AutoInSpec: Using Missing Test Coverage to Improve Specifications in GUIs

Myra B. Cohen, Si Huang
University of Nebraska-Lincoln
Department of Computer Science & Engineering
Lincoln, NE 68588-0115
Email: {myra,shuang}@cse.unl.edu

Atif M. Memon
University of Maryland
Department of Computer Science
College Park, MD 20742
Email: atif@cs.umd.edu

***Abstract*—Developers of a software's graphical user interface (GUI) often fail to document the interface specifications. Without these, models used for automated test generation and execution remain imperfect and incomplete. This leads to unexpected behavior that creates unrecoverable situations for test harnesses, and missed coverage. In this paper, we present AutoInSpec, a technique to infer an important class of specifications, temporal and state-based invariants between GUI events that have been incorrectly modeled. Unlike existing specification mining approaches that require full execution traces, or source code, and that mine all invariants, we simplify the problem. We guide AutoInSpec with coverage criteria and use a previously developed repair framework that builds coverage-adequate test suites, removing unexecutable sub-sequences from consideration. These failing sub-sequences are input to a logic-based inference engine, armed with known invariant templates, to obtain the missing specifications. We validate AutoInSpec on a set of well studied GUI applications.**

***Keywords*-GUI Testing; Specification Mining; Invariants; Genetic Algorithm; Covering Arrays**

I. Introduction

Absent, incomplete, and incorrect specifications of graphical user interface (GUI) front-ends cause non-trivial problems for model-based test automation. An unexpected window, a disabled menu item—while not necessarily a functional bug—will cause an automated GUI-based test harness to hang or fail, forcing manual intervention and missed test coverage. Models are needed for high quality system testing in GUIs, since they guide test generation and ensure sufficient coverage. Yet techniques that extract these models use heuristics to approximate the interface behavior – a tradeoff that enables scalability [1]–[4].

Although testers may be able to refine extracted models, the information needed for these refinements is not easily obtainable and often missed. Consider this specification: "*In MS Word for Mac 2011, the 'Copy to Scrapbook' menu-item remains disabled until the Scrapbook window is open and, simultaneously, some copyable object has been selected*." It is unlikely that we can find documentation that details the above behavior. Nor will a developer be able to precisely describe all such GUI behaviors. Consequently, many model-based testing approaches will over-approximate the system behavior assuming the menu-item is available when it is not. End users may take for granted the absence of such specifications – the interface disables the menu when unavailable, and the user either notices this state and infers the infeasibility of clicking on the disabled item, or ignores it. However, automated test harnesses lack our human cognitive ability; without precise specifications of such behaviors, they may not be able to recover from the unexpected unavailability of GUI actions, and an unanticipated dialog box can cause a harness to wait indefinitely if it is not prepared to interact.

We might turn to automated specification inference; there has been extensive research on this topic, mining logs and execution traces [5]–[7], and running test cases to detect white-box invariants [8]–[10]. But that work is too general on its own. Specifications output by trace-based inference techniques are only as good as the traces they consider and GUIs have an enormous combinatorial space of events. The number of traces/behaviors is so large that these techniques will either not scale, or if provided with a small subset of behaviors, may output incomplete specifications. Suppose a user can invoke any of the following events on a drawing canvas in any order: *Copy, Paste, Resize, Rotate90, Color, Erase*. The sequence <*Rotate90, Color, Copy, Paste*> may encounter different invariants than the sequence <*Rotate90, Color, Paste, Copy*>, but we have 36 unique length-two sequences, over 7,500 unique length-five sequences, and more than 60 million length ten sequences.

Second, we are interested in system testing, which means that we may not have full source code (e.g., for GUI library functions) and cannot use techniques that rely on instrumentation. Our invariant detection is limited to those methods which are blackbox.

We can leverage the following insights to refine specification inference for improving GUI test models. First, we have models for GUIs that can be automatically extracted, and are only interested in inferring those specifications that violate the model; we don't need to apply inference algorithms to the full combinatorial space. Second, if we can use a systematic test generation technique that covers a broad (and measured) set of interface behaviors, we can ensure we identify only the combinations of events that are of interest, removing many invariants from consideration.

In this paper, we build upon several developments from our prior work on GUI testing to create a novel technique for GUI model specification inference and invariant detection that we call *AutoInSpec*. AutoInSpec is driven by the use of combinatorial testing, which forces the execution of all combinations of events (of a specified arity) in all positions, meaning we are more likely to find violations of our model [11], [12]. We use the by-product of *test suite repair* [11], which has been used to increase combinatorial coverage of generated test suites, therefore we do not have to perform extra work to obtain our invariant input set. The normal output of test suite repair is a test suite that is coverage-adequate. AutoInSpec uses only failed sub-sequences (those missed by the repair), which is a fraction of the size of the original test suite. It then extracts new specifications for the model using an off-the-shelf logic programming environment. Although our prior work suggested a set of invariants that we might expect to find, these were not formalized or evaluated against real applications; a step we aslo take in this paper.

We have validated AutoInSpec on both synthetic programs and on non-trivial GUI applications with as many as 45K lines of code. Our results show that we learn some very interesting invariants, previously unknown to us despite using the same applications in many prior studies.

This paper makes the following research contributions:

- It presents an inference specification approach for model based-GUI testing that is based on a systematic combinatorial exploration of the event-space.
- We have formalized and encoded as logic templates a class of GUI invariants that are at the same abstraction level as the user interface, so that they can be easily evaluated with an off-the-shelf solver.
- We show that a prototype of our approach is feasible on a set of non-trivial GUI applications, finding previously unknown invariants in well-studied applications.

The next section presents background and a motivating example. Section III presents AutoInSpec and formally defines the invariants. We present a description of our implementation in Section IV along with a case study and its results to evaluate our approach in Sections V and VI. Section VII discusses related literature. Finally, we conclude and discuss future work in Section VIII.

II. Background and Motivating Example

There has been extensive research on developing models for GUI testing (please see [1] for an overview of the various models and [2]–[4], [11], [12] for more general information on GUI testing). In this section, we describe our earlier work on combinatorial based coverage criteria for fault detection, [12] because it is the most relevant to this work. Since our existing techniques at the time resulted in an unacceptably large number of infeasible test cases, we developed a framework to improve coverage (or *repair* the test suites) [11]. We also observed a set of constraints on combinations of events that we identified in real programs; we formalize these as invariants in Section III. This prior work, summarized next, forms the background for our current work.

We use a running example (Figure 1) to help summarize our prior work. The left part of Figure 1 shows two sequences (moving from top to bottom) of events in *Paint*. The first sequence begins by choosing the *Select All* option on the menu (top-left screen shot). This selects all of the objects on the canvas which can be seen in the dotted rectangle. If the user then selects *Copy To* from the menu, a dialog box opens (the left-bottom screen shot). The second screen shot sequence shows *Copy* preceding *Copy To*, but no dialog box appears. This second behavior is, in fact, the more common of the two, which happens when *any event* except *Select All* directly precedes *Copy To*. A complete specification for this GUI should include *'when Copy To is directly preceded by Select All, a dialog box will open'*. Let us assume, that this has been missed in the extracted model, and our test generator/harness is unaware of this invariant.

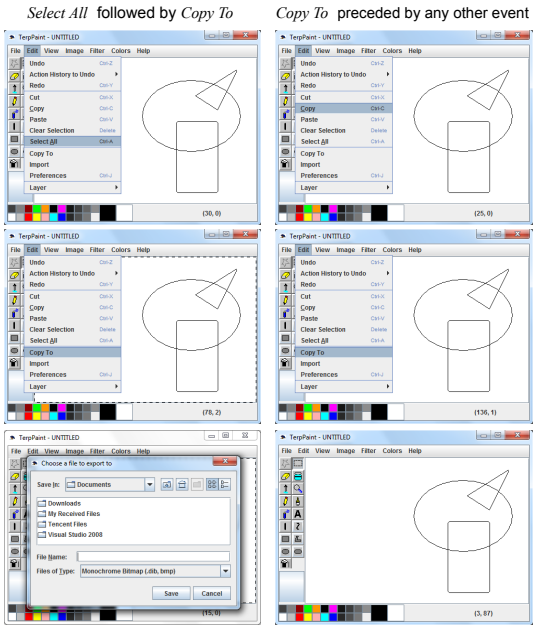
A. Combinatorial Coverage

For a GUI, G, with a set of events E_G, c is a finite sequence $< e_0, e_1, \ldots, e_{k-1} >$, where $e_i \in E_G, 0 \leq i < k$, and k is the length of the sequence. For convenience, we use $l(c)$ to denote the length of c. We also use $c_{G,k}$ to denote the set of all such length-k event sequences. For a GUI, G, C_k is a set of event sequences of the same length k for G. Formally, $C_k \subseteq c_{G,k}$.

A *covering array* ($CA(N; t, k, v)$) is an $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t of the v symbols *at least* once [13]. In other words, any subset of t columns of this array will contain all t-way combinations of the symbols. We use this definition of a covering array to define the GUI event sequences. We view the same event in different positions in the sequence as different events.

Suppose we want to generate sequences of length four, and each location in this sequence can contain exactly one of three events (*Copy*, *Copy To*, and *Select All*). There are a total of 81 sequences. We can instead sample this system, including all sub-sequences of shorter size, (length two) as a $CA(9; 2, 4, 3)$; we have 9 test sequences, and we cover all 2-way combinations in all locations at least once. There are 54 pairs that should be covered in this sample. The *strength* of event combinations is t. We set t=2 in the example, and include all pairs of events between all pairs of locations.

The v symbols are not necessarily the same for each column; each can have its own (different) v symbols (or values). One of all ordered subsets of size t of the v symbols appearing on t of the k columns is called a *t-set*. The total number of t-sets for a $CA(N; t, k, v)$ is $\binom{k}{t}v^t$. Given the strength t, the number of columns k, and the number



1. Part of Initial Covering Array

No	Event Sequences											Fail Point
1	Draw	Paste	Clear Select	Redo	Copy To	Draw	Select All	Copy To	Select	Clear Select		8
2	Select All	Free Select	Redo	Copy To	Free Select	Redo	Select	Copy	Paste	Redo		N/A
3	Paste	Copy To	Redo	Paste	Select All	Copy To	Free Select	Undo	Copy To	Select All		6
4	Select All	Copy To	Clear Select	Clear Select	Select	Copy	Copy To	Clear Select	Undo	Cut		2
5	Copy To	Clear Select	Undo	Copy To	Select All	Cut	Clear Select	Paste	Select	Clear Select		N/A
6	Copy To	Cut	Paste	Select All	Clear Select	Copy To	Cut	Free Select	Clear Select	Clear Select		N/A
7	Copy	Select All	Copy To	Select	Select	Draw	Free Select	Clear Select	Redo	Draw		3
8	Redo	Draw	Free Select	Free Select	Undo	Redo	Paste	Select All	Copy To	Select All		9
9	Undo	Select	Select All	Copy To	Free Select	Select All	Undo	Select All	Select All	Paste		4
10	Select	Free Select	Copy	Select All	Copy To	Clear Select	Cut	Select All	Select All	Draw		5
11	Copy	Copy	Cut	Paste	Select All	Copy To	Draw	Select All	Copy	Paste		7
...	...	...	...	...	...	...	...	...	...	...	...	...

2. Initial Missed t-sets

(Event, Position) pairs
{(Select, 0), (Select All,3)}
{(Free Select, 1), (Copy, 2)}
{(Free Select, 1), (Select All, 3)}
{(Select All, 3), (Copy To, 4)}
...

3. New Event Sequences from GA

No	Event Sequences											Fail Point
1	Select	Select	Copy	Select All	Free Select	Copy To	Undo	Clear Select	Free Select	Select All		N/A
2	Draw	Free Select	Copy	Select All	Select All	Draw	Select All	Select All	Copy To	Draw		N/A
...	...	...	...	...	...	...	...	...	...	...	...	...

4. Final Missed t-sets

(Event, Position) pairs
{(Select All, 0), (Copy To, 1)}
{(Select All, 1), (Copy To, 2)}
{(Select All, 2), (Copy To, 3)}
{(Select All, 3), (Copy To, 4)}
{(Select All, 4), (Copy To, 5)}
{(Select All, 5), (Copy To, 6)}
{(Select All, 6), (Copy To, 7)}
{(Select All, 7), (Copy To, 8)}

Figure 1. Motivating Example

of possible values v, the number of t-sets covered by an arbitrary $M \times k$ array A is called its *combinatorial coverage*.

Consider the *Paint* example shown in Figure 1. The set of 11 events of interest is {*Draw, Select All, Select, Free Select, Clear Select, Copy, Copy To, Undo, Redo, Cut, Paste*}. The right part of the figure (1.) shows the initial covering array, $(CA(171; 2, 10, 11))$. It has a total of 171 event sequences, of which 11 are shown; each sequence has 10 events. For t=2, these sequences have to cover $\binom{10}{2}11^2$=5445 t-sets to satisfy our coverage criteria.

B. Test Suite Repair

When we try to execute all sequences from the covering array on the GUI, some fail because of the missing specification. We see this in Figure 1 (1.) in the column marked *Fail Point*. This entry is N/A if the test runs to completion and the number of the last executable event otherwise. In the first sequence the test stops running on the eighth event, *Copy To*, as it waits for the dialog box to close – an occurrence that will not happen without previous model knowledge. Some t-sets are missed due to this and other problems; a few are shown in the *Initial Missed t-sets* (2.) part of the figure.

In [11], we developed a framework for repairing test suites to improve combinatorial coverage. The input to the framework is the GUI model and an initial set of test sequences in the form of a covering array. We use a genetic algorithm (GA) to generate new sequences that increase the coverage, while avoiding failed sub-sequences.

In Figure 1 (3.), we see some new sequences were obtained as a result of the GA repair, which covers all the t-sets that we had missed earlier, with the exception of the pairs *Select All, Copy To* in each of the possible consecutive locations. This results in 8 missed pairs in positions, 0 and 1, 1 and 2, 2 and 3, 3 and 4, ..., 7 and 8. This set of missed t-sets is shown in *Final Missed t-sets* of Figure 1 (4.). These t-sets will never be covered due to the design of the GUI. *It is this insight that helps us to reduce the problem of invariant detection, focusing only on what has not been covered.*

We also created a classification of some types of infeasible sequences that we observed in several real applications. We term these *event constraints* or simply constraints. We suggest four broad categories: *disabled*, *requires*, *consecutive* and *excludes*. The *Disabled event constraint* occurs when an event is always disabled. The *Requires constraint* indicates that some event needs another event to be executed before it is enabled. The *Event Consecutive constraint* means that two events cannot be executed consecutively. Usually, in this type of constraint, the execution of the first event disables the second event, making it unexecutable. The *Excludes constraint* is similar, however once the first event has been enabled there is no way to re-enable the second event within the current group of events. We use these as a starting point to formalize our invariants in the next section.

III. AutoInSpec

We present an overview of *AutoInSpec*'s process in Figure 2. There are four key steps in AutoInSpec, numbered in boxes. We begin (#1) with a test suite that is coverage adequate. The use of a systematic and broad coverage and sequences which are long enough to expose context [12] provides our starting point. Since we are forcing all combinations of events up to some strength t, and are doing this for

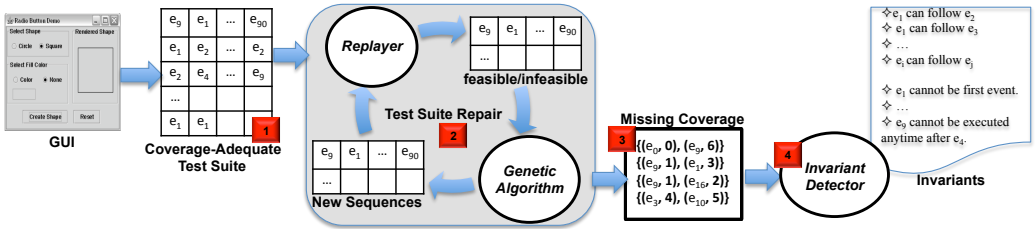


Figure 2. AutoInSpec: Invariant Inference Process

every position in the sequence, we have confidence that we are executing a broad range of scenarios. We also know that the missing coverage is the *only coverage* that violates our model; we can assume anything that did not fail is correct. If some of the test sequences do not execute to completion, then we pass these to our test suite repair framework (#2), which tries to complete the missing coverage. Although the normal output of this stage is a coverage-adequate test suite, we return only a by-product – the set of event-combinations that we cannot cover (#3). This is in the form of a list of t-sets combined with their relative positions in the sequence. These are then passed to our invariant detection engine (#4) which suggests the unknown specifications.

We now provide definitions and examples of the invariants that we have considered in this work. In all our definitions, k is the length of the sequences, e and its subscripted forms are events from the event set E, p and its subscripted forms are integers $0 \ldots k-1$.

Our repair framework returns a set, χ, of missed t-sets, each of the form $\{(e_{x_1}, p_1), (e_{x_2}, p_2), \ldots, (e_{x_t}, p_t)\}$, where $1 \leq t \leq k$, and $0 \leq p_1 < p_2 < p_3 < \ldots p_t \leq k-1$. We use the notation $\phi(s)$ to say that the t-set $s \in \chi$ was missed.

Definition: *disabledAtPosition(Event,Position)* invariant. Intuitively, this invariant, read as *"Event is always disabled at Position in all sequences"* holds if and only if all t-sets that contain ($Event$, $Position$) are missed. More formally,

$disabledAtPosition(e, i)$ *iff* $\phi(\{(e, i)\})$.

Definition: *disabled(Event)* invariant. Now, we can define the $disabled$ invariant in terms of $disabledAtPosition$. Intuitively, $Event$ is disabled if and only if all t-sets that contain it—irrespective of the position of $Event$—are missed. Formally,

$disabled(e)$ *iff* $\forall_{i=0\ldots k-1} disabledAtPosition(e, i)$.

Definition: *consecutive(EventSequence)* invariant. This invariant holds if and only if all t-sets that "contain" $EventSequence$, $< e_1, e_2, \ldots, e_t >$, are missed. By "contain", we mean that $e_1, e_2, \ldots, e_t$ appear in consecutive positions. Formally,

$$\begin{aligned}&consecutive(< e_1, e_2, \ldots, e_t >) \textit{ iff}\\ &\quad \forall_i \neg(disabled(e_i)) \wedge\\ &\quad \forall_{i=0\ldots k-t-1} \phi(\{(e_1, i), (e_2, i+1), \ldots, (e_t, i+t-1)\})\end{aligned}$$

Definition: *excludes(EventSequence, Event)* invariant. Intuitively, $EventSequence$, $< e_1, e_2, \ldots, e_t >$, excludes $Event$ if and only if all t-sets in which $EventSequence$ precedes $Event$ are missed. Formally,

$$\begin{aligned}&excludes(< e_1, e_2, \ldots, e_t >, e) \textit{ iff}\\ &\quad (\neg(consecutive(< e_1, e_2, \ldots, e_t >)) \wedge\\ &\quad (\forall_i \neg(disabled(e_i))) \wedge\\ &\quad \forall_{i=0\ldots k-t-2} \exists_p (i+t \leq p \leq k-1) \wedge\\ &\qquad \phi(\{(e_1, i), (e_2, i+1), \ldots, (e_t, i+t), (e, p)\}))\end{aligned}$$

Definition: *requires(EventSequences, Event)* invariant. Whenever a t-set that contains $Event$ is not missed, it is only when $Event$ is preceded by a member of the set $EventSequences$. In other words, all other t-sets that contain $Event$ will be missed. Formally,

$$\begin{aligned}&requires(\mathrm{SetOfSequences}, e) \textit{ iff}\\ &\quad (\forall_{e_i \in \mathrm{SetOfSequences}} \neg(disabled(e_i)) \wedge \neg(disabled(e) \wedge\\ &\quad \forall_{\{(e_1,i),(e_2,i+1),\ldots,(e_t,i+t)\} \in (U - \mathrm{SetOfSequences})}\\ &\qquad \phi(\{(e_1, i), (e_2, i+1), \ldots, (e_t, i+t), (e, p)\}))\end{aligned}$$

where U is the set of all event sequences.

IV. Invariant Detector

We use the logic programming language Prolog to implement our invariant detector and describe its implementation as such in pseudo Prolog code, but we can use other off-the-shelf logic engines to achieve the same goal. Using Prolog gives us certain advantages and makes our code more readable. First, it is based on the declarative programming paradigm, we are able to describe what our program should accomplish—as declarative clauses and facts—rather than how it should accomplish it. Hence, we are able to cleanly transcribe the formal definitions into executable code. Second, Prolog's inference engine is based on unification, an algorithmic process that attempts to find a substitution which makes two terms the same. As we will demonstrate via an example, this is a powerful tool. Third, because Prolog already combines pattern matching with backtracking, trying out all possible solutions, we do not need to explicitly write code to compute all possibilities for inference, making our code simpler.

Finally, we can simply 'turn on' a flag to obtain a "proof" of Prolog's reasoning, and output the reason why a particular invariant holds. For our example from Section II,

the program's output may be *"Select All cannot precede Copy To" because the following t-sets were missed: {(Select All, 0), (Copy To, 1)} {(Select All, 1), (Copy To, 2)} {(Select All, 2), (Copy To, 3)} {(Select All, 3), (Copy To, 4)} {(Select All, 4), (Copy To, 5)} {(Select All, 5), (Copy To, 6)} {(Select All, 6), (Copy To, 7)} {(Select All, 7), (Copy To, 8)}*. Such detailed reasoning is helpful both for debugging and for manual verification of the existence of the invariant.

We will use a running example to explain our implementation. Suppose that we have the following 18 missed t-sets, for t=2, from our test suite repair: { {(a, 0), (a, 1)}, {(a, 0), (b, 1)}, {(a, 0), (c, 1)}, {(a, 0), (a, 2)}, {(a, 0), (b, 2)}, {(a, 0), (c, 2)}, {(a, 1), (a, 0)}, {(a, 1), (b, 0)}, {(a, 1), (c, 0)}, {(a, 1), (a, 2)}, {(a, 1), (b, 2)}, {(a, 1), (c, 2)}, {(a, 2), (a, 1)}, {(a, 2), (b, 1)}, {(a, 2), (c, 1)}, {(a, 2), (a, 0)}, {(a, 2), (b, 0)}, {(a, 2), (c, 0)}}.

Also, suppose that we have only length 3 sequences; and our event set consists of 3 events, a, b, and c. The 18 missed t-sets form the input to our Prolog program. They are represented as a collection of 18 Prolog *facts*, one fact for each t-set. For example, the t-set {(a, 2), (c, 1)} is represented as the fact phi_t_2(a, 2, c, 1), where "phi_t_2" is our name for the fact. In all, we have 18 such facts that encode our input for this example.

Program execution in Prolog is done via queries. An example of a query is "?:-phi_t_2(A, X, B, Y)." where the "?" indicates a query, and A, B, X, and Y are uninstantiated (or unbound) variables that can be unified with literals. Given this query, the inference engine tries to find all possible solutions. It will "look at" all facts that match phi_t_2 and attempt to instantiate all variables. One possible instantiation is "A=a X=2 B=c Y=1" when a match is made with the fact phi_t_2(a, 2, c, 1). Because our particular query has all uninstantiated variables, they can be bound to any literal; and because Prolog will find all possible solutions, it will return 18 solutions, one for each fact that matches the query.

Another example is "?:-phi_t_2(a, X, a, Y)." where the first and third parameter are not variables; they are literals, in this case 'a'. In our facts listed above, there are 6 facts that have 'a' in parameter 1 and 3. Hence, we get 6 solutions. Prolog allows many types of complex queries; a more detailed discussion is beyond the scope of this work.

We now define the *disabledAtPosition* invariant as:

```
disabledAtPosition(Event, Position, K):-
 foreach(event(E),
  foreach(between(0, K-1, P),
   P != Position implies
    phi_t_2(Event, Position, E, P))).
```

i.e., the invariant holds for **Event** at **Position** if there exist a number of t-sets, enumerated by the two foreach clauses; the first enumerates all events **E** and the second enumerates all possible positions **P** between 0 and **K-1** (the largest value for an event's position). For our example of 3 events, the first foreach enumerates, in **E**, events a, b, and c. Because for us, **K**=3, the second foreach enumerates in **P** the values 0, 1, and 2. Hence, the two foreach are really a compact way of enumerating a number of required facts.

We further explain using two example queries. If we are interested in finding all events that are disabled at position 1, we issue the query "?:-disabledAtPosition(X, 1, 3)." We leave the event, the first parameter X, unbound for Prolog to return its solution. The second parameter is the position 1, and third is the length of our sequences. In the body of our definition of disabledAtPosition, the two foreach enumerate $3 \times 3 = 9$ possibilities as discussed above, and "**P**!=**Position**" filters out value '1' for **P**; finally, there are 6 enumerations of phi_t_2(**Event**, 1, **E**, **P**), for **E** having values a, b, and c; **P** with values 0 and 2. All 6 match only when **Event**=a. Because **Event** and **X**, in our query, are unified, both unbound, Prolog's solution is "X=a".

Lets suppose that we issue another query "?:-disabledAtPosition(X, Y, 3)" where we also leave the position unbound. We are asking Prolog to return all events X that are disabled in any position Y. This query has 3 solutions: "X=a Y=0", "X=a Y=1", and "X=a Y=2", i.e., event 'a' is disabled at positions 0, 1, and 2.

In fact, in our example, event 'a' is disabled at all positions, which naturally leads us to our encoding of *disabled*.

```
disabled(Event, K):-
 foreach(between(0,K-1,Position),
  disabledAtPosition(Event,Position, K)).
```

which checks whether **Event** is disabled for all values of **Position** between 0 and **K**-1. The query "?:-disabled(X, 3)" returns "X=a".

We note that our definition for *disabled* uses *disabledAtPosition*. In general, because the definitions of some invariants use other invariants, the inference process is done in a specific order and intermediate results are stored in a *learned database* which is read before the next rule processes. Our implementation infers invariants in the order we have defined them, i.e., *disabledAtPosition*, *disabled*, *consecutive*, *excludes*, and *requires*.

Because of the nature of event execution using a test case replayer, certain timing conditions or close events may cause incorrect reporting of missed sequences. In fact, we see this in our real example of {*Select All, Copy To*}. Since the test harness closes after the last event is triggered, the open dialog will not impact execution when it is in the last position. Suppose that due to replayer timing conditions {*(a, 0), (b, 1)*} in the above example is not reported as a missed t-set. Our previously described encoding would prevent us from inferring the invariant. The query "?:-disabledAtPosition(X, 0, 3)", asking "*which event* X *is disabled at position* 0*?*" would return no result. The query "?:-disabledAtPosition(X, Y, 3)" would return only 2 solutions: "X=a Y=1", and "X=a Y=2".

And the query "?:-disabled(X, 3)" would return no result. All this because of an unreported missed t-set.

In order to handle such unexpected replayer-related problems, and still get some useful information from the invariant detector, our particular implementation allows for some *error*. We have rewritten our Prolog programs to work with imperfect data; i.e., some number of t-sets may be misclassified as not missed. The key to adding fuzzy-matching logic to our implementation is to maintain two integer values with each inferred invariant: (1) **T**, the total number of t-sets expected to be reported as missed for the inference and (2) **Er**, the number of t-sets that were expected to be reported as missed but were not. We then compute the *error percentage* as $(\mathbf{Er}/\mathbf{T}) * 100$.

V. Case Study

We conduct a case study to evaluate AutoInSpec using a set of well-understood applications which have been used in multiple papers on GUI testing. Experimental artifacts including inference inputs and results can be found on the AutoInSpec website[1]. We answer the following research questions in this study:

RQ1: How well does AutoInSpec uncover seeded invariants?

RQ2: To what extent does AutoInSpec find invariants that are unknown?

RQ3: How well does the AutoInSpec invariant detecter classify real invariants with respect to a manually derived oracle?

A. Objects of Study

We use two sets of Java GUI subjects for these experiments. The first set are used for RQ1 and were originally created for validating the quality of the repair framework in improving the coverage of our sequences [11]. These applications are synthetic – their only functionality is to implement a specific invariant. We call them seeded. We provide a summary of these in Table I and point the reader to [11] for more details. The first subject, `2Cons` contains a single 2-way consecutive invariant, *Event 0* cannot precede *Event 1*. `2Excl` has a 2-way invariant that disables *Event 1* after *Event 0* is performed. Likewise, the `Disb` program has a disabled event and the `Reqs` contains a 2-way requires invariant (*Event 2* requires *Event 1*). The last program, `Cmpd` contains three invariants.

For the other two research questions, we have selected a set of non-trivial Java GUI subjects that we have used in earlier work [12]. Unlike the synthetic subjects, we do not know the true invariants. The selected subjects include four applications from the TerpOffice series and two open source programs from SourceForge – CrosswordSage and FreeMind. TerpPaint, TerpPresent, TerpWord and TerpSpreadSheet are office-like applications that are used for painting, presentation, word processing and spreadsheet processing, respectively. CrosswordSage is a tool for creating professional looking crosswords with powerful word suggestion capabilities, and FreeMind is a mind-mapping software.

[1]http://www.cse.unl.edu/~myra/artifacts/autoinspec_2012/

Table I
SYNTHETIC APPLICATIONS

No.	Name	#Events	2-way requires
1	`2Cons`	3	2-way consecutive
2	`2Excl`	3	2-way excludes
3	`Disb`	3	disabled
4	`Reqs`	3	2-way requires
5	`Cmpd`	5	(compound) Reqs, 2Cons, 3Cons(3-way consecutive)

The events in these programs were divided [12] into subgroups of events with like functionality. This is one way that a designer/maintainer/tester may reduce the complexity of the overall task while combining events which are likely to interact. The numbers of events in each group are listed in Table II. Each group focuses on one specific task. For example, events in group 1 of TerpWord 3.0 (abbreviated as TW-G1) are all related to Table operations; events in group 2 of FreeMind 0.80 (abbreviated as FM-G2) are all used for formatting the map and displays. We also show the lines of code for each subject in the table. We used the length 10, 2-way covering arrays from [12] as our starting point. For this study, we selected the groups that had missing coverage, and that we were able to run and reproduce.

B. Methodology

We reuse existing covering arrays from [12], but rebuild the models for the programs using the GUITAR framework [14]. Due to some changes in versions of Java and the change in the replayer from the original work we had a small number of events that we could not replay. We set these to NULL. These account for 2 events out of 114 in our experiments. These are passed into our repair framework [11] which utilizes the CASA covering array generator for combinatorial coverage [15]. We use t=2. This is the weakest (least expensive) coverage, but believe most invariants will be of this arity.

In [12], preliminary efforts had been made to improve the models manually by first running all possible length-2 event sequences. Enabling events have been identified and added to reveal constraints on the structural operations. For example, before a *Save As* dialog is closed, no events on the parental window can be performed. Therefore, an event *Save* will be added after all the necessary operations on the *Save As* dialog to close it. We incorporate these events into our starting model. Some non-structural enabling events had also been added, which are invariants that were discovered through iteration with the subjects [12]. For example, when there is an *Undo* at the beginning of a sequence, it is ignored and the rest of the sequence is executed. We do not

Table II
NON-TRIVIAL APPLICATIONS

No.	Program Name	LOC	Group	No. Events	Abbreviated Name	Task Description
1	TerpPaint 3.0	13315	4	11	TPa-G4	Clipboard Operations
2	TerpPresent 3.0	44591	5	14	TPr-G5	Content
3	TerpWord 3.0	22806	1	14	TW-G1	Table Operations
4	TerpSpreadSheet 3.0	6337	1	14	TS-G1	Format Cell
5			5	8	TS-G6	Table Format
6	CrosswordSage 0.35	3220	4	14	CS-G4	Preference Settings
7	FreeMind 0.80	24665	1	11	FM-G1	Map Operations
8			2	18	FM-G2	Format
9			4	10	FM-G4	Clipboard Operations

include these in our models, but leave them as invariants that AutoInSpec should find.

We use the missed t-sets to detect invariants in two ways. First we asked one of the co-authors of this work to independently determine (manually) the invariants based on the missed t-sets. This was done by examining the patterns and running the applications. We use this as our oracle. We then passed these to our Prolog programs using an error of 25% (which we chose heuristically), that provided us with another set of invariants. We recorded these and the remaining two authors validated each against the oracle, and by running the application to confirm differences. All of the actual output is available on the website containing experimental artifacts.

C. Threats to Validity

We list some of the primary threats to validity here. First, since the authors were involved in developing the oracle (manually determining invariants in the real applications) this may cause a bias. To mitigate the threat, only one author was responsible for that part and was most familiar with the low level details of each application. The other two authors independently developed and evaluated the invariants with the automated technique without discussion or interaction with the other author. We also acknowledge that a different set of invariants may provide similar results, but leave this as a future evaluation.

Second, we cannot know if our results will generalize to all applications, but have selected applications that have been used in other testing research. Third, for the real applications, some of the invariants found may be an artifact of the event grouping. We believe that this threat does not invalidate our results since we are only concerned with the detection of invariants, and the modeling was done for a different study and considered to be correct by the person who performed that work (not one of the authors of this paper). The use of existing artifacts reduces other threats, which was our reason for using them. We plan to reduce the grouping threat by removing it in future experiments. Finally, we have validated our internal programs used in this work to a reasonable degree and have made our artifacts available to other researchers, but it is possible that there may be some faults remaining in the implementations.

VI. RESULTS

In this section we present the results of our case study and answer each of our research questions in turn.

A. RQ1: How well does AutoInSpec uncover seeded invariants?

We ran AutoInSpec on the synthetic subjects and present these results in Table III. For each subject (row) we list the invariants that were detected in order. For the first four subjects, we see that the invariants match exactly with the expectation. For instance, in `2Cons` the invariant *E0* cannot precede *E1* is the known invariant for this subject. For the last subject, `Cmpd` we see a sightly different result. In this subject we have three invariants so this should help us understand how well AutoInSpec works when we have more than one invariant together. Two of the three invariants were correctly detected (the *Requires* and *Consecutive*). We see that *E4* requires *E2* and that *E0* cannot precede *E1*. Both of these are expected. The last invariant found (*Disabled*), says *E4* is disabled in position 0. This invariant is a result of the first one – since *E4* requires *E2* it cannot be in the first position and strengthens the behavior specification, but it is not something that we considered prior to this experiment. We also missed one invariant present in this program (a 3-way *Consecutive* invariant), but since it is a 3-way invariant and we used a 2-way covering array, we did not expect to find it. The arity of invariants will impact our ability to detect them. We need to perform more experimentation to understand the arity of invariants in practice.

Summary of RQ1: AutoInSpec is effective at finding the known invariants in isolation, as well as when invariants exist together. It also found an additional invariant that we did not expect, which strengthens what we know about the behavior of the `Cmpd` application.

B. RQ2: Does AutoInSpec discover unknown invariants?

For this RQ we turn to Table IV. It shows the manual oracle on the left side and the invariants we found with

Table III
INVARIANTS FOUND IN THE SYNTHETIC APPLICATIONS

Subj.	Invariants			
	Disabled	**Excludes**	**Requires**	**Consecutive**
`2Cons`	none	none	none	E0 can't precede E1
`2Excl`	none	E0 Excl. E1	none	none
`Disb`	E0 disb.	none	none	none
`Reqs`	none	none	E2 req. E1	none
`Cmpd`	none	E4 disb. at pos. 0	E4 req. E2	E0 can't precede E1

AutoInSpec on the right. In the middle we use a `Y` to indicate that our invariants match, `N` to indicate they do not, and an `S` in the special case where one invariant is a subset of the other. We return to this column in RQ3.

AutoInSpec found 19 invariants in these subjects (the oracle found 25). Some of these invariants, such as the first one for TW-G1, *Undo cannot appear in the first position*, is an invariant that we knew about before using AutoInSpec. We still consider this a discovered invariant, but believe it is one that would be easy for us to model. However, only a small number of the invariants fit into this category.

The rest are invariants that were not known and/or modeled in our prior work using these subjects [12]. Given that these are very well studied applications which have been used by us before, we find this result compelling. For instance we were unaware of a dialog opening when *Select All* is followed immediately by a *Copy To*, creating the first invariant that we find in this table.

Another interesting observation is that we have two groups with events that we thought were in the same group, in fact, were not. This can be seen in FM-G2 where the *Cloud Color* event can never be enabled and TS-G1 and TS-G5 where *Undo* and *Redo* are never enabled. In the original experimentation, these invariants were not obvious to the modeler, and despite the existence of infeasible sequences, were not detected.

An example of another invariant that we missed during modeling is in CS-G4. In this invariant, a set of fields require a specific check box to be checked before they can be enabled. This box is the first box on the window so the automated model extraction tool would visit this box first and assume the rest of the fields are enabled. However, the default for this window when opened is that this box is unchecked. Without having tried the extra combinations, we would not catch this invariant.

Summary of RQ2: We conclude that we are able to discover unknown invariants using AutoInSpec.

C. RQ3: How does the AutoInSpec invariant detector compare with a manual oracle on real invariants?

The last question asks how well we compare with the manual oracle. We match 14 of the 25 invariants the oracle found exactly. (In CS-G4, we found 2 invariants, while the oracle found only one, bringing the total invariants to 26 – we use this as our denominator in calculations). Seven invariants were not found at all. In the remaining five cases, we have a subset match. In each case, either one of the oracle or AutoInSpec is a subset of the other. We have made the events bold-faced that are not found in the other. For instance, in TW-G1, *Undo* needs one of a set of events to enable it. The oracle includes *Redo* while AutoInSpec missed this. We believe this is due to the other invariants in this group. Since *Redo* cannot appear in the first two positions and has an additional invariant requiring an *Undo*, it is possible there was too much noise in the dataset to find this. In the same subject, we see in the last invariant that *Redo* cannot precede *Redo*. We determined that this invariant is correct; the manual oracle missed this one.

Summary of RQ3: AutoInSpec matches 54% of the oracles exactly. In 19% of the cases, it matches partially, sharing most of the events in common. In total, it missed 27%.

VII. RELATED WORK

Software specification inference is used to discover the behavior of software systems when specifications are missing or incomplete. Unlike our work, most inference algorithms use system traces; we first derive a subset in the form of missing t-sets. The class of specifications on which we focus are temporal or state-based. Gabel and Su mine temporal properties to find bugs [16], while Ramanathan et al. use inter-procedural path-sensitive static analysis to infer function precedence protocols [17]. Krka et al. [18] utilize inferred program invariants and method invocation sequences to obtain an object-level model. Others use inference techniques for discovering interactions between components [19].

Ernst et al. [9], [10] lay the foundations for dynamic detection of invariants. They instrument variables within the code to detect invariants, but we work at the black box level. Recent work on database invariants [20], considers a state-based environment, but defines a mapping between database elements and variables; we require no such mapping. Data from repositories may be used to infer certain classes of specifications. Wasylkowski and Zeller mine temporal specifications by combining static analysis with model checking [21]. Livshits and Zimmermann extract likely error patterns by mining software repositories [22]. Xie and Pei develop algorithms for mining API usages from code [23]. Other

Table IV
INVARIANTS LEARNED. MATCH IS ONE OF (Y)ES, (N)O OR S(UBSET). * INDICATES THAT THE INVARIANT WAS KNOWN AHEAD OF TIME

Subject	Oracle	Match?	Automated Invariant
TPa-G4	*Select All* and *Copy To* cannot execute consecutively	Y	*Select All* cannot precede *Copy To*
TPr-G5	Two consecutive *Opens* cannot be executed	Y	*Open* cannot precede *Open*
	Open, *Save* and *Save As* cannot be immediately followed by *Save*	Y	*Open*, *Save*, and *Save as* cannot precede *Save*
	Insert Image, *Undo* and *Redo* cannot be immediately followed by *Close* if they are in the first position	N	**no invariant found**
TW-G1	**Undo* cannot appear in the first position	Y	*Undo* always disabled at position 0
	Undo needs one of *Insert Table*, *Append Row*, *Append Column*, *Insert Row*, *Insert Column*, *Delete Row*, *Delete Column*, **Redo**, *Write on Document Pane* or *Write on HTML Pane* to enable it	S	*Undo* requires one of *Insert Table*, *Append Row*, *Append Column*, *Insert Row*, *Insert Column*, *Delete Row*, *Delete Column*, *Write on Document Pane* *Write on HTML Pane*, **Next Cell** or **Previous Cell**
	Each *Undo* event needs an undoable before it	N	**no invariant found**
	**Redo* cannot appear in the first or second position	Y	*Redo* always disabled at position 0 and 1
	Redo needs an *Undo* to enable it	N	*Redo* requires *Previous Cell* or *Write on syntax pane*
	Delete Row cannot be followed by any table-related operations	N	**no invariant found**
	Insert Table, *Append Row*, *Append Column*, *Insert Row*, *Insert Column*, *Delete Row*, *Delete Column*, *Write on Document Pane* and *Write on HTML Pane* cannot be followed by *Redo*	S	*Insert Table*, *Append Row*, *Append Column*, *Insert Row*, *Insert Column*,*Delete Row*, *Delete Column*, *Write on Document Pane* and *Write on HTML Pane*, and **Redo** cannot precede *Redo*
TS-G1	*Undo* and *Redo* are disabled	Y	*Undo* and *Redo* are disabled
TS-G5	*Undo* and *Redo* are disabled	Y	*Undo* and *Redo* are disabled
CS-G4	*Proxy Address*, *Proxy Port*, *User Name* and *Password* require *Use Proxy* to be checked (unchecked at startup)	Y	*Proxy Address*, *Proxy Port*, *User Name* and *Password* are disabled at position 0
		S	*Proxy Address*, *Proxy Port*, *User Name* and *Password* require **Auto-check for Newer Version** or *Use Proxy Server* or **New Crossword** or **Solve New Word**
FM-G1	**Undo* cannot appear in the first position.	Y	*Undo* is always disabled at position 0
	Undo needs one of *Automatic Layout*, *Blinking Node* or *Show Icon Hierarchically* to enable it	S	*Undo* requires one of **Scale**, **Toggle Toolbar**, **Toggle Left**, **Zoom In**, **Zoom Out**, **Zoom to Fit to Page**, *Automatic Layout*, *Blinking Node* or *Show Icon Hierarchically*
	**Redo* cannot appear in the first or second position	Y	*Redo* is always disabled at position 0 and 1
	Redo needs an *Undo* to enable it	N	**no invariant found**
FM-G2	*Cloud Color* is disabled	Y	*Cloud Color* is disabled
FM-G4	**Undo* cannot appear in the first position	Y	*Undo* is always disabled at position 0
	Undo needs *Paste* to enable it	S	Undo requires one of *Paste*, **Paste Format**, **Cut**, **Copy**, **Copy Single**, **Select Visible Branch**
	Each *Undo* event needs an undoable before it	N	**no invariant found**
	**Redo* cannot appear in the first or second position	Y	*Redo* always disabled at position 0 and 1
	Each *Redo* needs an *Undo* before it	N	**no invariant found**
	Paste cannot be immediately followed by *Redo*	Y	*Paste* cannot precede *Redo*

work in the domain of GUIs, usability evaluation, of Dwyer et al., describes how static analysis may be used to reason about user-interaction properties of GUIs [24].

AutoInSpec is unique in that its reduces the inference problem by considering only missed coverage, and it uses an off-the-shelf solver. Furthermore it works entirely from the blackbox perspective.

VIII. CONCLUSIONS

We have presented AutoInSpec, a technique to automatically uncover a class of GUI invariants, (temporal and state-based invariants which are incorrectly modeled). Our approach leverages covering arrays and a test suite repair process. The covering arrays give us fine-grained control over the event sequences, allowing us to precisely specify how they should be synthesized. By using missed coverage, we have a mechanism to examine only sequences disallowed by the GUI. We used AutoInSpec to uncover invariants in both synthetic and real applications. We found 100 percent of the known invariants for our synthetic applications that were discoverable by our chosen covering array strength, and found an additional invariant that tightened our knowledge

of the GUI behavior. In the real applications, we found 25 invariants (14 of which we matched exactly). We missed 27% but most of these were of a similar type of invariant. We plan to add that invariant class as future work.

We have identified several factors to explore as future work. The length of the sequence will impact not only the types of invariants found, but the precision of our results. The covering array strength controls how well we sample the sequences and the error percentage allows for consideration of imperfect data. Finally, the human must play a role in this work, as discovered specifications may be intended or may be faulty behavior. We intend to study the impact of these factors and incorporate the results into AutoInSpec.

ACKNOWLEDGMENTS

We thank B. Nguyen for help with the replayer infrastructure and X. Yuan for sharing her experimental artifacts. This work was partially supported by the US National Science Foundation under grants CCF-0747009, CNS-1205472, CNS-0855139, CNS-1205501 and CNS-0855055, the Air Force Office of Scientific Research through award FA9550-10-1-0406, and the Office of Naval Research under grant N00014-05-1-0421.

REFERENCES

[1] F. Belli, M. Beyazit, and A. Memon, "Testing is an event-centric activity," in *Intl. Conf. on Soft. Secur. and Reliab., (SERE)*, 2012, pp. 198–206.

[2] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Soft. Testing, Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, 2007.

[3] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study: Research articles," *Softw. Test. Verif. Reliab.*, vol. 16, no. 1, pp. 3–32, Mar. 2006.

[4] L. White, H. Almezen, and S. Sastry, "Firewall regression testing of GUI sequences and their interactions," in *Intl. Conf. on Soft. Maint., (ICSM)*, 2003, pp. 398–408.

[5] T.-A. Doan, D. Lo, S. Maoz, and S.-C. Khoo, "LM: A miner for scenario-based specifications," in *Intl. Conf. on Soft. Eng., (ICSE)*, 2010, pp. 319–320.

[6] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *Intl. Conf. on Soft. Eng., (ICSE)*, 2006, pp. 282–291.

[7] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: better together," in *Intl. Conf. on Aut. Soft. Eng., (ASE)*, 2010, pp. 387–396.

[8] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining," *IEEE Trans. on Soft. Eng.*, vol. 38, pp. 243–257, 2012.

[9] J. H. Perkins and M. D. Ernst, "Efficient incremental algorithms for dynamic detection of likely invariants," *SIGSOFT Soft. Eng. Notes*, vol. 29, no. 6, pp. 23–32, 2004.

[10] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Intl. Conf. on Soft. Eng., (ICSE)*, 2000, pp. 449–458.

[11] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI test suites using a genetic algorithm," in *Intl. Conf. Soft. Test, (ICST)*, April 2010, pp. 245–254.

[12] X. Yuan, M. Cohen, and A. Memon, "GUI interaction testing: Incorporating event context," *IEEE Trans. on Soft. Eng.*, vol. 37, no. 4, pp. 559 –574, 2011.

[13] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *Intl. Conf. on Soft Eng., (ICSE)*, 2003, pp. 38–48.

[14] "GUITAR – a GUI Testing frAmewoRk," website, 2009, http://guitar.sourceforge.net.

[15] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Symp. on Search Based Soft. Eng. (SSBSE)*, May 2009, pp. 13–22.

[16] M. Gabel and Z. Su, "Symbolic mining of temporal specifications," in *Intl. Conf. on Soft. Eng., (ICSE)*, 2008, pp. 51–60.

[17] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *Intl. Conf. on Soft. Eng., (ICSE)*, 2007, pp. 240–250.

[18] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic, "Using dynamic execution traces and program invariants to enhance behavioral model inference," in *Intl. Conf. on Soft. Eng., (ICSE)*, 2010, pp. 179–182.

[19] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Intl. Conf. on Soft. Eng., (ICSE)*, 2008, pp. 501–510.

[20] J. Cobb, J. A. Jones, G. M. Kapfhammer, and M. J. Harrold, "Dynamic invariant detection for relational databases," in *Intl. Work. on Dyn. Anal., (WODA)*, 2011, pp. 12–17.

[21] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *Intl. Conf. on Aut. Soft. Eng., (ASE)*, 2009, pp. 295–306.

[22] B. Livshits and T. Zimmermann, *DynaMine: Finding Usage Patterns and Their Violations by Mining Software Repositories*. CRC Press, 2011.

[23] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Intl. Work. on Min. Soft. Repo., (MSR)*, 2006, pp. 54–57.

[24] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser, "Analyzing interaction orderings with model checking," in *Intl.Conf. on Aut. Soft. Eng., (ASE)*, 2004, pp. 154–163.