Automated Test Case Generation for an Autopilot Requirement Prototype

Dimitra Giannakopoulou, Neha Rungta, Michael Feary
NASA Ames Research Center, Moffett Field, CA 94035
{dimitra.giannakopoulou,neha.s.rungta,michael.s.feary}@nasa.gov

***Abstract*—Designing safety-critical automation with robust human interaction is a difficult task that is susceptible to a number of known Human-Automation Interaction (HAI) vulnerabilities. It is therefore essential to develop automated tools that provide support both in the design and rapid evaluation of such automation. The Automation Design and Evaluation Prototyping Toolset (ADEPT) enables the rapid development of an executable specification for automation behavior and user interaction. ADEPT supports a number of analysis capabilities, thus enabling the detection of HAI vulnerabilities early in the design process, when modifications are less costly. In this paper, we advocate the introduction of a new capability to model-based prototyping tools such as ADEPT. The new capability is based on symbolic execution that allows us to automatically generate quality test suites based on the system design. Symbolic execution is used to generate both user input and test oracles; user input drives the testing of the system implementation, and test oracles ensure that the system behaves as designed. We present early results in the context of a component in the Autopilot system modeled in ADEPT, and discuss the challenges of test case generation in the HAI domain.**

***Index Terms*—human computer interactions, symbolic execution, testing, formal methods**

I. Introduction

The design of automation behavior in the presence of user interactions is a complex task that is susceptible to a number of known Human-Automation Interaction (HAI) vulnerabilities. Such vulnerabilities can be generic, for example non-determinism or incompleteness in requirements; or the vulnerabilities can be specific to HAI, for example mode confusion. Analysis of such vulnerabilities is better performed during the early design stage, where both the analysis cost and the cost of making modifications are lower. It is a well known fact that the cost of finding and fixing errors in a system increases by an order of magnitude as we move from one phase of the software development process to the next.

It is essential to develop automated tools that provide support both in the design and quick prototyping of such automation. Several tools have been developed for performing analyses in this domain. In this paper, we focus on the Automation Design and Evaluation Prototyping Toolset (ADEPT) developed at NASA Ames [10]. ADEPT was constructed to enable the rapid development of an *executable* specification for automation behavior and user interaction. More specifically, a design of the system and its associated interface are described in ADEPT in a concise, hierarchical, and tabular form. This abstract model is used to generate a prototype of the system and its interface as a Java program.

ADEPT supports a number of analysis capabilities that help in creating robust designs and prototypes that can be used as requirements for the construction of the actual system implementation. This work advocates the introduction of a new capability to model-based prototyping tools such as ADEPT. We propose an approach based on symbolic execution to automatically generate quality test suites based on the system design. We generate both user input and test oracles; user input drives the testing of the system implementation, and test oracles ensure that the system conforms to the ADEPT specification. Even though our work is discussed in the context of the ADEPT toolset and symbolic execution project (SPF) of the Java Pathfinder tool (JPF), the approach is general. The ideas presented in this work can be applied to other HAI design toolsets where the specification models have clear semantics. Similarly, a wide variety of tools that support symbolic execution can be used to generate the test cases. As part of the future work, we intend to experiment with other symbolic execution engines.

Our approach consists of translating ADEPT models into Java programs that can be symbolically executed by SPF. The translation takes into account the fact that our goal is to generate test cases. Our translation process generates programs that are different from (more abstract than) the prototypes that are generated by the ADEPT tool. In the context of this work, user inputs are of two types: (a) sequences of user actions that describe interactions with the user interface; for example, pressing buttons and (b) values entered when prompted by the system; for example, entering a target altitude. By treating such values as symbolic, and solving the constraints introduced by the different choice points of the model, symbolic execution (when applicable) is able to generate test cases that will drive the program through all its possible paths.

We use a component from an Autopilot system modeled in ADEPT as the running example through this paper. We use the example to present our preliminarily results and use the results to discuss the challenges and benefits of test case generation in the HAI domain. The remainder of the paper is organized as follows. Section II provides background on ADEPT models and symbolic execution. Section III describes our proposed approach that consists of translating ADEPT models to Java programs for test case generation, and applying symbolic execution to those programs. Section IV reviews related work, and finally Section V closes the paper with conclusions and plans for future work.

II. BACKGROUND

This section provides background on ADEPT models, symbolic execution, and Symbolic Pathfinder (SPF).

A. ADEPT model

The Automation Design and Evaluation Prototyping Toolset (ADEPT) enables the rapid development of an executable specification for automation behavior and user interaction. The term executable specification refers to the concept of a testable prototype whose purpose is to support development of a more accurate and complete requirements specification. The specification of the lateral system component of an autopilot requirement prototype is shown in Table I.

Variables in the ADEPT requirement prototype are explicitly defined. Some variables are internal to the system, whereas other variables can be observed or modified by the user through the user interface. The operations of the executable specification in an ADEPT model are defined by the designer in an unambiguous format. Each operation consists of three parts: (a) pre-conditions, (b) user actions and (c) update rules. A pre-condition is a set of constraints where the the corresponding operation is executed only if the constraints are satisfied by the current values of the ADEPT model variables. User actions reflect button presses or other actions through which the user can directly influence the execution of the system. Update rules represent changes in the ADEPT model variables that result from the execution of the corresponding operation.

The lateral system ADEPT table from the autopilot prototype model is shown in Table I. There are two main types of entries in the lateral system table: `Inputs` and `Outputs`. `Inputs` consist of pre-conditions and user actions. The `Outputs` section specifies the changes in the system state that result from the execution of operations.

The columns numbered (0 through 9) represent the different operations (one operation per column). For example, in a given state if the pre-conditions in a column are met and the user performs the action specified in the column, then the update rules in the `Outputs` part of the same column are applied to the state. Each gray row in the `Inputs` section corresponds to a variable or user action. The pre-conditions are essentially a conjunction of disjunctions. In a pre-condition (variable or action shown in a gray row), at least one of the *bullet*-ed conditions (•) in the white rows below it must hold. Note that continuous variables are not supported and must therefore be discretized during the modeling phase.

Several variables are shown in the lateral system table example in Table I. Consider the variables in the `Inputs` section. The variable `simulationStatus` has two possible values `paused` or `running`. Another variable `selected lateral target error` has ranges defined for it. The range of the variable can be either $>$ 179, $<=$ 179 && $>=$ -179, or $<$ -179. In certain cases more than one value of a variable are marked with • in a column such as column 5. As mentioned in the previous paragraph, the semantics is that `lateral system table output state` can be equal to *either* `hold selected target lateral target` *or* `capture and maintain lateral flight plan`. If no value is specified for a variable for a particular operation (meaning that there is no corresponding *bullet*), the value of the variable is irrelevant (unconstrained) for the operation. In the `Outputs` consider the variable `preselected lateral target`, it is assigned the value of the `lateral direction` when operation in column 8 is executed. There are no update rules for the `lateral direction` variable in Table I because this table is only a very small part of a bigger autopilot example. There are other tables in the autopilot model that specify updates to certain variables shown in Table I.

TABLE I
LATERAL SYSTEM COMPONENT

	0	1	2	3	4	5	6	7	8	9
lateralSystemTable										
Inputs										
simulationStatus										
paused	•									
running		•	•	•						
lateral Interface Action OutputState										
noAction	•	•	•	•						
user presses Lateral Target knob					•	•	•	•		
user presses Lateral Hold button									•	
user presses Lateral flight plan button										•
lateral system table output state										
capture and maintain selected lateral target		•			•					
hold selected lateral target			•			•	•	•		
capture and maintain lateral flight plan				•		•	•	•		
selected lateral target error										
> 179							•			
$<= 179 \&\& >= -179$						•				
< -179								•		
Outputs										
lateral system table output state										
capture and maintain selected lateral target					•	•	•	•		
hold selected lateral target									•	
capture and maintain lateral flight plan										•
selected lateral target error										
$-= 360$							•			
$+= 360$								•		
0									•	
preselected lateral target										
lateral direction									•	
selected lateral Target										
preselected lateral target					•	•	•	•		
lateral direction									•	
lateral target										
selected lateral target		•			•	•	•	•		
lateral direction			•						•	
lateral flight plan target				•						•
lateral target error										
selected lateral target error						•	•	•		
lateral flight plan target error										•
0									•	

B. Symbolic Execution

Symbolic execution is a program analysis technique for systematically exploring a large number of program execution paths [7], [13]. It uses symbolic values in place of concrete (actual) values as program inputs. The resulting output values are computed as expressions defined over constants and

symbolic input values, using a specified set of operators.

A symbolic execution tree characterizes all execution paths explored during symbolic execution. Each node in the tree represents a symbolic program state, and each edge represents a transition between two states. A symbolic program state contains a unique program location identifier (`Loc`), symbolic expressions for the symbolic input variables, and a path condition (`PC`). During symbolic execution, the path condition is used to collect constraints on the program expressions, and describes the current path through the symbolic execution tree. Path conditions are checked for satisfiability during symbolic execution; when a path condition is infeasible, symbolic execution stops exploration of that path and backtracks. In programs with loops and recursion, infinitely long execution paths may be generated. In order to guarantee termination of the execution in such cases, a user-specified depth bound is provided as input to symbolic execution.

We illustrate symbolic execution with the following example that is codified from the specification in Table I:

```
if(selectedLateralTargetError>179)
 selectedLateralTargetError-= 360
else if(selectedLateralTargetError<-179)
 selectedLateralTargetError+= 360
```

The path condition is set to `true` at the start of execution. When the line `selectedLateralTargetError` > 179 is executed and evaluates to `TRUE`, the value of `selectedLateralTargetError` is set to the expression `selectedLateralTargetError` $- 360$. During the execution of line `selectedLateralTargetError` < -179, the expression `selectedLateralTargetError` $+ 360$ is computed and stored as the value of `selectedLateralTargetError`. A *symbolic summary* for the piece of code above consists of the two path conditions that represent feasible execution paths for the code. The two path conditions generated are as follows:

- `selectedLateralTargetError` > 179
- `selectedLateralTargetError` < -179

The path conditions are *solved* to generate a set of concrete values (test inputs) which when can be used concrete test cases.

In this work we use Symbolic PathFinder (SPF) [17], [15], a symbolic execution extension to the Java PathFinder model checker–a Java bytecode analysis framework [23]. SPF is an open source execution engine that symbolically executes Java bytecode. SPF supports a variety of constraint solvers/decision procedures for solving path conditions; in this work we use the Choco constraint solver [6]. In general, state matching is undecidable when states represent path conditions on unbounded input data. Hence, SPF does not perform any state matching and explores the symbolic execution tree using a stateless search. Furthermore, if the solver is unable to determine the satisfiability of the path condition within a certain time bound, SPF treats the path condition as unsatisfiable. While this situation does not occur for any of the artifacts in our study, this limitation of constraint solvers could in general affect this approach, causing it to miss generating affected path conditions in the modified program. Loops and recursion can be bounded by placing a limit on the search depth in SPF or by limiting the number of constraints encoded for any given path; SPF indicates when one of these bounds has been reached during symbolic execution. There are no loops or recursive calls in the artifacts used in our empirical study, hence, we do not specify a depth bound.

III. Approach

In this section we first describe the translation of ADEPT models into Java programs. The Java programs are then symbolically executed by SPF. The translation process is designed with test-case generation in mind. Specifically, we encode the user input to the interface as symbolic variables. During the symbolic execution process, solving the constraints allows us to generate test cases that drive the model through all possible configuration sequences.

A. Translation into a Java program

We generate a Java program, as an intermediate representation, from an ADEPT model in order to automatically generate test cases. Translating the model into a Java program allows us to use the Java Pathfinder toolkit to automatically analyze the model and generate test cases. A portion of the Java program generated for the component shown in Table I is shown in Fig. 1. This translation is currently performed in a manual but systematic way. The plan is to automate this step in the future; this is feasible because the ADEPT models have well-defined unambiguous syntax and semantics. Two operations shown in columns labeled 6 and 7 in Table I are shown in Fig. 1.

A class is created for the lateral system in Table I. The fields of the class are variables in the ADEPT model. Those variables that are used in preconditions and determine the execution of update rules are declared as symbolic using the `@Symbolic` annotation (lines 3 to 7 in Fig. 1). This annotation is used by SPF to treat the fields in a class as symbolic. The *outputState* variable represents the `lateral system table output state` in Table I. In order to facilitate the symbolic execution process, it is defined as an integer that can have three values: zero, one, and two. The variables that are not used as preconditions in Table I are declared concretely with the initial values that are specified in the ADEPT model. The variables with concrete values are shown on lines 9 to 11 in Fig. 1.

The `execute` method in Fig. 1 represents the main method that is invoked from a driver. This method is invoked n times, where n is specified by the user. An example driver is shown below:

```
for(int i = 0; i < n; i++)
   execute(new symVar(), new symVar(),
                          new symVar())
```

Invoking the `execute` method allows us to generate test sequences that are n long. Each time the `execute` method is invoked fresh symbolic variables (`new symVar()`) are

```
1: public class lateralSystemTable{
2:
3: @Symbolic("true")
4: boolean isNominal;
5:
6: @Symbolic("true")
7: int outputState;
8:
9: int preSelectedLateralTarget = 180;
10: int lateralDirection = 180;
11: int selectedLateralTarget = 180;
12: ...
13:
14: public void execute(
15:     boolean userPressesLateralTargetKnob
16:     boolean userPressesLateralHoldButton
17:     boolean userPressesLateralFlightPlanButton){
18: ...
19:  if(isNominal == false &&
20:    ((outputState == 1) || (outputState == 2)) &&
21:    selectedLateralTargetError > 179 &&
22:    (userPressesLateralTargetButton == true &&
23:    userPressesLateralHoldButton == false &&
24:    userPressesLateralFlightPlanbutton == false)){
25:      applyRule06()
26: }
27: if(isNominal == false &&
28:    ((outputState == 1) || (outputState == 2)) &&
29:    selectedLateralTargetError < −179 &&
30:    (userPressesLateralTargetButton == true &&
31:    userPressesLateralHoldButton == false &&
32:    userPressesLateralFlightPlanbutton == false)){
33:      applyRule07()
34:  }...
35: }
36:
37: public void applyRule06(){
38:    outputState = 0;
39:    selectedLateralTargetError− = 360;
40:    selectedLateralTarget = preSelectedLateralTarget;
41:    lateralTarget = selectedLateralTarget;
42:    lateralTargetError = selectedLateralTargetError;
43: }
44:
45: public void applyRule07(){
46:    outputState = 0;
47:    selectedLateralTargetError+ = 360;
48:    selectedLateralTarget = preSelectedLateralTarget;
49:    lateralTarget = selectedLateralTarget;
50:    lateralTargetError = selectedLateralTargetError;
51: }
```

Fig. 1. Part of the lateral system table translated into a Java program as an intermediate representation that enables test case generation.

generated. The reason for this is described in Section. III-B. For each of the operations in Table I, there is an `if` conditional branch statement. This checks whether the variables satisfy the conditions that enable the execution of the corresponding update rule. Each update rule is added as a separate method. The methods `applyRule06` and `applyRule07` on lines 37 and 45 respectively represent the update rules for columns 6 and 7 in Table I.

Semantics of user actions One of the technical challenges of this work was to encode the semantics of a particular user action in a Java program. Consider these two options (a) a user presses a momentary switch (button) that is only active while the user is pushing it and then returns to an unpressed state or (b) the user could push a toggle switch that continues to stay in its on or off state. The modeling of pressing switches or buttons can get fairly complex when the cognitive aspects of the human are added to the semantics of the button press. In this work, however, we restrict the semantics of the button push to a momentary switch. In future work we plan to add more sophisticated semantics of user actions in our model.

$$true \tag{1}$$

$$\begin{aligned} isNominal == false \\ outputState == CONST\_2 \wedge outputState \neq CONST\_1 \wedge \\ selectedLateralTargetError > CONST\_179 \wedge \\ userPressesLateralTargetButton(s_1) == true \wedge \\ userPressesLateralHoldButton(s_2) == false \wedge \\ userPressesLateralFlightPlanbutton(s_3) == false \end{aligned} \tag{2}$$

$$\begin{aligned} outputState == CONST\_2 \wedge outputState \neq CONST\_1 \wedge \\ selectedLateralTargetError - 360 > CONST\_179 \wedge \\ userPressesLateralTargetButton(s_4) == true \wedge \\ userPressesLateralHoldButton(s_5) == false \wedge \\ userPressesLateralFlightPlanbutton(s_6) == false \wedge \end{aligned} \tag{3}$$

Fig. 2. A path condition that represent a path through the symbolic execution tree for the lateral system program in Fig. 1

The `lateral Interface action OutputState` variable represents different user actions. The user can perform one of four actions: `noAction`, `user presses lateral target knob`, `user presses lateral Hold button`, and `user presses lateral flight plan button`. The last three actions are encoded as boolean variables in the `execute` method in Fig. 1. Each time the `execute` method is invoked the variables are re-initialized as symbolic variables. Note that we do not explicitly represent `noAction`. When all the other user actions are false it implies `noAction` is taken. This is simply an optimization.

B. Test Case Generation

The `execute` method is symbolically executed n number of times to generate sequences of user actions and variable inputs. Each time the `execute` is invoked it generates fresh symbolic values for the user actions encoded as boolean variables. This allows us to model the semantics of a momentary switch as described previously. Recall that path conditions are generated during the symbolic execution of a program. An example of a path condition along a certain program path during symbolic execution is shown in Fig. 2.

When symbolic execution begins, the path condition at the start is set to $true$ represented in Eq. (1). When the symbolic

execution encounters the `if` condition at lines $19-24$ in Fig. 1, it adds the constraint (Eq. (2)) to the path condition shown in Fig. 2. There is a fairly straightforward mapping between the conditions in the `if` statement starting at line 19 in Fig. 1 to constraints in the path condition in Eq. (2) within Fig. 2. The conjunction of Eq. (1) and Eq. (2) in the path condition is satisfiable. There exists a set of assignments that can be made to Eq. (2) that will make the path condition feasible. An example of a satisfying assignment for the variables is the following:

- `isNominal` = *false*
- `outputState` = 1
- `selectedLateralTargeterror` = 180
- `userPressesLateralTargetButton` = *true*
- `userPressesLateralHoldButton` = *false*
- `userPressesLateralFlightPlanButton` = *false*

A constraint solver can generate satisfying assignments for constraints such as the one shown above. The concrete values generated by the constraint solver can be assigned to the variables in the system in order to execute the operation encoded in `applyRule06` shown in Fig. 1 The values of certain variables are updated during the execution of `applyRule06`. The updated variables are shown below:

- `outputState` = 1
- `selectedLateralTargetError` = `selectedLateralTargetError - 360`

Note that `selectedLateralTargetError` is updated symbolically so 360 is subtracted from its symbolic value during symbolic execution.

When the `execute` method is invoked where $i = 2$ in the driver, the constraint shown in Eq. (3) of Fig. 2 is added again at lines 19 to 24. Now, however, the value of the `outputState` variable is one assigned during the execution of `applyRule06`. The constraint $outputState \neq CONST\_1 \wedge outputState == CONST\_2$ is not satisfiable, hence, the path condition is also not satisfiable. There exists no assignment to the symbolic variables that can satisfy the path condition. Note that the boolean variables that represent the user actions in Eq. (3) have a different signature (s_4, s_5, and s_6) compared to the user action variables in Eq. (2) (s_1, s_2, and s_3).

The example in Fig. 2 shows that after the execution of `applyRule06`, it cannot be executed again. Similarly, the conditions to execute `applyRule07` cannot be satisfied after `applyRule06`. The only two sequences that can be generated from the program in Fig. 1 are: (a) execute rule 6 and (b) execute rule 7. The output of the symbolic execution also provides the details of the values of the initial system, and the user actions required to perform the update as described earlier.

C. Coverage Results

The entire Java program for the lateral system in Table I is executed symbolically using SPF. We present results for when n is set to one. The goal is to generate test cases that allow us to obtain full coverage of the model, such that each rule gets executed at least once. SPF generates 2313 states in 4 seconds to symbolically execute all the rules (total 10) in the lateral system Java program. In this it achieves a 100% statement, branch, method, and basic block coverage. Coverage results for the lateral system table:

Metric	Number	Coverage
Bytecode	432	100%
Line	99	100%
basic-block	123	100%
branch	58	100%
method	14	100%

It generates 16 test cases that have values which when executed concretely along with specified user actions all rules in the Java program are executed. These values and user actions can be used to test the system represented by the ADEPT model.

IV. Related Work

Human automation interaction (HAI) is an interdisciplinary domain that has been studied for several years by researchers and spans cognitive science, systems engineering, computer science, and human factors. The complexity of designing and verifying such systems has also driven researchers to investigate the use of formal specification and verification methods in their development. Early results focused on specific applications [18], [5] but later on using theories like graph theory, model-checking or theorem proving [20], [4], [8] were also investigated.

Formal methods rely on a formal model for the HAI system under study (recently formal methods can also directly handle programming languages such as Java or C). Several approaches have been proposed. For example Campos et al. [4], [5] developed a framework where the system is modeled in terms of interactors, and desired properties in the MAL logic. Some generic usability properties are defined [5] and model checking is used for verification. Thimbleby et al. [20], [19] use graphs to represent models and define usability properties through graph structural properties. Curzon et al. [8] define systems and properties with modal logic and check them with theorem proving. Finally, Bolton et al. [3], [2] use model checking to analyze application-specific HAI properties; their work is based on translation of user task models expressed in the EOFM environment into the input language of the SAL model checker.

The ADEPT toolset supports tabular specifications of an HAI system and can automatically check generic properties such as determinism and completeness [9]. The focus so far has been on easy specification and fast prototyping. In this work, we investigate the possibility of generating test cases based on ADEPT models.

Model-based test-case generation has been studied extensively in the literature. In a black-box setting where the source code of a program under test is not available, model checking and symbolic execution, for example, can be used to extract large numbers of test cases from high level descriptions of

inputs to the system, or from tests described in a parameterized fashion [12], [21]. In a white-box setting, symbolic execution has been studied extensively for generating inputs that will drive the program towards achieving a desired degree of coverage [16], [14]. Test-case generation for GUI applications has also been studied using runtime rather than formal techniques [24]. Finally, previous work has investigated the possibility of model checking the Java autopilot prototype generated by ADEPT using JPF [22]. In the context of aerospace applications, several test case generation techniques based on formal methods have been investigated [11].

V. Conclusion and Future Work

This paper describes our early work with generating test cases for HAI systems based on models created in the ADEPT tool set. Our work consists of automatically translating ADEPT models into Java programs that can be analyzed by the symbolic engine of Java Pathfinder, SPF, to generate user inputs. The generated tests can be targeted to achieve desired coverage, and can significantly facilitate the testing process for HAI systems though automation of both the test generation and the test execution process. The generated tests could be executed on the prototype generated by ADEPT in order to obtain test oracles for the actual system implementation.

Our approach was demonstrated on a component of an Autopilot specification. Our initial results indicate that the approach is promising. However, our framework is still preliminary. Some features of ADEPT are not yet supported, and we need to experiment with alternative approaches to the problem in order to decide what would be more efficient. For example, not only would we like to experiment with different ways of translating the ADEPT models, but also with different test case generation engines.

Symbolic execution itself is an expensive approach that may not scale to the size of the systems targeted, and has limitations in the presence of loops and non-linear constraints. Even in our small example, the number of combinations that are feasible by enabling all the possible rules to be applied to the system creates a very large number of potential sequences. However, the capability that it provides to generate automatically high quality test suites justifies its use for safety critical components of our targeted applications.

As part of our future work, we will investigate ways of increasing the scalability of symbolic execution through compositional techniques [1]. Moreover, we wish to investigate meaningful coverage criteria for the rule-based specifications of ADEPT in order to potentially reduce the number of possible cases generated without losing quality of the resulting test suites. Finally, we will work on customizing symbolic execution techniques for HAI applications - where the focus is on generating *sequences* of human inputs that take into account the semantics of user interaction mechanisms.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.

[2] M. L. Bolton and E. J. Bass. Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs. *ISSE*, 6(3):219–231, 2010.

[3] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu. Using formal methods to predict human error and system failures. In *Proceedings of the 2nd Applied Human Factors and Ergonomics International Conference*, pages 14–17, July 2008.

[4] J. C. Campos and M. D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3–4):275–310, 2001.

[5] J. C. Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In *Proceedings of the 15th International Workshop on the Design, Verification and Specification of Interactive Systems*, number 5136 in Lecture Notes in Computer Science, pages 72–85. Springer-Verlag, July 2008.

[6] Choco. Main–page Choco. http://www.emn.fr/z-info/choco-solver/, 2010.

[7] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[8] P. Curzon, R. Rukšėnas, and A. Blandford. An approach to formal verification of human-computer interaction. *Formal Aspects of Computing*, 19(4):513–550, Nov. 2007.

[9] M. S. Feary. A toolset for supporting iterative human – automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, Mar. 2010.

[10] M. S. Feary. A toolset for supporting iterative human automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, 2010.

[11] D. Giannakopoulou, D. Bushnell, J. Schumann, H. Erzberger, and K. Heere. Formal testing for separation assurance. *Annals of Mathematics and Artificial Intelligence*, pages 1–26, 2011. 10.1007/s10472-011-9224-3.

[12] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *ICSE*, pages 225–234, 2010.

[13] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[14] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *ICSM*, pages 1–10, 2010.

[15] C. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.

[16] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *STTT*, 11(4):339–353, 2009.

[17] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–25, 2008.

[18] J. Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, Feb. 2002.

[19] H. Thimbleby. *Press On: Principles of Interaction Programming*. The MIT Press, Nov. 2007.

[20] H. Thimbleby and J. Gow. Applying graph theory to interaction design. In J. Gulliksen, editor, *Engineering Interactive Systems 2007/DSVIS 2007*, number 4940 in Lecture Notes in Computer Science, pages 501–518. Springer-Verlag, 2008.

[21] S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In *FASE*, pages 294–309, 2011.

[22] O. Tkachuk, G. Brat, and W. Visser. Using code level model checking to discover automation surprises. In *Digital Avionics Systems Conference (DASC)*, 2002.

[23] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[24] X. Yuan and A. M. Memon. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Trans. Software Eng.*, 36(1):81–95, 2010.