibility [10]. Semantics aware trace analysis [11] and BCT [12] use traces captured at runtime to attempt to isolate the causes of regression failures. Guru [13] uses sequences of message sends to define equivalence of differently factored method implementations in an object-oriented system. State based encapsulation provides a method for reasoning about the equivalence of individual classes [14]. Our programs are deterministic so labelled transition system trace equivalence suffices for our purpose, but to extend our approach to the non-deterministic concurrent case it might be interesting to consider using the Weak Bismulation up to [17] technique.

Future: We are using our theorem as the basis for a tool. The tool is currently able to find automatically that V1 and V2 from section 2 are *not* equivalent wrt. affected objects of class `Logger`, `PrintStream`, `List` and `Student`. We plan to describe and develop the tool further in future work.

Acknowledgments

K. Rustan M. Leino for discussions and extensive help with Dafny including a new version of Dafny with some improvements to module refinement. Reuben Rowe for discussions about our proof. Nada Amin for the very helpful tutorial video *How to write your next POPL paper in Dafny*, which included the `Option` technique for encoding partial functions.

References

[1] Yin, Zuoning and Yuan, Ding and Zhou, Yuanyuan and Pasupathy, Shankar and Bairavasundaram, Lakshmi. 2011. How do fixes become bugs?. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

[2] Jihun Park and Miryung Kim and Ray, B. and Doo-Hwan Bae. 2012. An empirical study of supplementary bug fixes. Mining Software Repositories (MSR), 9th IEEE Working Conference on.

[3] Lahiri, Shuvendu K. and Vaswani, Kapil and Hoare, C. A. R.. 2010. Differential static analysis: opportunities, applications, and challenges. Proceedings of the FSE/SDP workshop on Future of software engineering research.

[4] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning.

[5] K. Rustan M. Leino. 2008. This is Boogie 2.

[6] Christakis, Maria and Leino, K. Rustan M. and Schulte, Wolfram. 2014. Formalizing and Verifying a Modern Build Language. FM 2014: Formal Methods.

[7] Godlin, B. and Strichman, O.. 2009. Regression verification. Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE.

[8] Hawblitzel, Chris and Kawaguchi, Ming and Lahiri, Shuvendu K and Rebêlo, Henrique. 2013. Towards modularly comparing programs using automated theorem provers. International Conference on Automated Deduction.

[9] Shuvendu Lahiri and Ken McMillan and Rahul Sharma and Chris Hawblitzel. 2013. Differential Assertion Checking. Foundations of Software Engineering.

[10] Welsch, Yannick and Poetzsch-Heffter, Arnd. 2012. Verifying backwards compatibility of object-oriented libraries using Boogie. Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs.

[11] Hoffman, Kevin J. and Eugster, Patrick and Jagannathan, Suresh. 2009. Semantics-aware trace analysis. Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation.

[12] Mariani, Leonardo and Pastore, Fabrizio and Pezze, Mauro. 2011. Dynamic Analysis for Diagnosing Integration Faults. IEEE Trans. Softw. Eng..

[13] Moore, Ivan. 1996. Automatic inheritance hierarchy restructuring and method refactoring. Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.

[14] David Naumann and Anindya Banerjee. 2012. State Based Encapsulation for Modular Reasoning about Behaviour-Preserving Refactorings. Aliasing in Object-oriented Programming. Springer State-of-the-art Surveys.

[15] Tim Wood and Sophia Drossopoulou. 2013. Refactoring Boundary. Imperial College Computing Student Workshop

[16] Amin, Nada and Leino, K. Rustan M. and Rompf, Tiark. 2014. Computing with an SMT Solver. Proceedings of the 8th International Conference on Tests and Proofs

[17] Sangiorgi, Davide and Milner, Robin. 1992. The problem of "Weak Bisimulation up to". CONCUR'92