Specification Inference

Wei Le

March 22, 2021

What is specification?

- specification – requirement of a program or a function, typically they are "formal" – mathematical, computable ..
  - annotations
  - "types" in the code
  - mathematical description of software using formal languages like JML, Z and alloy

Motivation: why do we study and research it?

- Since 2011, engineers at Amazon Web Services (AWS) have been using formal specification and model checking to help solve difficult design problems in critical systems [1]
- Microsoft uses annotations to verify buffer overflows [2]
- A strongly typed language would have reduced bugs by 15% [3]
- Assertions are great for testing, debugging ... [4]

Types of specification

- *pre-condition*, *post-condition*: program conditions that must be hold before/after executing a program or a procedure
- *program invariant*: conditions that hold for all program paths at a program point
- *assertion*: conditions that programmers expect/require a program to hold along all execution paths
- *typestate*: the API/system call only can be performed on a proper state of a program (typically refer to some resource problems). We also call this source sink problem

Specification for different systems

- Infer deterministic specifications of multi-threaded programs
- Infer specification for distributed systems
- Infer specification for embedded systems
- Infer specification for nerual network: what are the invariants during training?

Topics

Specification languages, check and verify programs using specifications (both static and dynamic analysis), automatically infer specifications (off-line trace analysis),

- Specifying changes (2015): change contract and differential assertions
- Diakon (2000): dynamic analysis to detect likely invariant
- Infer finite machines (2002): offline dynamic analysis

Software change contract

- *change contract*: express the intended program behavior changes across program versions
- based on a specification language called Java modeling language (JML)

Program behaviors: pre-/post-conditions, but how to specify changes of pre-/post-conditions?

condition_old: using variables in version 1: v == 0
condition_new: using variables in version 2: v == 0
Problem: the two are not comparable if the values of variables are changed.

What about something like this?

$$whenever\ in > 0\ holds,\ out' == out + 1$$
$$whenever\ out > 0\ holds,\ out' == out + 1$$

Software change contract

The contributions of the work:

- design a novel approach to specify changes
- evaluate its expressiveness, usability
- develop static and dynamic checkers to check if software changes conform to the specification

Software Change Contract: Examples

Bug 51668 - <junitreport> broken on JDK 7 when a SecurityManager is set Fails with: "Use of the extension element 'redirect' is not allowed when the secure processing feature is set to true." It turns out to apply to any environment in which there is a system security manager set. JDK 7's TransformerFactoryImpl constructor introduced:

```
if (System.getSecurityManager() != null) {
    _isSecureMode = true; _isNotSecureProcessing = false;
}
```

which conflicts with <redirect:write>.

(a) a sample Bugzilla report for software Ant

```
// file: XMLResultAggregator.scc
package org.apache.tools.ant.taskdefs.optional.junit;

public class XMLResultAggregator extends Task implements XMLConstants {
  /*@ changed_behavior
    @ requires System.getSecurityManager() != null &&
    @   System.getProperty("java.runtime.version").startsWith("1.7") &&
    @   getDestinationFile().exists() == false;
    @ when_signaled (BuildException e) e.getMessage().contains(
    @   "Use of the extension element 'redirect' is not allowed " +
    @   "when the secure processing feature is set to true.");
    @ signals (BuildException e) false;
    @ ensures getDestinationFile().exists();
    @*/
  public void execute() throws BuildException;
}
```

(b) a change contract corresponding to the bug report in (a)

Software Change Contract: Examples

- requires φ: the input constraint for the old and new version
- when_signaled ψ, signaled ψ': exception output for old and new versions
- when_ensure θ, ensure θ': normal output condition for the old and new versions

Software Change Contract: Examples

```
// file: SourceTypeBinding.scc
package org.eclipse.jdt.internal.compiler.lookup;

class SourceTypeBinding extends ReferenceBinding {
    /*@ changed_behavior
      @  requires method.parameters.length > 0;
      @  when_ensured method.parameterNonNullness[0].booleanValue() ==>
      @            isNonNull(method.sourceMethod().arguments[0]) == false;
      @  ensures method.parameterNonNullness[0].booleanValue() ==>
      @            isNonNull(method.sourceMethod().arguments[0]) == true;
      @*/
    public MethodBinding resolveTypesFor(MethodBinding method);

    /*@ pure model boolean isNonNull(Argument arg) {
    return (arg.binding.tagBits & TagBits.AnnotationNonNull) != 0; } @*/
}
```

(c) a core-developer-level change contract

Software Change Contract: Examples

```
public class DirectoryScanner implements FileScanner {

  private /*@ new_field @*/ int mode;

  // If !cs at the entry of the method, the behavior of the method changes.
  // If cs at the entry of the method, the behavior of the method is preserved.
  /*@ changed_behavior
    @ when_required true;
    @ requires !cs;
    @ ensures /* omitted: description about behavioral changes */;
    @ preserves_when cs;
    @*/
  File findFile(File base, String path, /*@ old_param @*/ int mode, /*@ new_param @*/ boolean cs);
}
```

Software Change Contract Language

```
// the full change contract, (φ, ψ, θ; φ', ψ', θ')
/*@ changed_behavior
  @  when_required φ;   when_ensured ψ;   when_signaled (T1 x) θ;
  @  requires φ';        ensures ψ';       signals (T2 x) θ';
  @*/
```

(b) a boilerplate for the full change contract (the greek letters denote predicates, and T_1 and T_2 represent exception types)

Software Change Contract Language

method-spec ::= *spec-case-seq*

spec-case-seq ::= *spec-case* [also *spec-case*]∗

spec-case ::= **changed_behavior** *clause-seq*

clause-seq ::= [*clause*]∗

clause ::= requires *pred*; | ensures *pred*; | signals (*reference-type* [*ident*]) *pred*;
| **when_ensured** *pred*; | **when_signaled** (*reference-type* [*ident*]) *pred*;
| **when_required** *pred*; | **preserves_when** *pred*;

exp ::= . . . | \result | \old(*exp*) | **prev**(*exp*)

param-modifier ::= . . . | **new_param** | **old_param**

jml-modifier ::= . . . | **new_field** | **old_field**

(a) the grammar of our change contract language, which is an extension of a JML subset (standard regular expression notation ∗ is used)

Software Change Contract: Evaluating specification techniques

Goal: is the language expressive?
Approach: recruited 16 final year undergraduate students to finish the following tasks:

- write a change contract given a description (W)
- explain a change contract in English (RD)
- accomplish the code based on change contract (RM)

Software Change Contract: results

Table II. Distribution of Correct Answer Rates Depending on the Criterion Used to Categorize Questions

Three Categorization Criteria						
Question Type			Program Source		Change Kind	
RM	RD	W	Artificial	AspectJ	B	S
100%	86%	93%	92%	92%	85%	97%

correct answer rate 92%, ave 53 min for a total of 20 questions
conclusions: easily learned and used in dependent of real life programs or constructed programs, structure changes are easier than behavior changes

Checking software change contract

Definition 3 (*CCC*). Given a full-blown change contract $(\varphi, \psi, \theta; \varphi', \psi', \theta')$ of a method m, we say that CCC succeeds in m iff the following two properties hold. For all $(S_{in}, S_{out}) \in B[m\_v1]$ and $(S'_{in}, S'_{out}) \in B[m\_v2]$,

$$(P1)\ S_{in} \approx S'_{in} \wedge (S_{in} \models \varphi \wedge S_{out} \models ((\neg ex \Rightarrow \psi) \vee (ex \Rightarrow \theta)))$$
$$\Rightarrow (S'_{in} \models \varphi' \Rightarrow S'_{out} \models ((\neg ex \Rightarrow \psi') \wedge (ex \Rightarrow \theta')));$$

$$(P2)\ S_{in} \approx S'_{in} \wedge \neg(S_{in} \models \varphi \wedge S_{out} \models ((\neg ex \Rightarrow \psi) \vee (ex \Rightarrow \theta)))$$
$$\Rightarrow S_{out} \approx S'_{out}$$

- P1: the behavior of a method changes
- P2: the behavior of a method remains the same

Update condition: which pattern of the behavior of m_{v1} triggers behavioral changes in m_{v2}

Dynamic Checking

- generate tests to trigger the changed behavior: the update condition holds based the test results of the first version
- repair tests for the new version based on structure changes
- run tests for the new version

Evaluating CCC: Experimental Setup

- software subject: 10 versions of changes for Java program Ant
- convert to change contract from three sources:
  - transform bug reports to change contract
  - incorrect program changes found from previous studies
  - two structural changes

Evaluating CCC: Results

Change		Randoop	Test generation		Test repair		Contract checking	
Old	New	T_{first} (s)	T_{first} (s)	# of tests/m	# of errors	# of fixes	# of passes	# of violations
0632cd	b6c725	290	5	17	0	0	17	0
c39b90	2f95b7	0.4	0.4	1	0	0	0	0
32e66f	f0e466	62	9	4	0	0	4	0
a84f2e	1de96b	32	0.9	58	0	0	6	0
cbda11	9a0689	>300	0.2	252	0	0	0	250
dfa59d	de3f32	>300	1	79	0	0	0	79
5bee9d	1532f4	1	0.3	762	1239	1239	172	506
1de7b3	626f28c	5	1	183	263	263	0	183
3a1518	aef2f7	0.3	0.2	1209	1832	1832	1209	0
f87075	d17d1f	0.2	0.2	955	2	2	955	0

Static Checking

- Scope on a clean language and then extend to Java specifics
- key idea: composed program
- An example:

```
// the previous version (v1)
int p(int x) {
  body_1
}
```

```
/*@ changed_behavior
  @  requires φ;
  @  when_ensured ψ;
  @  ensures ψ';
  @*/
int p(int x);
```

```
// the updated version (v2)
int p(int x) {
  body_2
}
```

(a) the two versions of procedure p and their change contract in the middle

Static Checking: Composed Program

```
/***** Part I: assume (1) isomorphic input and (2) the requires clause *****/
assume x_v1 == x_v2; // parameters should be isomorphic
boolean requires_clause = ⟦φ⟧; // store the value of the requires clause

/***** Part II: interpret v1 to see if the update condition is true *****/
boolean update_condition = false; // the update condition is initially false.
int result_v1; // the variable to hold the return value of m at v1
result_v1 = ⟦body1⟧; // interpret body1 and store the return value at result_v1

// set the update condition true if the when_ensured clause is true.
boolean when_ensured_clause = ⟦ψ⟧;
if(requires_clause && when_ensured_clause) {
  update_condition = true;
}

/***** Part III: interpret v2 to see if there is any change contract violation *****/
int result_v2; // the variable to hold the return value of m at v2
result_v2 = ⟦body2⟧; // interpret body2 and store the return value at result_v2

if(update_condition) {
  // we expect the ensures clause to be true
  boolean ensures_clause = ⟦ψ'⟧;
  assert ensures_clause;
} else {
  // we expect no change
  assert result_v1==result_v2;
}
```

Static Checking: Composed Program

If our composed program (CP) is correct (i.e., no assertion error is possible), then CCC succeeds

When one of the assertions in CP is violated, a change contract violation occurs.

Static Checking: Experiment Setup

- Joda-time: 18 change instances, iBUGS dataset: pre-fix and post-fix revisions available
- Z3 and openJML (verifying programs written in JML)
- 4 types of changes and applications:
  - V: it verifies the program changes as intended
  - L: localize buggy methods
  - R: debugging, regression errors
  - C: classify causes for a test failure (is it the test code incorrect or programs contain bugs?)

Static Checking: Results

		Revision		Diff		Contract Size (lines)		Kind		Time (s)		
Usage	Bug #	Previous	Updated	–	+	CC (lines/mthds)	JML	B	S	Total	Z3	Verified
V	1788282	pre-fix (iBUGS)	post-fix (iBUGS)	98	82	3/1	2	✔	✘	7.7	1.4 (18%)	✔
	1877843			62	81	3/1	23	✔	✘	8.1	1.9 (23%)	✔
	2111763			9	14	2/1	3	✔	✘	6.7	7.5 (4%)	✔
	2487417			25	28	2/1	5	✔	✘	6.2	4.7 (7%)	✔
	2783325			2	14	(1 + 1)/1	0	✔	✔	6.2	2.6 (4%)	✔
	2903029			78	45	2/2	4	✔	✘	6.5 6.5	1.0 (16%) 0.6 (10%)	✔ ✘
L	2025928	pre-fix (iBUGS)	post-fix (iBUGS)	8	6	22/7	6	✔	✘	7.6 8.5 7.0 8.5 9.5 8.0	1.0 (14%) 1.5 (18%) 1.4 (21%) 1.7 (20%) 3.2 (35%) 0.9 (11%)	✔ ✔ ✔ ✔ ✔ ✔
R	1887104	7755b 7755b	c41ef a478f	95 1417	222 3524	2/1	10	✔	✘	8.4 6.7	1.0 (12%) 0.9 (15%)	✘ ✔
C	–	7b179	7b179′ 7b179″ 1c524	2038	962	(8 + 3)/3	4	✔	✔	7.9 7.1 6.7	2.3 (30%) 1.9 (28%) 1.8 (27%)	✘ ✘ ✔

Pre-fix/post-fix indicates the previous/updated revision provided through the iBUGS dataset; in the first column, V stands for Verification, L Localization, R Regression, and C Classification; each usage is detailed in each section.

Differential Assertion 2013 (Optional)

- Goal: to perform incremental verification and quickly verify evolving programs
- "relative specification": are there inputs for which P2 accesses buffer regions that are not accessed by P1?
- Given P and P' that contain a set of assertions, does there exist an environment in which P passes but P' fails?
- An example relative specification:

  $$\text{axiom}(\forall x : \text{int}, y : \text{int} :: x \le y \Rightarrow \mathit{Valid}(y) \Rightarrow \mathit{Valid}(x))$$

- Generate a composed program and we can verify the relative specification as if we verify a single programs:

```
assume i1 == i2 && g1 == g2;
 call p1(i1); call p2(i2);
assert (ok.1 ==> ok.2);
```

Differential Assertion: an example

```
void StringCopy.1(
      wchar_t *dst,
      wchar_t *src,
      int size)
{
      wchar_t *dtmp = dst,
              *stmp = src;
      int i;
      for (i = 0;
           *stmp &&
            i < size - 1;
           i++)
         *dtmp++ = *stmp++;
      *dtmp = 0;
}
```

```
void StringCopy.2(
      wchar_t *dst,
      wchar_t *src,
      int size)
{
      wchar_t *dtmp = dst,
              *stmp = src;
      int i;
      for (i = 0;
           i < size - 1 &&
           *stmp;
           i++)
         *dtmp++ = *stmp++;
      *dtmp = 0;
}
```

```
pre   stmp.1 == stmp.2 &&
      dtmp.1 == dtmp.2 &&
      Mem_char.1 == Mem_char.2 &&
      i.1 == i.2 &&
      size.1 == size.2 &&
      ok.1 <==> ok.2

post  ok.1 ==> ok.2 &&
      dtmp.1 == dtmp.2
proc MS_loop.1_loop.2(dst.1, ..., dst.2, ...);
```

Differential Assertion: an example

- inputs of two versions are the same: stmp.1 == stmp.2, dtmp.1 == dtmp.2, size.1==size.2
- heaps of two versions are the same: Mem_char.1 == Mem_char.2, i.1==i.2
- two versions have the same correctness state: ok.1 <==> ok.2
- MS_loop.1_loop.2(dst.1, ..., dst.2, ...): composed loops

Evaluation: Experimental Setup

- Subject: Verisec suite
- Infrastructure: SYMDIFF, Z3
- Applications:
  - verify bug fixes
  - filtering alarms for evolving programs compared to checking assertions on a single program

Evaluation: Results on Windows Driver Kit

Name	Diff	SymDiff	single	sound	unsound	shallow	nonmodular	LOC	#procs
firefly	1	1	1	1	1	1	1	634	7
moufilter	4	2	0	0	0	0	0	504	6
pciide	4	0	1	0	0	0	0	182	5
sfloppy	14	6	11	1	1	1	2	3404	20
diskperf	4	4	4	3	2	2	2	2319	24
event	1	1	0	0	0	0	1	555	5
cancel	3	1	0	1	0	0	0	476	5
Total	31	15	16	6	4	4	6	8074	72

- diff: number of procedures syntactically modified
- symdiff: the tool SymDiff fails
- single: the number of warnings generated by verifying single versions
- sound/unsound/shallow/nonmodular: the number of warnings generated by verifying using differential assertions (different configurations for handling procedural calls: sound – using summary of callees, unsound – ignore callees, shallow – assume callees are the same, nonmodular – inline callees)

Diakon

See 1999 ICSE slides from the first paper of Diakon

Mining Specification 2002

- motivation: verifying program specific properties needs program specific specification
- output: the temporal and data dependencies when a program interacts with API (application programming interface) and ADT (abstract datatype)
- input: traces of a program's run-time interaction with an API or ADT

Mining Specification: code

```
int s = socket(AF_INET, SOCK_STREAM, 0);
...
bind(s, &serv_addr, sizeof(serv_addr));
...
listen(s, 5);
...
while(1) {
  int ns = accept(s, &addr, &len);
  if (ns < 0) break;
  do {
     read(ns, buffer, 255);
     ...
     write(ns, buffer, size);
     if (cond1) return;
  } while (cond2)
  close(ns);
}
close(s);
```

Figure 1: An example program using the *socket* API.

Mining Specification: trace

```
 1 socket(domain = 2, type = 1, proto = 0,
          return = 7)
 2 bind(so = 7, addr = 0x400120, addr_len = 6,
        return = 0)
 3 listen(so = 7, backlog = 5, return = 0)
 4 accept(so = 7, addr = 0x400200,
          addr_len = 0x400240, return = 8)
 5 read(fd = 8, buf = 0x400320, len = 255,
        return = 12)
 6 write(fd = 8, buf = 0x400320, len = 12,
         return = 12)
 7 read(fd = 8, buf = 0x400320, len = 255,
        return = 7)
 8 write(fd = 8, buf = 0x400320, len = 7,
         return = 7)
 9 close(fd = 8, return = 0)
10 accept(so = 7, addr = 0x400200,
          addr_len = 0x400240, return = 10)
11 read(fd = 10, buf = 0x400320, len = 255,
        return = 13)
12 write(fd = 10, buf = 0x400320, len = 13,
         return = 13)
13 close(fd = 10, return = 0)
14 close(fd = 7, return = 0)
```

Figure 2: Part of the input to our mining process: a trace of an execution of the program in Figure 1.

Mining Specification: automata

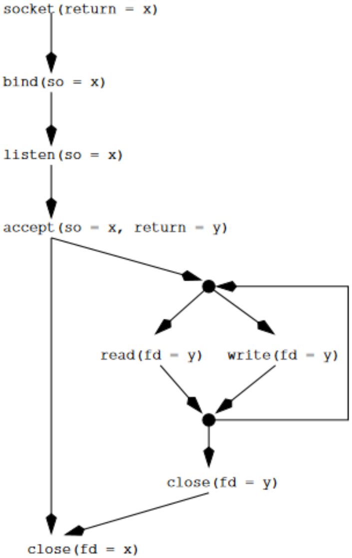


Figure 3: The output of our mining process: a specification automaton for the socket protocol.

Mining Specification: workflow

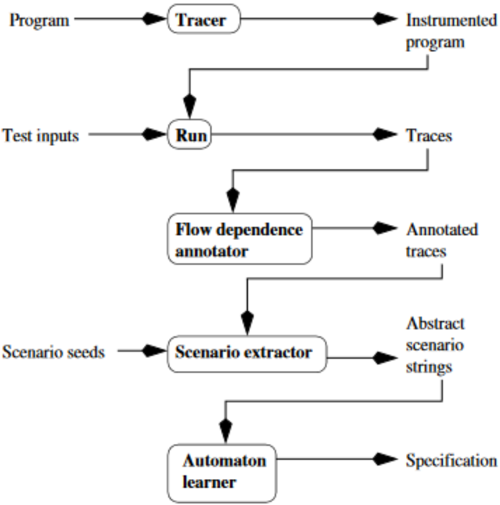


Figure 4: Overview of our specification mining system.

Mining Specification: workflow

Step 1 – tracing: 1) instrument C stdio library 2) generate instrumented x11 API, replace current executable with instrumented versions (graphical output in UNIX has to go through the standard UNIX windowing system: the X Window System, release 11)

```
int instrumented_socket(int domain,
                        int type,
                        int proto)
{
  int rc = socket(domain, type, proto);
  fprintf(the_trace_fp,
          "socket(domain = %d, type = %d, "
          "proto = %d, return = %d)\n",
          domain, type, proto, rc);
  return rc;
}
```

Figure 5: Illustration of trace instrumentation (instrumented version of `socket`).

Mining Specification: workflow

Step 2 – flow dependence annotator

- dependency analysis (manually define which call is define, which call is use): define – change the state of an object, use – depend on the object of a state; aim to extract a small sets of dependent interactions – scenarios
- type inference: assigns a type for each interaction attribute

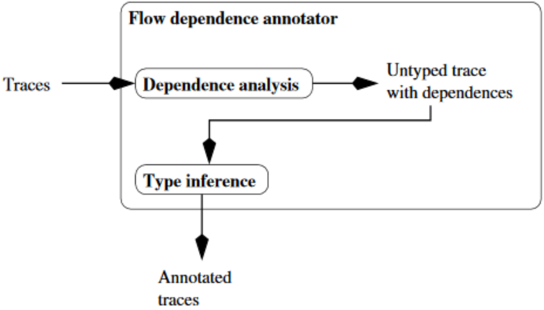


Figure 6: Detailed view of the flow dependence annotator.

Definers: `socket.return`
`bind.so`
`listen.so`
`accept.return`
`close.fd`

Users: `bind.so`
`listen.so`
`accept.so`
`read.fd`
`write.fd`
`close.fd`

Type(`socket.return`) = `T0`
Type(`bind.so`) = `T0`
Type(`listen.so`) = `T0`
Type(`accept.so`) = `T0`
Type(`accept.return`) = `T0`
Type(`read.fd`) = `T0`
Type(`write.fd`) = `T0`
Type(`close.fd`) = `T0`

```
int s = socket(AF_INET, SOCK_STREAM, 0);
...
bind(s, &serv_addr, sizeof(serv_addr));
...
listen(s, 5);
...
while(1) {
  int ns = accept(s, &addr, &len);
  if (ns < 0) break;
  do {
     read(ns, buffer, 255);
     ...
     write(ns, buffer, size);
     if (cond1) return;
  } while (cond2)
  close(ns);
}
close(s);
```

Mining Specification: workflow

Step 3 – scenario extraction: a scenario is a set of interactions related by flow dependencies; given a N that represents how many interactions in the trace

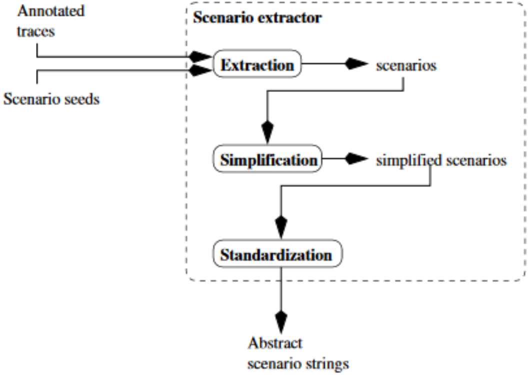


Figure 9: Detailed view of the scenario extractor.

Mining Specification: workflow

seed: accept(so, return)

```
1 socket(domain = 2, type = 1, proto = 0,
         return = 7)
2 bind(so = 7, addr = 0x400120, addr_len = 6,
       return = 0)
3 listen(so = 7, backlog = 5, return = 0)
4 accept(so = 7, addr = 0x400200,
         addr_len = 0x400240,
         return = 8) [seed]
5 read(fd = 8, buf = 0x400320, len = 255,
       return = 12)
6 write(fd = 8, buf = 0x400320, len = 12,
        return = 12)
7 read(fd = 8, buf = 0x400320, len = 255,
       return = 7)
8 write(fd = 8, buf = 0x400320, len = 7,
        return = 7)
9 close(fd = 8, return = 0)
```

Figure 10: A scenario extracted from around line 4 of Figure 2, with $N = 10$

```
1 socket(return = 7)
2 bind(so = 7)
3 listen(so = 7)
4 accept(so = 7, return = 8) [seed]
5 read(fd = 8)
6 write(fd = 8)
7 read(fd = 8)
8 write(fd = 8)
9 close(fd = 8)
```

Figure 11: The simplification of the scenario in Figure 10.

```
1   socket(return = x0:T0)                          (A)
2   bind(so = x0:T0)                                (B)
3   listen(so = x0:T0)                              (C)
4   accept(so = x0:T0, return = x1:T0) [seed]       (D)
5   read(fd = x1:T0)                                (E)
7   read(fd = x1:T0)                                (E)
6   write(fd = x1:T0)                               (F)
8   write(fd = x1:T0)                               (F)
9   close(fd = x1:T0)                               (G)
```

Figure 12: Scenario string for the simplified scenario from Figure 11.

Mining Specification: workflow

Step 4 – automaton learning

- Learn a PFSA from the string (k-tail algorithm)
- Convert from PFSA to NFA with edges labeled by standardized interactions by dropping off infrequent edges (caused due to heuristics in the algorithm)

Evaluation - experimental setup

- subject: X11 programs that uses the Xlib and X Toolkit libraries
- implementation: Executable Editing Library (EEL) for binary instrumentation
- challenge of coping with very few correct traces at the beginning (see paper for the process)

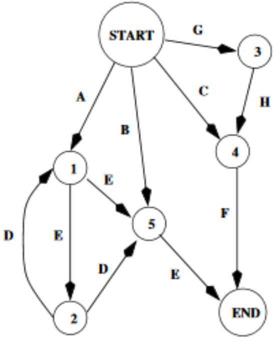
Evaluation - results

Name	Verifies?	Reason for failure	Action
xcb	n/a	n/a	accept
bitmap	no	spec. too narrow	accept
ups	no	bug!	reject
ted	no	spec. too narrow	accept
rxvt	yes	n/a	accept
xterm	no	spec. too narrow	accept
display	no	spec. too narrow	accept
xcutsel	no	spec. too narrow	accept
kterm	yes	n/a	accept
pixmap	yes	n/a	accept
cxterm	yes	n/a	accept
xconsole	no	benign violation	reject
nedit	no	spec. too narrow	accept
e93	no	bug!	reject
xclipboard	no	benign violation	reject
clipboard	no	benign violation	reject

Table 2: Results of processing each client program, in the order in which they were processed.

Verify: it takes a trace, a specification and a max scenario size; it verifies that the trace satisfies the spec

Evaluation - results



A – XNextEvent(time – X21_0)
B – XNextEvent(time – X21_0) **or B** – XtDispatchEvent(time – X21_0)
or B = XIfEvent(time = X21_0)
C = XtDispatchEvent(time = X21_0) **or C** = XtEventHandler(time = X21_0)
or C – XtLastTimeStampProcessed(time – X21_0)
D – XGetSelectionOwner
E – XSetSelectionOwner(time – X21_0)
F = XtOwnSelection(time = X21_0)
G = XtActionHookProc(time = X21_0)
H – XInternAtom

Figure 22: The NFA from the selection ownership specification.

Further Reading

1. Use of Formal Methods at Amazon Web Services
2. Modular Checking for Buff er Overflows in the Large
3. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript
4. Use of Assertions
5. Dynamically Discovering Likely Program Invariant to Support Program Evolution, 2001
6. Dynamically Discovering Likely Program Invariant, PhD thesis by Michael Ernst
7. Software Change Contract
8. Do I Use the Wrong Definition?
9. Differential Assertions
10. Mining specifications