Using Sub-cases to Improve Log-based Oracles Inference

I.S.W.B. Prasetya

J. Hage

A. Elyasov

Technical Report UU-CS-2012-012
Sept. 2012

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Using Sub-cases to Improve Log-based Oracles Inference

I.S.W.B. Prasetya J. Hage A. Elyasov
Dept. of Inf. and Comp. Sciences, Utrecht Univ.
email: {S.W.B.Prasetya,J.Hage,A.Elyasov}@uu.nl

***Abstract*—The reverse engineering of test-oracles is a notoriously hard problem. A popular approach is to use a tool like Daikon to dynamically infer pre- and post-conditions from execution traces (logs). Unfortunately, the underlying expressiveness of Daikon is very limited, so it cannot infer complex specifications. In this paper, we use sub-cases to circumvent the restriction. For a given target method, this paper proposes to first divide it into sub-cases. A sub-case may represent a complex subset of the method's executions, but without necessarily requiring a complex formula to express it. E.g. we use visits (or visit-patterns, if we want to be more general) over the method's control flow nodes to identify sub-cases. For each sub-case, pre- and post-conditions are inferred with Daikon. Daikon's limitations still apply, but more useful oracles can be inferred, as we shall illustrate by a case study.**

***Keywords*-Oracle Inference, Specification Inference, Specification Mining**

I. Introduction

There has been a lot of research on test-case generation (see, e.g., [1], [2], [3]). The focus of these works is on the generation of test-inputs. But for every test-case we also need to establish so-called 'oracles'. An *oracle* is a predicate to judge whether the execution of a test-case meets our expectations. For example, it can verify that the value returned by the tested program equals some expected value, or it can call a boolean method that essentially represents the program's specification. The latter approach is particularly attractive, because a single specification can be shared by multiple test-cases, but unfortunately in practice people rarely write such specifications.

To some extent oracles can be reverse engineered from a program, but only if the program can be reasonably assumed to be correct. The resulting oracles are therefore not helpful for testing the program. Instead, they are used for future regression tests. Oracles can be inferred both statically or dynamically [4], [5], [6], each with its own strengths and weaknesses. Our approach is largely dynamic. Dynamic inference is usually done by analyzing data, e.g. logs, generated at run time. Many modern applications already employ logging. E.g. web servers produce access logs, embedded software produce operational logs, database servers produce transaction logs, and operating systems produce system logs (usually to log device changes and operations as well as security related events).

Tools like Daikon [4] are used to infer oracles from logs. If the events in the logs can be treated or encoded as method calls, and translated to Daikon's format, Daikon can infer class invariants, pre-conditions, and post-conditions. However, Daikon's strength is limited. First of all, it has a restricted vocabulary (the kinds of basic terms it recognizes). Second, even if we stay within that vocabulary, a post-condition like $\mathbf{this}.x{>}0 \vee (q_1 \wedge q_2)$ cannot be inferred. In general, Daikon cannot infer disjunctive predicates [7], [8], because to do so requires all possible disjunctions to be inspected and the number of possibilities grows exponentially. One way to circumvent this problem is to use splitters.

A *splitter* is a predicate used to partition log entries. Let m be the method in question. In the above example $\mathbf{this}.x{>}0$ can be used as the splitter: Daikon then puts all log entries that correspond to m's calls that violate $\mathbf{this}.x{>}0$ in a set L_1 and the others in L_2. Any oracle q' inferred from L_1 implies $\mathbf{this}.x{>}0 \vee q'$ as an oracle for m. However, this approach requires the split predicate $\mathbf{this}.x{>}0$ to be known upfront, either by figuring it out manually or by calculating it statically [8]. Static calculation boils down to either calculating the weakest pre-condition or the strongest post-condition with respect to some easier to formulate scenarios, e.g. scenarios that visit only certain branches in m. In practice this may be difficult to do, because of loops, recursion, and calls to modules of which the source code is not available.

In our approach, we use what we call 'sub-cases' to specify scenarios within m. A sub-case is very similar to a splitter, but differs from traditional splitters in that it is a predicate over log segments, rather than a predicate over log entries. In particular, we want to express visit patterns through log-points within m. For example, we can express the scenario where the execution of m passes a certain branch, or express the scenario where m calls $openfile$ and $closefile$ alternately. Simple visit patterns are often able to express meaningful scenarios. In contrast, the corresponding traditional splitter, which is either the weakest pre-condition or the strongest post-condition of the scenario, can be a complex predicate. Or worse, cannot be calculated as pointed out above. Sub-cases are thus suitable for a purely dynamic approach. The price we pay is that deeper logging is required to give clues about what happens inside m, so that sub-cases can be identified. But the deep logging can be kept light, e.g. state serialization is not necessary.

We have implemented our approach to infer sub-case oracles for ActionScript programs. ActionScript is a popular

object oriented language to write browser-based programs. In the current implementation we limit ourselves to sub-cases expressed in terms of visits to nodes in the target method's control flow graph. As a case study, we applied the approach to infer oracles for a number of methods from a retro-game called Asteroid. Game testing is notoriously difficult. What appears to be a simple game can turn out to involve quite complex positions calculation. It may contain many scenarios that are hard to reconstruct programmatically, so that the only feasible way to test the game is by actually playing it. A further consequence of this is that the role of oracle is often taken by the tester himself, who has to visually judge whether the game behaves correctly. In such a setup, oracle inference is even more useful.

A. Contribution

The most important contributions are: (1) the idea of using visit patterns as a simple yet powerful way to strengthen Daikon-based oracles inference, (2) an implementation of the approach, and (3) a theory that provides a general, formal underpinning of the idea, —useful if one wants to re-implement the idea in a different setting.

B. Sections overview

To give a general idea of how our approach is used, Section III briefly explains the architecture of our implementation. Section IV provides the theory so that we can talk about relevant concepts more precisely. It defines precisely what vehavior 'case' and 'sub-case' oracles are. Section V briefly discusses how to encode log entries in the Daikon format. Section VI discusses the use of control flow nodes to express sub-cases of methods. Section VII discusses our case study. Section VIII considers some related work and Section IX concludes.

II. Motivating Example

To first illustrate our idea, consider the following simple method `Update`, of some class C:

```
1 Update(){
2   x++ ;
3   if (this.x <= 0 )
4       this.x=0 ;
5   if (this.x == 11)
6       this.x=10 ;
7 }
```

Let x stand for **this**.x. The intended pre-condition is $x \leq 10$, and the post-condition is:

$$\begin{array}{lll} & (old(x) = -1 & \wedge \quad x=0) \\ \vee & (old(x)=10 & \wedge \quad x=10) \\ \vee & (0 \leq old(x) < 10 & \wedge \quad x=old(x)+1) \end{array}$$

Although each atom above is in Daikon's vocabulary, the whole specification cannot be inferred as Daikon does not infer disjunctions. If it was to do so, it would have to check various candidate disjunctions. The number of possibilities is exponential.

Our idea is to split `Update` into 'sub-cases', and derive oracles of the sub-cases instead. Let us define sub-cases $\texttt{Update}_4$ and $\texttt{Update}_6$ to stand for the sets of `Update`'s executions that visit the branch at line 4 respectively 6; and the sub-case $\texttt{Update}_{3,5,7}$ represents executions that visit lines 3,5,7, and skipping both then-branches. These sub-cases fully partition the executions of `Update`. Each disjuncts in the above specification in fact corresponds to one of these sub-cases. If we treat the sub-cases as programs, these are their intended specifications:

sub-case	pre-cond	post-cond
$\texttt{Update}_4$	$x = -1$	$x = 0$
$\texttt{Update}_6$	$x = 10$	$x = 10$
$\texttt{Update}_{3,5,7}$	$0 \leq x < 10$	$x = old(x) + 1$

These sub-cases specifications, including the pre-conditions, can all be inferred with Daikon. Note that using traditional splitters we will have to calculate e.g. these pre-conditions ourselves as we need them to be used as splitters. If for example the method above calls `foo`() at the beginning, whose source code is not available, static analysis to calculate them will not work.

For the above idea to work, the logs do need to contain sufficient information: we must be able to indentify which log segments belong to which sub-cases. We do this by logging control flow nodes. In general any form of deep logging will do, which then determines what kind of sub-cases we can express.

In a more complex situation, the intended specifications of some sub-cases can be disjunctive, which are again problematical. In theory we can still split a sub-case E into smaller ones, as long as they are expressible in term of the information exposed by the logs. But even if we do not do so, the sub-case itself is a specification, as it specifies a subset of possible executions. For pre or post-condition of the form $p \wedge (q_1 \vee q_2 ...)$ we will lose q_k's, but may still be able to infer p. Although weaker, in conjunction with E we may still get a useful oracle. In any case, it is still better than getting no oracle at all, which was the case if we just directly use Daikon.

III. Architecture

Figure 1 shows the global architecture of our implementation. Given a target application App, we specify which methods are to be logged. Using our ABCI bytecode instrumenter, the neccessary logging code is injected. The entry and exit points of those methods will be logged, and also locations that correspond to the start of the nodes in their control flow graphs. At the entry and exit points, the parameters of the method, the state of the receiver object, and the return value are serialized and written to the log. To keep the overhead reasonable, nodes do not perform any object serialization.

The logger is an instance of the class `FittestLogger`, which is then attached to App'. `FittestLogger` has a number of nice features. It has a powerful way to specify how instances of classes of interest are to be serialized. It produces compact but deeply structured XML-like logs; the logs can also be collected in a distributed fashion by a remote server.

Then we use another one of our tools, `haslog`, to convert FITTEST logs to Daikon logs, and, more importantly, to re-group the logs into sub-cases. Then we use Daikon to to infer oracles for the sub-cases.

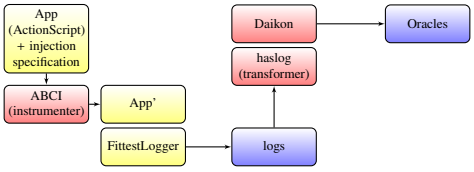

Figure 1. The architecture of our sub-cases oracles inference system.

ABCI, `FittestLogger`, and `haslog` are products of an ongoing project and part of the FITTEST testing framework (see [9] for more information).

IV. Theory

This section formally defines the concepts of 'behavior cases', 'sub-cases', and their oracles. Importantly, these will be fully defined in terms of logs, so that they can be used in pure log-based inference. Although in our case study we restrict ourselves to methods as 'behavior cases', and their visits to the nodes in their control flow graphs as 'sub cases', the concepts are more general than that. Instantiations different from ours are thus possible. We restrict ourselves to Hoare triple-like oracles, because they can be mapped directly to Daikon inference.

Concretely, a log ℓ of a program P is a sequence of entries, written in some format. Each entry provides information about some event of interest that occurs during an execution of P. Abstractly, we model ℓ as a sequence of events. Each event e in ℓ is assumed to be described like this:

$$e = \langle \eta(x), s \rangle \quad (1)$$

where η is the type/name of the event, s represents the relevant part of P's state when the event occurs, and x is the event's parameter. For simplicity events are assumed to have at most one parameter. Our implementation can deal with multiple parameters. The x and s components are optional, and the information they contain depends on the event type.

For example if e represents a user clicking on a GUI button, then s may represent the state of the button, and x specifies which mouse button was clicked. If e represents the entrance to a method m, s may represent the state of m's receiver object, and x is m's parameter.

For now we assume P to contain no recursion; this is handled later in Section VI.

A behavior case characterizes log segments whose start and end are signalized by certain events:

Definition 1: A *behavior case* is a pair C of event-types (η_E, η_X). An *instance* of C is a log segment that starts with η_E and ends with η_X. All events between them should be of other types. □

E.g. a behavior case can be used to identify log segments generated by a method m inside P if we log m's entry and exit points; these events then identify those segments. Note however that the concept itself is very general. Any set of segments can be turned to a behavior case by uniquely marking their start and end with special events.

The definition implies that instances of the same case cannot overlap. Instances of different cases can in theory overlap.

Some notations. $\mathcal{E}$ denotes the universe of all possible events. Logs and log segments are thus members of $\mathcal{E}^*$. $\mathcal{N}$ denotes the universe of event-types. For $\ell \in \mathcal{E}^*$, $\eta(\ell)$ denotes the corresponding sequence of event-types. If $\sigma = [u_0, u_1, ..., u_{k-2}, u_{k-1}]$, with $k \geq 2$, $\mathsf{mid}(\sigma)$ denotes the middle part $[u_1, ..., u_{k-2}]$. □

Oracles are predicates over $\mathcal{E}^*$, with the following concept of validity:

Definition 2: An oracle o is valid on P if it is valid on any log generated by P. □

Definition 3: A *case oracle* of a behavior case $C = (\eta_E, \eta_X)$ is an expression of the form:

$$p \Leftarrow \boxed{s \rightarrow \eta_E(x), \eta_X(y) \rightarrow s'} \Rightarrow q$$

where x, y, s, s' are free variables, and p, q are predicates. Such an oracle is valid on a log ℓ if it is valid on all all instances of C (in ℓ). It is valid on an instance: $[(\eta_E(x), s), ..., (\eta_X(y), s')]$ if p holds on (x, s) and q holds on (y, s'). □

If we want to be able to relates the exit and entry states in q, we can extend q to be a predicate over (x, s, y, s').

Definition 4: The p and q above are called respectively *past*- and *post*-conditions (we will explain why we avoid the term pre-condition). □

For example, suppose we have a behavior case specified by events of types E and X as its start and respectively end. The following oracle says that the behavior increases the value of its state s with d:

$$true \Leftarrow \boxed{s \rightarrow E(d), X() \rightarrow s'} \Rightarrow s'{=}s{+}d$$

Importantly, case oracles can be straightforwardly inferred using Daikon, provided p and q are made of conjunctions of atoms within Daikon's vocabulary. The encoding to Daikon logs is explained in Section V.

The roles of p and q in case oracles are analogous to pre- and post-conditions in Hoare triple, but we caution the reader of the following subtle difference. A Hoare triple e.g.

$\{P\}\ S\ \{Q\}$ means that *if* S is executed on P, then it will end in Q. So, the pre-condition has the role of an *assumption*. Our inference approach uses Daikon as the back-end. Daikon only infers based on the logs it sees; it does not speculate on logs it does not see. Consequently, it cannot actually infer assumptive pre-conditions. If the above specification is inferred by Daikon, it would instead mean that whenever S is executed, P would hold at the start, and Q at the end. So, both P and Q take the role of conclusions, one past-time and the other future-time. For this reason we avoid re-using Hoare triple notation, and we call p past-condition rather than pre-condition.

Definition 5: A *sub-case* of a behavior case C is a pair $E = (C, \phi)$ where ϕ is a predicate over $\mathcal{E}^*$. Instances of E are instances σ of C such that $\mathsf{mid}(\sigma)$ satisfies ϕ. □

Notice that a sub-case essentially specifies a subset of a case. We can split a case into a set of sub-cases. Rather than trying to infer case oracles we can thus, as hinted in Section I, try to infer sub-case oracles instead.

Definition 6: A *sub-case oracle* of a sub-case $E = (C, \phi)$ is an expression of this form (which looks almost like a case oracle):

$$p \Leftarrow \boxed{s \rightarrow \eta_E(x), \boxed{\phi}, \eta_X(y) \rightarrow s'} \Rightarrow q$$

It is valid on a log ℓ if it is valid on all instances of E. It is valid on an instance σ if the corresponding case oracle (the same expression as above, but without ϕ) is valid on σ. □

Some notations. If $g{:}U{\rightarrow}V$ and $f : \mathsf{set}(V) \rightarrow \mathsf{set}(W)$, the lifted composition is: $(f \circ g)(U) = f\ \{g(u) \mid u{\in}U\}$. □

A *proto-inferencer* is a function F that takes a set of logs as input and returns a set of oracles.

Definition 7: An *inference procedure* is a proto-inferencer F such that for all $o \in F(L)$ and for all $\ell{\in}L$, o is a valid oracle on ℓ. □

Definition 8: A *filtered inference procedure* is denoted by $F|f$, where $f : \mathcal{E}^*{\rightarrow}\mathcal{E}^*$ is called a 'filter', and F is proto-inferencer. They have the following property: for all $o \in (F \circ f)(L)$, o is a valid oracle on $f(\ell)$. □

A (filtered) inference procedure is an inference procedure for a behavior case C, if all oracles it produces are case oracles of C. Inference procedures for a sub-case E are defined analogously.

Theorem 1: Any inference procedure F for a behavior case C can be turned to a filtered inference procedure for a sub-case E of C. □

Proof: Let L be a set of logs, and $L' = \{f(\ell) | \ell \in L\}$, where $f(\ell)$ will filter ℓ by throwing away all instances of C which are not instances of E. Because F is an inference procedure for C, then for every $o \in F(L')$ and every $\ell' \in L'$, o is a valid oracle of C on ℓ'. Notice that $F(L') = (F \circ f)(L)$. So, $F|f$ is a *filtered* inference procedure for C. However, by the definition of f we have: for all $\ell \in L$, all instances of C in $f(\ell)$ are actually instances of E. So, any oracle o inferred by $F|f$ is also an oracle of E. □

We have said that case oracles can be inferred with Daikon. By the theorem above, sub-case oracles can also be inferred with Daikon. However, a filter f is required to do this (see the proof) to identify log segments that match the given sub-case E. This filter is basically an implementation of the ϕ part of E. Notice that ϕ is a predicate over $\mathcal{E}^*$. It can in theory be temporal, and checking it can be costly. For effeciency, we can choose to limit ϕ, e.g. by constraining it to be a regular expression over event types:

Definition 9: Let C be a behavior case and r a regular expression over the alphabet $\mathcal{N}$. The pair (C, r) is a *regular* behavior sub-case of E. It defines the sub-case (C, ϕ) where $\phi(\sigma)$ is true if $\eta(\sigma)$ is accepted by r, and else false. □

V. Daikon Encoding

This section briefly explains how to use Daikon to infer case oracles. By Theorem 1, it can then be used to infer sub-case oracles. The proof describes how this can be done.

Out of the box, Daikon is a tool to infer state predicates from 'execution traces' [4]. Traces are synonymous to logs, and 'events' are called 'program points' in Daikon. Various categories of program points are available, including the categories `ENTER` and `EXIT` to represent the entrance and exit of a method. Suppose m is a method of class C, whose entry and exit are logged in the Daikon format. The logging will generate program points of types m(..):::`ENTER` and m(..):::`EXIT`. Below is an example of a Daikon log (less interesting details are removed) showing two program points (events) representing a call $o.m$(3). The first one corresponds to the entrance to m, the second one to its exit:

```
m(..):::ENTER
arg1
3
this.x
1

m(..):::EXIT
arg1
3
this.x
4
```

Every program point may contain information about "variables", which can be used to represent either parameters (e.g. `arg1` above) of the corresponding event or state information, such as `this.x` above.

So, the above log says that on entrance of the call to $o.m$(3) the value of $o.x$ is 1, and on exit it becomes 4.

From a set of logs consisting of events of the above types, Daikon can infer the past- and post-conditions for m.

If $C = (\eta_E, \eta_X)$ is a behavior case, an inference procedure for C can be built as follows. Let L be the input set of logs. We translate L to Daikon logs. Only instances of C are needed, the rest can be thrown away (but we do not have to). In particular, events of type η_E should be mapped to method

entries $\eta(..)$:::ENTER and those of type η_X to $\eta(..)$::EXIT. The resulting past/post-conditions for $\eta(..)$ are case oracles for C.

In practice the translation to Daikon logs will require more work. For example, each program point must have the corresponding declaration. We have to declare its name and type, and the "variables" it constitute. Daikon also expect values of variables to be typed (int, bool, etc). So, if the used logger does not log this information, then the translator must try to guess it. The description of the Daikon syntax can be found in its documentation [10]. Daikon format is however rather verbose. Our implementation (Section III) produces logs in the so-called FITTEST format [9], our tool `haslog` (Figure 1) can translate them to Daikon.

VI. Specifying Sub-cases of Methods

Let P be an application, of which we want to infer the oracles of some methods in P. To do so, each target method m needs to be instrumented so that the corresponding behavior case (m_E, m_X) is defined, where m_E is the event representing the entrance of m, and m_X represents its exit. To allow sub-cases to be expressed, and thus oracles for them to be inferred, we also need to instrumentation within m's body.

Let G_m be m's control flow graph (CFG). Every node i in G_m is represented by a unique number and represents a 'block' of consecutive instructions in m's body that does not contain any jump or branching instruction except the last instruction in the block, and furthermore no instruction in the block is targeted by a jump or a branching instruction except for the the block's first instruction.

Notation. If r is a regular expression, we write $\diamond r$ to mean $\mathsf{any}^*\ r\ \mathsf{any}^*$. So, it matches a sequence σ that contains a segment that matches r. □

Our implementation instruments every $i \in G_m$ so that when it is visited it produces an event of type m_i. So, the regular sub-case:

$$((m_E, m_X), \diamond m_3)$$

represents thus the executions of m that visit the node 3. Let τ be a finite sequence event types induced by the nodes in G_m. The regular sub-case:

$$((m_E, m_X), \diamond \tau)$$

represents executions of m that visit all the nodes in τ (and in the same order). If $|\tau| = 2$, it corresponds to edge-visit. If $|\tau| = 3$, it corresponds to edge-pair-visit.

Definition 1 however requires that instances of the case $C = (m_E, m_X)$ should not overlap. This property is broken if m is recursive, which will result in nested instances of C. Our implementation pre-process the logs to first remove the nesting. For example consider the log below, that corresponds to the call $m(1)$ that in turn recursively calls $m(0)$:

$$\left[\begin{array}{l} \langle m_E(1), 10\rangle, \langle m_0(), ()\rangle, \\ \quad \langle m_E(0), 10\rangle, \langle m_0(), ()\rangle, \langle m_1(), ()\rangle, \langle m_X(0), 0\rangle, \\ \langle m_2(), ()\rangle, \langle m_X(1), 0\rangle \end{array}\right]$$

Our pre-processing will rewrite this to:

$$\left[\begin{array}{l} \langle m_E(1), 10\rangle, \langle m_0(), ()\rangle, \langle m_2(), ()\rangle, \langle m_X(1), 0\rangle \\ \langle m_E(0), 10\rangle, \langle m_0(), ()\rangle, \langle m_1(), ()\rangle, \langle m_X(0), 0\rangle, \end{array}\right]$$

Notice that the two instances of C are now no longer nested. Because a case oracle is a predicate over its instances (and not over sequences of these instances) the order of the instances in the resulting log above does not actually matter.

The pre-processing algorithm is shown below; given an input log ℓ, $regroup([], m, \ell)$ will produce a regrouped log ℓ'.

function REGROUP($stack, m, \ell$)
  if $empty(\ell)$ **then return** []
  $stack.push(\ell_0)$
  if $type\ell_0 = m_X$ **then**
    $z \leftarrow stack.popUntil(m_E)$
    return z ++ REGROUP($stack, m, tail(\ell)$)
  else return REGROUP($stack, m, tail(\ell)$)
  end if
end function

In some situations we may prefer to express subcases in terms of semantically more meaningful events than just node visits. For example, suppose the method m uses a file f. Assume first that we log m's calls to $open(f)$ and $close(f)$ so that they produce events of type f_O and respectively f_C. The regular sub-case:

$$((m_E, m_X), (f_O\ f_C)^*)$$

can be used to represent the executions of m that always closes f after opening it. Where as:

$$((m_E, m_X), (f_O\ f_C)^* f_O^+)$$

represents the executions of m that fails to close an open f before it exits.

However notice that subcases like these can be encoded as node visits by mapping f_0 and f_C to the corresponding nodes in m where they occur. The target program can be scanned to construct such a mapping; then our inference approach can be reused.

VII. Case Study

Our case study is a Flash game called Asteroid, written in ActionScript. Asteroid is an action 2D game in which a player controls a space ship to destroy asteroids. It consists of 15 classes, with in total 1780 lines of source code. The most complex class is `Game`, where the main part of the game logic is implemented. It is 702 lines, and has 18 methods. Testing games is notoriously difficult. They often contain

components that are hard to unit-test because they must be executed in the context of the used game framework, so that the only way to test their game logic is by playing them. This is also the case with our case study.

We set up an experiment to infer oracles for two methods: `Update`() of the class `Bullet` and `Update`() of the class `Game`. The first implements bullets' position calculation. The second implements the game's main logic. Both methods have no parameters, but their target objects have multiple fields. Figure 2 shows their statistics.

	#lines	*#nodes*	*McCabe*
`Bullet`'s `Update`	19	9	3
`Game`'s `Update`	145	79	49

Figure 2. *#nodes* is the number of nodes the method's CFG has, and *McCabe* is its cyclomatic complexity [11].

After instrumenting the two methods we play the game a number of times to produce 265 MB logs. The executions produce over 18 thousands call to `Game`'s `Update` and over 45 thousands call to `Bullet`'s `Update`. Despite the massive amount, they do not fully cover all 'scenarios', as some scenarios are simply very hard to expose in a manual game play (e.g. hitting the ESC button at the same moment as the last ship is destroyed). In any case, 100% edge coverage on both methods is achieved.

For the experiment the program is assumed to be correct. No specification for it exists unfortunately; so we manually checked the inferred oracles to see if they are correct or false positives.

A. `Bullet`'s `Update`

Figure 3 shows the method that calculates a bullet's new position.

```
1  class Bullet extends GameSprite {
2    var speed:Point;
3    var life:Number;
4    ...
5    public function Update():void {
6      x += speed.x;
7      y += speed.y;
8      if (x+width <= 0 ) x = W-width ;
9          else if(x >= W) x = 0;
10     if (y+height <= 0) y = H-heigth ;
11         else if(y >= H) y = 0;
12     super.Update();
13   }}
```

Figure 3.

The method moves the bullet further by its speed vector, but will wrap the bullet around the screen if it would otherwise pass beyond the screen borders. Although seemingly simple, the calculation has a number of nasty corner cases which are influenced by several other classes, such as the class `Game` that controls the starting positions of bullets. Such context sensitivity is quite typical in an OO program.

`Update` (45097 samples)

past-c:	$x \le 521.0$		(≤ 525)
	$x \ge -24.0$		(≥ -25)
	$y \le 399.0$		(≤ 400)
	$y \ge -20.0$		(≥ -25)
post-c:	$x' \le 499.99533810498167$	$\surd$	(< 500)
	$x' \ge -1.99999999999999$	$\surd$	(> -2)
	$y' \le 374.99238475781965$	$\surd$	(< 375)
	$y' \ge -1.9832464672030596$		(> -2)

Figure 4. Inferred case oracles for `Update`.

Figure 4 shows the *case oracles* about bullets' positions we infer from Daikon. The table induces oracles of the form:

$$p \Leftarrow \boxed{x, y \to \texttt{Update}_E(), \texttt{Update}_X() \to x', y'} \Rightarrow q \qquad (2)$$

where p and q are either *true* or a past-condition respectively post-condition from the table.

Unsurprisingly, only the overall upper and lower bounds of the x and y coordinates can be inferred. Note that they are *float* numbers. Those marked with $\surd$ are considered *acceptable*, assumming $\epsilon = 0.01$ accuracy. The bracketed values are the real upper/lower bounds.

Daikon's default algorithm to infer boundary values assume either uniform distribution of the values, or that values close to the bounds appear more frequently [12]. Neither assumption is true for bullets (values close to borders are less likely to occur). In this case it is more reasonable to just take the minimum/maximum observable values as the bounds, which we did in the experiment.

The inferred bounds in the past-conditions are noticably less accurate. During the training the screen size is fixed to 500×375. We first expected that valid bullets coordinates must lie between 0 and those values. Thanks to Daikon, we then realized that this is not true. Bullets can be created off the screen when the ship is shooting from a position just beyond a screen border, but not yet wrapped to the otherside. Maneuvering the ship to the furthest of such positions is very difficult, which explains the inaccuracy in the past-conditions.

It may be worth noting that the bounds in the past-conditions are just as hard to get with a static approach. We cannot infer them by just analyzing the class `Bullet` alone. E.g. the class `Game` and `Ship` also influence them. Additionally, the game's implicit top-level loop (to run `Game`'s `Update` repeatedly) also needs to be taken into account.

Figure 5 shows the method's CFG. Nodes 1074 and 1076 correspond to the branches at lines 8 and 9; and nodes 1078 and 1080 to the branches at lines 10 and 11.

Definition 10: A *slice S* is a set of sub-cases that belong to the same case C. It is a *complete slice* if any execution of C is an execution of a member of S (S itself does not have to be disjoint). □

For example, these are complete slices of `Update`:

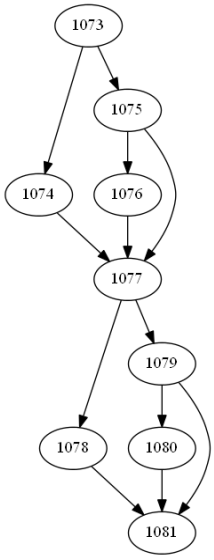

Figure 5. *The CFG of the method Bullet/Update.*

$$S_x = \{\Diamond\texttt{Update}_{1074}, \Diamond\texttt{Update}_{1075}\}$$
$$S_y = \{\Diamond\texttt{Update}_{1078}, \Diamond\texttt{Update}_{1079}\}$$

Applying our approach to the sub-cases in S_x results in five *new* oracles as shown in Figure 6 (oracles of the parent case are also oracles of its sub-cases; we do not list them below); the table induces oracles of the following form (U abbreviates `Update`):

$$p \Leftarrow \boxed{x, y \to \mathtt{U}_E(), \boxed{\Diamond\mathtt{U}_k}, \mathtt{U}_X() \to x', y'} \Rightarrow q \qquad (3)$$

$\Diamond\texttt{Update}_{1074}$ (416 samples)

past-c:	$x \leq 8.0$	√√	
	$y \leq 398.0$		(≤ 400)
	$y \geq 0.0$		(≥ -25)
post-c:	$x' = 498.0$	√√	
	$x' > x$	√√	
	$y' \leq 374.8384379119933$		(< 375)
	$y' \geq 0.0$		(> -2)
	$y' < x'$	√√	
	$x' > y$	√√	

$\Diamond\texttt{Update}_{1075}$ (44681 samples)

past-c:	$x' >= -1.7452406437283434$	(> -2)
post-c:	–	

Figure 6. Sub-case oracles for nodes 1074 and 1075.

Those marked with √√ are correct oracles (not necessarily the strongest one). The others are false positives. We notice that they are again related to inaccuracy in identifying bounds, caused by missing corner scenarios in the logs.

Notice the inferred past-condition for 1074: $x \leq 8$. Static analysis can give a stronger past-condition, namely:

$\Diamond\texttt{Update}_{1078}$ (426 samples)

past-c:	$x \leq 507.0$		(≤ 525)
	$x \geq -0.8724641969320359$		(≥ -25)
	$y \leq 7.486447365551301$		(≤ 8.0)
post-c:	$x' \leq 499.84809620246335$		(< 500)
	$x' \geq 0.0$		(> -2)
	$y' = 373.0$	√√	
	$y' > y$	√√	

$\Diamond\texttt{Update}_{1079}$ (44671 samples)

past-c:	$y' >= -2.0$	√√	(> -2)
post-c:	–		

Figure 7. Sub-case oracles for nodes 1078 and 1079.

$$x + speed.x + width \leq 0$$

where *width* is actually a constant 2. Dynamic inference of bounds of a linear formula is unfortunately problematic. With respect to the given set L of input logs, any such a formula always have lower and upper bounds. This induces an infinite number of oracles, but only a handful of them are actually correct (the others are either derivatives or can be refuted by some new execution which was not in L).

A dynamic approach will thus require a 'hint' to be supplied, in the form of some choices of which linear formulas are to be tried. For example if we encode $x + speed.x$ as a 'fake' variable x_0 in the log, it is then not a problem to infer:

$$x_0 \leq -2.0$$

Reminder: Discussion. Daikon can't actually infer the above. It seems to be a bug. We did write a script to find the maximum value of x', and verified the above oracle. □

We see that the 'upper' part of `Update` deals with x, and its lower part with y. This is reflected in the oracles learned from the slice S_x, which are mostly about x. The slice S_y will give us oracles about y, shown in Figure 7.

Consider again the slice S_x. Noticing that 1075 has two outgoing branches, we can decide to split its sub-cases into two further sub-case representing the branches. However, this time it is not possible to use a single node visit to identify the new sub-cases (because of the use of 'if-then' without 'else' at node 1075). Edge visits will be required, namely $\Diamond\texttt{Update}_{[1075,1076]}$ and $\Diamond\texttt{Update}_{[1075,1077]}$. Doing this split give us more oracles about x as shown in Figure 8.

We can also use visits to e.g. prime paths [13] as sub-cases, but their number is unfortunately exponential. Alternatively, we can either limit the length of the paths, or select just some of the paths. For example, the path [1073, 1075, 1077, 1079, 1081] is interesting because it is the only path that does not wrap the bullet around the screen. Selecting this as an additional sub-case gives new oracles shown in Figure 9.

$\diamond$Update$_{[1075,1076]}$ (301 samples)

past-c:	$x \geq 490.46505978071957$		(≥ 490)
	$x \neq 0$	$\surd\surd$	
	$y \leq 373.0$		(≤ 400)
	$y \geq -2.0$		(≥ -25)
	$y < x$	$\surd\surd$	
post-c:	$x' = 0.0$	$\surd\surd$	
	$x' < x$	$\surd\surd$	
	$y' \leq 373.0$		(≤ 375)
	$y' \geq -1.3537860237055863$		(≥ -2)
	$y' < x$	$\surd\surd$	

$\diamond$Update$_{[1075,1077]}$ (44380 samples)

past-c:	$x \leq 500.0$	$\surd\surd$	(< 500)
post-c:	$x' \neq 0$	$\surd\surd$	
	$x' \leq 499.99533810498167$	$\surd$	(< 500)

Figure 8. Sub-case oracles for the outgoing branches of 1075.

Update$_{[1073,1075,1077,1079,1081]}$ (43625 samples)

past-c:	$x \leq 500.0$	$\surd\surd$	(< 490)
	$x \geq -1.7452406437283434$		(> -2)
	$y \leq 377.0$	$\surd\surd$	(< 365)
	$y \geq -2.0$	$\surd\surd$	(> -2)
post-c:	$x' \leq 499.99533810498167$	$\surd$	(< 500)
	$x' \geq -1.999999999999993$	$\surd$	(> -2)
	$x' \neq 0$		
	$y' \leq 374.99238475781965$	$\surd$	(< 375)
	$y' \geq -1.9832464672030596$		(> -2)

Figure 9. Sub-case oracles for the outgoing branches of 1075.

Unfortunately, due to a bug in Daikon these post-c oracles cannot be inferred. Then it would become very nice because they happen to characterize the path:

$$x'-x-speed.x = 0 \quad \text{and} \quad y'-y-speed.y = 0$$

Still, if we encode e.g. $x_0 = x + speed.x$, Daikon has no problem deriving $x' = x_0$.

B. Game's Update

This method is much more complex; see again the statistics in Figure 2. Moreover, instances of Game have complex state structures (over 40 fields, some of them are arrays or objects). Recall that the logging requires some state information to be logged at the method's entrance and exits. We cannot just serialize the whole state of a Game as the overhead would be unacceptable. For this experiment we decided to select the following state information:

- *state*, an integer representing the game's global state (playing, paused, or at the main-menu).
- *gameOver* and *shipVisible* with the obvious meaning.
- *numBullets* representing the number of bullets currently alive.

As the slice (Def. 10) we take $S_\pi = \{\text{Update}_k \mid k \in \pi\}$ where π is the set of all nodes in Update's CFG that either read or write to any of those variables, but excluding the root node; π turns out to consist of 14 nodes. The slice is (in this example) complete, but unlike the slices in the previous example, the sub-cases of this slice are not disjoint.

variable	type	case-o	**new**	w-edge
state	int	1	31 (5/0/26)	+(0,0,5)
shipVisible	bool	0	17 (5/4/8)	+(0,0,2)
gameOver	bool	0	18 (5/5/8)	+(0,0,1)
numBullets	int	1	31 (5/15/11)	+(0,1,1)
total		2	97 (20/24/53)	+(0,1,9)

Figure 10. Sub-cases oracles gained using slice S_π

We constrained Daikon to derive oracles for one variable at a time. This prevents Daikon from producing accidental relations between variables. Figure 10 summarizes the result; each row represents a variable (from the chosen state-projection defined above). The third column shows the number of *case oracle* we get for the whole Update, which is the number of past- and post-condition predicates we get from Daikon. As we can see, very little can be inferred at that level.

The 4th column gives the number of *new* oracles we get from the slice S_π. In each entry k (k_1, k_2, k_3), k is the total number of new oracles, k_1 are those that get too few witnesses (< 7), k_2 are those that are false positives, $k_3 = k-k_1-k_2$ is the number of correct oracles for which we have 'enough' evidence. In total we get 53 new such oracles, which is a considerable improvement to just 2 in the previous situation.

As in the previous example, we can split a sub-case $\diamond\text{Update}_k$ to $\diamond\text{Update}_{[k,l]}$ and $\diamond\text{Update}_{[k,m]}$ if the node k branches out to l and m. These may give use more oracles. So we define a new slice $S_{\pi 2} = \{\diamond\text{Update}_{[k,l]} \mid (k,l) \in \pi 2\}$, where $\pi 2$ is the set of outgoing branches of the nodes in π.

The fifth column in Figure 10 shows how much new oracles we get from $S_{\pi 2}$. In each entry $+(l_1, l_2, l_3)$, l_1 us the number of reported oracles but with too few witnesses, l_2 is the number of false positives, and l_3 is the number of correct oracles, with enough evidence. We can see that this gives us 9 new oracles.

To check the usefulness of the oracles we made 15 mutants of Game's Update, each contains just one error. The errors are manually seeded. In particular, we want to limit to only errors that affect the variables previously selected to be used in the oracles (listed earlier in this subsection). Errors that do not affect them are undetectable by the oracles.

$N = 15$	survive	case-kill	subcase-kill	o-kill
	6 (4/2)	0	8 (7/1)	1

Figure 11. Mutants killed by the oracles

"Case-kill" is the number of errors (out of 15) that can be found by case oracles alone. "Subcase-kill" is the number of errors that can be detected by the sub-case oracles (8). We only count the detection by "valid" oracles; these are those which during the inference had sufficient number witnesses

(≥7) and are not marked as false positive. Of these 8, 7 are detected by the slice S_π, and 1 is detected by the $S_{\pi 2}$. This indicates that edge-visits do gives us additional strength. "Survive" is the number of errors that are left undetected (8). It is expected that we will not be able to find all errors. The important observation is the increase in the number of errors found due to sub-cases oracles (from 0 to 8); in our oppinion the improvement is substantial.

When selecting the slice S_π (and $S_{\pi 2}$) we only consider the nodes that directly read or write to the selected set of variables. The selection can be strengthened by adding more nodes, e.g. based on data flow. Selecting more nodes yield more oracles. For example, if we just include all nodes of `Update`, one more error can be detected (the "o-kill" column above). However, some balancing will be needed as more sub-cases will also lead to more false positives.

False positives are not always useless. Some may have overwhelming number of witnesses, and can be considered to be properties of typical (frequently occurring) executions of the corresponding subcases. A violation to such an oracle may be worth to be investigated. Out of 6 errors undetected by the oracles, 2 are detected by such false positives.

VIII. Related Work

In general, disjunctions can be inferred by Daikon by first splitting events in logs into groups. Any specification that holds for one group but not for the others induces such a disjunct. Our sub-cases are splitters. Daikon's native splitters are state predicates over events (of the same type) [4]. At their most general, sub-cases subsume the native Daikon splitters, because sub-cases are predicates over event sequences. The splitters obtained by our more restrictive implementation are complementary to the native splitters. Kuzmina et al use static analysis to calculate Daikon splitters [8]. In our solution we only use visit patterns as sub-cases, which are powerful and easy to express. Dodoo et al investigated random-based and clustering-based splitting [7]. A random approach can infer a disjunct which is satisfied by most (but not all) executions, so that the chance to get a group consisting only of such executions is not too small. Clustering works if a sub-case can be re-expressed in terms of 'distances'. That is, executions that belong to it can be identified by being 'close' to each other. Whereas in our approach we require deep logging, clustering can relax this requirement.

We now discuss various approaches to dynamically infer different types of specifications/oracles.

To infer global invariants, past-, and post-conditions Daikon is probably still the main tool to use [4]. It is a specialized tool, with a limited vocabulary for which it performs well. A more general purpose approach is by Feather [14]: logs are converted to tables in a relational database, and database queries are used to express specifications and check whether they are valid. Ducasse et al [15] convert logs to Prolog facts, and use Prolog queries for the same purpose. These approaches allow complex specifications to be expressed and queried, but we (or another tool) have to figure out ourselves, which specifications are sensible to query.

Daikon is not suitable to infer algebraic specifications, because the terms appearing in such specifications are typically specific for the target class. Henkel and Diwan [5] use reflection to first discover candidate operators of the target algebra. Then, candidate equations are constructed. The target class is then driven to execute the terms in the equations and to check whether the equations are valid. Directedly driving the class is crucial. Elyasov [16] studied the inference of common equations such as commutativity and absorbtion from undirected/random executions. Interestingly, he exploits the equations to rewrite logs to get shorter but still equivalent variants.

Finite State Automata (FSA) can also be used as oracles [17]. Marchetto et al [18] use logs to infer an FSA that models the behavior of the target program. Dallmeier et al [19] use a function to map concrete states of objects of a target class C to a finite domain F. By generating executions on methods of C, we can infer how each method affects the states in terms of F. Effectively, we build an FSA describing how these methods operate over F. Raffelt et al [20] use an active learning algorithm: it actively drives and queries the program to construct a matching FSA.

As an oracle, an FSA is more powerful if e.g. the states are decorated with predicates (e.g. $x>0$). Marchetto et al [18] exploit abstraction functions that abstract the target program's real states to such predicates. However, these functions must be manually designed. Lorenzoli et al [21] use Daikon to infer such state predicates. The Daikon approach can be said to be 'brute force', as it simply quantifies over a large amount of candidate formulas. On the other hand, FSA inference is usually incremental: the FSA is gradually grown as the logs are scanned. A brute force approach is not suitable to infer temporal properties as in, e.g. LTL. The number of candidate properties quickly explodes, and we cannot know upfront which ones to try first since this is very problem specific. Simple and common temporal patterns can still be feasibly inferred. Weimer and Necula [22] studied the inference of regular expressions (over event types) of the form $(e_1 e_2)*$. Gabel and Su [6] give a more general algorithm, that can infer instances of a given regular expression template R. This template can be any regular expression involving at most two different event types. But note that the choice of R is fixed upfront by the user.

Log-based analyses have their advantages, but they are inherently unsound (we may get false positives). Static inference, usually in the form of type inference, does not suffer from this problem. On the other hand, static analysis does not scale very well to large applications. Dynamic

inference can be improved, as shown by Dallmeier et al [19], by statically verifying dynamically inferred oracles. Such a combination is still unsound (due to the limitations of the static approaches when dealing with loops and recursion), but it is stronger (than without static verification).

IX. Conclusion and Future Work

Sub-cases is a quite general concept of splitting. Regular sub-cases give a convenient and semantically meaningful way to express splitting. In our case study, they greatly increased the number of oracles we obtained. They do require deeper logging. Being log-based, our approach is much simpler to implement than a static approach. On the other hand, our approach is unsound (we do get false positives).

Future work. CFG-based sub-cases are sensitive to modifications to the CFG. Fortunately modifications are only local, which calls for an algorithm that can identify which oracles are (likely) to remain valid. The problem seems to be related to CFG-based tests selection in regression, where several algorithms are known [23], [24].

Some execution paths are very difficult to produce, which implies that logs are likely to be incomplete, and thereby our inference unsound. Using an evolutionary-based execution driver may improve the accuracy of the inference. Another approach is to combine it with static verification (see Section VIII). Since Daikon's native splitters and our (restricted) implementation of the sub-case splitters are complementary, it makes sense to combine them in a future study.

X. Acknowledgement

The research is part of the project 257574 named FITTEST, funded EU project. We thank Paolo Tonella, Nguyen Cu, and Alessandro Marchetto from Fondazione Bruno Kessler for the many discussions we had.

References

[1] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of haskell programs," *SIGPLAN*, vol. 35, no. 9, pp. 268–279, 2000.

[2] I. S. W. B. Prasetya, T. E. J. Vos, and A. I. Baars, "Trace-based reflexive testing of OO programs with T2," in *Int. Conf. on Softw. Testing, Verification, and Validation (ICST)*. IEEE, 2008.

[3] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on Java predicates," in *ISSTA '02: Proc. of the 2002 ACM SIGSOFT Int. Symp. on Soft. Testing and Analysis*. ACM Press, 2002.

[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007.

[5] J. Henkel and A. Diwan, "Discovering algebraic specifications from java classes," in *In ECOOP*. Springer, 2003.

[6] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *32nd Int. Conf. on Softw. Engineering (ICSE)*. ACM, 2010, pp. 15–24.

[7] N. Dodoo, L. Lin, and M. Ernst, "Selecting, Refining, and Evaluating Predicates for Program Analysis," MIT Lab for Computer Science, Tech. Rep. MIT-LCS-TR-914, 2003.

[8] N. Kuzmina, J. Paul, R. Gamboa, and J. Caldwell, "Extending dynamic constraint detection with disjunctive constraints," in *Int. Workshop on Dynamic Analysis (WODA)*. New York, NY, USA: ACM, 2008.

[9] I. S. W. B. Prasetya, A. Middelkoop, A. Elyasov, and J. Hage, "D6.1: FITTEST Logging Approach, Project no. 257574, FITTEST Future Internet Testing," 2011.

[10] "The daikon invariant detector developer manual," 2010, version 4.6.4. [Online]. Available: http://groups.csail.mit.edu/pag/daikon/

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[12] M. Ernst, "Dynamically discovering likely program invariants," Ph.D. dissertation, University of Washington Department of Computer Science and Engineering, Aug. 2000.

[13] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[14] M. S. Feather, "Rapid application of lightweight formal methods for consistency analyses," *Software Engineering, IEEE Transactions on*, vol. 24, no. 11, pp. 949–959, 1998.

[15] S. Ducasse, T. Girba, and R. Wuyts, "Object-oriented legacy system trace-based logic testing," in *10th European Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE, 2006.

[16] A. Elyasov, I. Prasetya, and J. Hage, "Log-based reduction by rewriting," Dept. of Inf. & Comp. Sciences, Utrecht Univ., Tech. Rep., 2013.

[17] J. H. Andrews, "Testing using log file analysis: tools, methods, and issues," in *13th IEEE Int. Conf. on Automated Softw. Engineering*, 1998, pp. 157–166.

[18] A. Marchetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Applications," in *ICST*. IEEE, 2008, pp. 121–130.

[19] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *Int. Workshop on Dynamic Systems Analysis (WODA)*. ACM, 2006, pp. 17–24.

[20] H. Raffelt, M. Merten, B. Steffen, and T. Margaria, "Dynamic testing via automata learning," *Int. Journal on Softw. Tools for Technology Transfer (STTT)*, vol. 11, pp. 307–324, 2009.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *30th Int. Conf. on Softw. engineering*. ACM, 2008, pp. 501–510.

[22] W. Weimer and G. Necula, "Mining temporal specifications for error detection," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2005, vol. 3440, pp. 461–476.

[23] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[24] T. Ball, "On the limit of control flow analysis for regression test selection," in *ISSTA*, 1998, pp. 134–142.