Task-based Automated Test Case Generation for Human-machine Interaction

Meng Li
University at Buffalo, SUNY

Matthew L. Bolton
University at Buffalo, SUNY

Testing is an effective approach for finding discrepancies between intended and actual system behavior. However, the complexity of modern system can make it difficult for analysts to anticipate all the interactions that need to be tested. This is particularly true for human-interactive systems where humans may do things that were not anticipated by analysts. We address this by introducing a novel approach to automated test case generation for human-machine interaction. We do this by combining formal models of human-machine interfaces with formal models of human task behavior. We then use the robust search capabilities of model checking to generate test sequences guaranteed to satisfy test coverage criteria. We demonstrate the capabilities of our approach with of a pod-based coffee machine. Results and future research are discussed.

INTRODUCTION

Testing is an effective approach for finding discrepancies between intended and actual system behavior and unanticipated problems. Testing, including the design and execution of test cases, is often regarded as the most intellectually-demanding, time-consuming, and expensive parts of system development (Bertolino, 2007). As such, it can be difficult (if not impossible) for testers to anticipate all of the system conditions that need to be evaluated. This is especially true of human-machine systems. This is because the human operator is an additional concurrent component to the system and one whose behavior is not governed by the system's implementation.

To address these issues, researchers have developed automated test case generation (Ammann & Offutt, 2016). Many test case generators use formal methods: mathematically based languages, techniques, and tools for the modeling, specification, and analysis of systems. These are model-based approaches for creating tests that are efficient and provide guarantees about their completeness (with respect to the model). In particular, model checking can be used to automatically generate test cases. It does this by parsing the model of the system with efficient and exhaustive searches to create test cases (traces through the model) that satisfy specified coverage criteria: descriptions of the system conditions the test must encounter.

In this paper, we introduce a new method for automatically generating test cases for human-machine systems using human task behavior and system models. With a given formal coverage criterion (based on the interface and task models), model-checking-based-tools are used to automatically create tests from the models that satisfy task and interface coverage criteria.

Below we describe the literature relevant to understanding our method. This is followed by a description of our objectives. We then present our method and discuss its implementation. Then, we apply our method to the evaluation of a coffee machine by generating tests for it using two coverage criteria and executing them on an actual system. We report these results and discuss their implications for future research.

BACKGROUND

Our approach uses formal task analytic behavior models and model-checking-based tools for automated test case generation. We discuss these and the use of test case generation in human-machine systems below.

Task-analytic Models of Human Task Behavior

Task analytic behavior models (or task models) are produced by a task analysis (Kirwan & Ainsworth, 1992) to capture the behaviors human operators use to achieve goals when interacting with a system. Task models describe how people actually interact with an existing system or, if used during design, how designers expect people to interact with the system. They can be used in system engineering in a number of different capacities including human-automation interface creation, training development, and usability analyses.

Task models represent the input-output behavior of human operators by explicitly describing the environmental conditions under which humans pursue goals and how they achieve those goals by performing actions. Thus, they can be interpreted computationally. This allows them to be included in formal methods analyses (Bolton, Bass, & Siminiceanu, 2013) like model checking and automated test case generation.

Formal Methods and Model Checking

Formal methods are rigorous tools and techniques for the modeling, specifying, and verifying systems. Formal modeling constructs system models using well-defined mathematical languages. Specifications are desired system properties. Formal verification mathematically proves whether the specification properties are always satisfied by the model.

Model checking (Clarke, Grumberg, & Peled, 1999) is an automated approach to formal verification. The formal model is usually represented as a state machine: states and transitions between states. Specification properties are usually logically asserted using temporal logic. Model checking performs formal verification by exhaustively searching the models statespace to determine whether the model satisfies the specification. If there is no violation, the model checker has proven that the specification is true in the model. Otherwise, the model checking returns a counterexample: a counterproof that shows steps through the system model that violate the specification.

Model checking is widely used in computer software and hardware engineering. They have also received attention from the human-machine interaction community that seeks to apply them in the engineering of reliable human-machine systems (Bolton, 2017a; Bolton et al., 2013). In particular, task behavior models can be included in larger formal verification analyses of complex systems (Bolton et al., 2013) and formal-methods-based automated test case generation.

Automated Test Case Generation

A test case is a sequence of inputs and expected outputs that can be run on a system (Ammann & Offutt, 2016). Automated test case generation automatically creates test cases from information (design documents or implementations) available about the target system. The generic, model-based test case generation process has three steps: (1) constructing a model of the system, (2) developing coverage criteria to reflect the testing requirements, and (3) generating the test cases.

Coverage criteria can vary based on analyst goals. When testing software, an analyst may wish to execute every function in the source code (functional coverage), every line of code (statement coverage), or every branching point in the program (condition coverage) (Ammann & Offutt, 2016). For state machine models (like those used in model checking), a tester may want to ensure that tests hit every system state (state coverage) or every transition (edge coverage) in the system model.

Because of its propensity for traversing system models and producing traces, model checking can be used for automated test generation. There are two general approaches to this (Fraser, Wotawa, & Ammann, 2009). In the first, test cases are created by checking specification properties expected to produce counterexamples. By generating multiple counterexamples with multiple specifications, analysts can ultimately satisfy coverage criteria. However, this approach can result in tests that are inefficient. Thus, other approaches allow analysts to explicitly describe coverage criteria and use model checking to find tests that satisfy them. For example, the Symbolic Analysis Laboratory (SAL) (Hamon, de Moura, & Rushby, 2004) allows analysts to identify a number of Boolean "trap" variables in the model that describe the coverage criteria. These variables become true when corresponding parts of the coverage criteria are satisfied. The model checker then uses efficient search algorithms to drive all of the trap variables to true.

Test generation has two purposes. Analysts may use tests to validate that an implementation matches a formal model's behavior and thus any properties verified against it. Analysts also use tests to collect measures from the actual system that were not possible to represent in the formal model.

Generating Test Cases for Human-machine Interaction

A small but growing body of literature has been exploring how formal-methods-based test case generation can be used for human-machine systems. There are generally two approaches: those that are interface-based and those that are task-based.

Interface-based methods use the human-machine interface as the system model (Bowen & Reeves, 2009; d'Ausbourg, 1998; Memon, Pollack, & Lou Soffa, 2001). In such methods, coverage criteria is asserted over a formal model of the interface. For example interface state coverage specifies that every state of the interface is visited in generated tests.

Task-based methods treat the task model as the system model and generate tests from it. In these, actions that conform with the task model are deemed valid and contextualized by system conditions prescribed in the task (Barbosa, Paiva, & Campos, 2011; Campos, Fayollas, Martinie, & Navarre, 2016; Vieira, Leduc, Hasling, Subramanyan, & Kazmeier, 2006). In these approaches, coverage criteria are asserted over elements of the task. For example, activity coverage could be used to generate tests that would have the user (or an automated sequence) execute every activity included in a task.

All of these approaches have been used successfully in the testing of human-machine systems. However, they have limitations. Interface-based approaches include sequences of actions that are random and are thus inefficient divorced from what goals the human is pursuing and how they can try to achieve them. Task-based approaches do not use an actual description of the machine's and its interface's behavior. Thus, tests may not capture realistic interactions that are dictated by the unrepresented parts of the system's automation. Given the missing elements in each approach, neither can express coverage criteria that genuinely represents human-automation interaction.

OBJECTIVE

In this research, we introduce a new approach to automated test case generation. This avoids the limitations of the previous methods by allowing tests to be generated that are contextualized in terms of both human task behavior and system behavior. Thus, coverage criteria can be expressed over both task and interface models. Our method uses task models represented in EOFM and employs its supported formal system modeling architecture. Thus, we discuss EOFM and its modeling architecture below. We then present our method. This is followed by an application of our approach to the testing of a pod-based coffee machine. We report on problems we discovered executing our tests on the coffee machine.

METHODS

The Enhanced Operator Function Model

We make use of EOFM (Bolton, Siminiceanu, & Bass, 2011). This XML-based language models human task behavior as an input/output system. Inputs come from other parts of the system like the interface or environment. Outputs are human actions. The task model describes how human actions are generated based on input and local variables.

EOFMs are represented as a hierarchy of goal-directed activities and actions. Activity strategic knowledge is modeled explicitly as Boolean conditions using input and local variables. These can assert what must be true for an activity to start executing (*Precondition*s), repeat (*RepeatCondition*s), or complete (*CompletionCondition*s). Activities decompose into sub-activities and ultimately atomic actions. A decomposition operator describe the temporal relationships between, and the cardinality of, the decomposed activities or actions. EOFM has nine such operators (Bolton et al., 2011). In this work, we use two: `xor`, where exactly one activity or action in the decomposition executes, and `ord`, where all activities or actions must execute in the order they appear in the decomposition.

EOFMs can be visually rendered as tree-like graphs (see Figure 3). They also have formal semantics (Bolton et al., 2011; Bolton, Zheng, Molinaro, Houser, & Li, 2017) that mathematically describe how they can execute. Every activity and action is treated as a state machine that transitions between three states: *Ready* (the initial state), *Executing*, and *Done*. The strategic knowledge conditions of an activity and implicit conditions based on the activity's or action's location in the task determine whether it can start, end, or reset based on its position in the task (Bolton et al., 2011, 2017).

EOFM has a Java-based translator that will convert an EOFM into the input language of SAL (De Moura, Owre, & Shankar, 2003) using these formal semantics. This allows EOFM to be used as part of a larger system model in model checking analyses. This larger system model is constructed around EOFM's formal modeling architecture (Bolton & Bass, 2010) which can (as needed) account for human operator mission goals, machine automation behavior, human-machine interfaces, and environmental dynamism.

Our Method for Automated Test Case Generation

Our approach to automated test case generation for human-machine interaction is shown in Figure 1. In this, it is assumed that an analyst has access to a description of the system they want to evaluate and a normative task model (from a task analysis). Through automated generation and manual operations, these artifacts are converted into both a formal system model (encompassing the EOFM's architectural concepts; Bolton and Bass 2010) and coverage criteria (which can be expressed over both the task model and the other elements of the architecture).

With a formal system model and coverage criteria, model-checking-based automated test case generation is used to create tests that satisfy the coverage criteria. These show how human operators perform tasks (including atomic actions) as well as the response these behaviors produce in the other elements of the system. These can be executed on an actual implementation of the system to validate that the system conforms to the model.

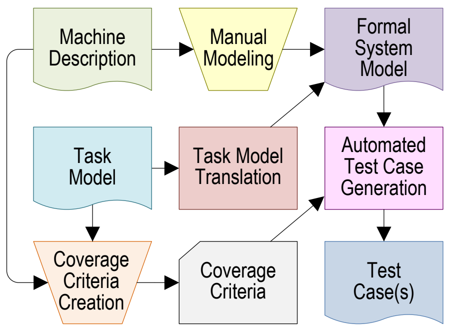

Figure 1. Flow diagram of our test case generation method.

Method Implementation

We implemented our method using EOFM for modeling human operator tasks. This allowed us to use EOFM's translator for incorporating task behavior in a formal system model. This also allowed us to use EOFM's support for automatically creating (Li, Wei, Zheng, & Bolton, 2017) the other formal elements of the system. To create tests, we used the automated test case generator of SAL (Hamon et al., 2004). This was employed because it is the formal modeling and analysis suite supported by EOFM. In SAL, an analyst identifies coverage criteria using Boolean trap variables. This is currently done manually by the analyst in our implementation. The SAL test case generator uses its model-checking-based search algorithms to find a trace (test sequences) through the model that causes all of the trap variables to become true.

APPLICATION

To demonstrate the capabilities of our method, we apply it to a realistic application: a pod-based coffee machine (Figure 2). For the purpose of this analyses, we used information from the coffee machine's manual to create an EOFM to represent the normative human task behavior for interacting with the device. In the task (Figure 3), the human can turn the device on (a) or off (b) by pressing the power button (`hPressPowerButton`). The lid can be opened if it is closed (c) and closed if it is open (d). If the lid is open and the water indictor shows that not enough water has been entered, the user can add water to the reservoir (e). The user can lift the handle (f) if the handle is down and the machine is not brewing. If the handle is up, the user can enter a pod (g) if one is not present or remove a pod (h) if one is. If the handle is up, it can be lowered (i). If the machine is brewing, the user can pause brewing by pressing the brew button (j) or wait for brewing to finish (k). A mug can be placed on the platform (m) if no mug is there. Brewing can be started (l) if the machine's power is on, the handle is down, there is enough water in the reservoir, a mug is placed on the platform, and the machine's brewing is paused or stopped. Finally, a user can remove a mug (n), where removing the mug during brewing will result in the machine pausing.

Using this task as input, we created a functional design of the system's interface using the generation method described by Li et al. (2017). This produced the interface model shown in Figure 4. In this, the interface is represented by seven, concurrent, synchronously composed state machines: the brewing state of the machine (`iBrewingState`; a); the state of the pod (`iPodEntered`; b); the water indicator (`iEnoughWater`; c); the lid (`iLid`; d); the handle (iHandle; e); the power (`iPower`; f); and the placed state of the mug (`iMug`; g).

We then automatically converted the task model into the input language of SAL using EOFM's translator (Bolton & Bass, 2010) to form a larger formal system model. In doing this, we automatically generated specification properties (using the process from Bolton, Jimenez, van Paassen, and Trujillo (2014)) to check that the interface will always support the human operator's task. These were verified with the formal system model using SAL's symbolic model checker.

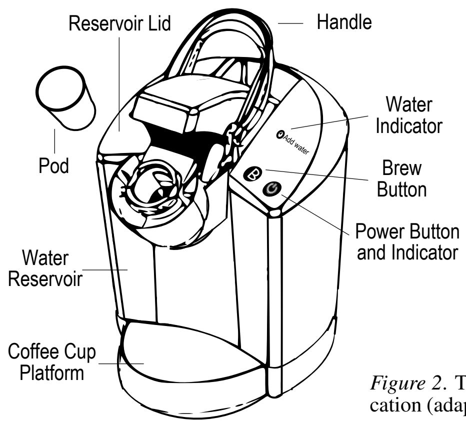

Figure 2. The coffee machine application (adapted from Bolton 2017b).

In this application, we wanted to ensure that we generated tests that could cover both the interface states of the machine and the activities of the tasks. Thus we applied our method with both interface state coverage and task activity coverage. For interface coverage, we created trap variables representing each of the 84 reachable interface states from Figure 4. For activity coverage, we created trap variables that would become true whenever each activity or action in a task model was executing.

RESULTS

SAL's automated test case generator (`sal-atg`) was applied to our coffee machine model with the two coverage criteria using a Linux workstation with a 3.7 Ghz Xeon Process and 128 gigabits of RAM. Parameters were assigned so that an unlimited search depth was used (`-smcinit`), alternate branches could be pursued (`-branch`), and a search depth of 30 on any given model branch (`-ed 30`).

For interface coverage, test generation took 91.19 seconds and produced a test with 162 sequential actions. For activity coverage, generation took 9.55 seconds and produced a test with 34 sequential actions. We executed each test on an actual coffee machine while noting any discrepancies between system behavior and that predicted in the test. Discrepancies constituted validation failures for the implementation. We also noted incidental usability issues related to the undoability of actions related to the water in the system. Results are summarized in Table 1.

DISCUSSION

In this work, we introduced a novel method for the automated test case generation of human-machine interaction. This improves on preexisting methods that only accounted for system interface behavior or human task behavior. By including both in our analyses, we ensure that the test sequences generated are contextualized in terms of their interaction.

The coffee machine application illustrates the power of our approach. Specifically, we generated test cases that satisfy interface state and activity coverage from a formal model verified to always support the human operator's task. We then executed these tests on an implementation of the system to validate that the implementation conformed with the model and to gain incidental insights into system usability.

The tests identified four issues with the pod-based coffee machine. In the first, the physical design of the machine prevented the user from adding or remove a mug when the handle was lifted. The other three related to automation behavior: the machine could dispense coffee beyond the mug's capacity, the machine would not pause brewing when the mug was removed, and there was a situation where the machine would not begin

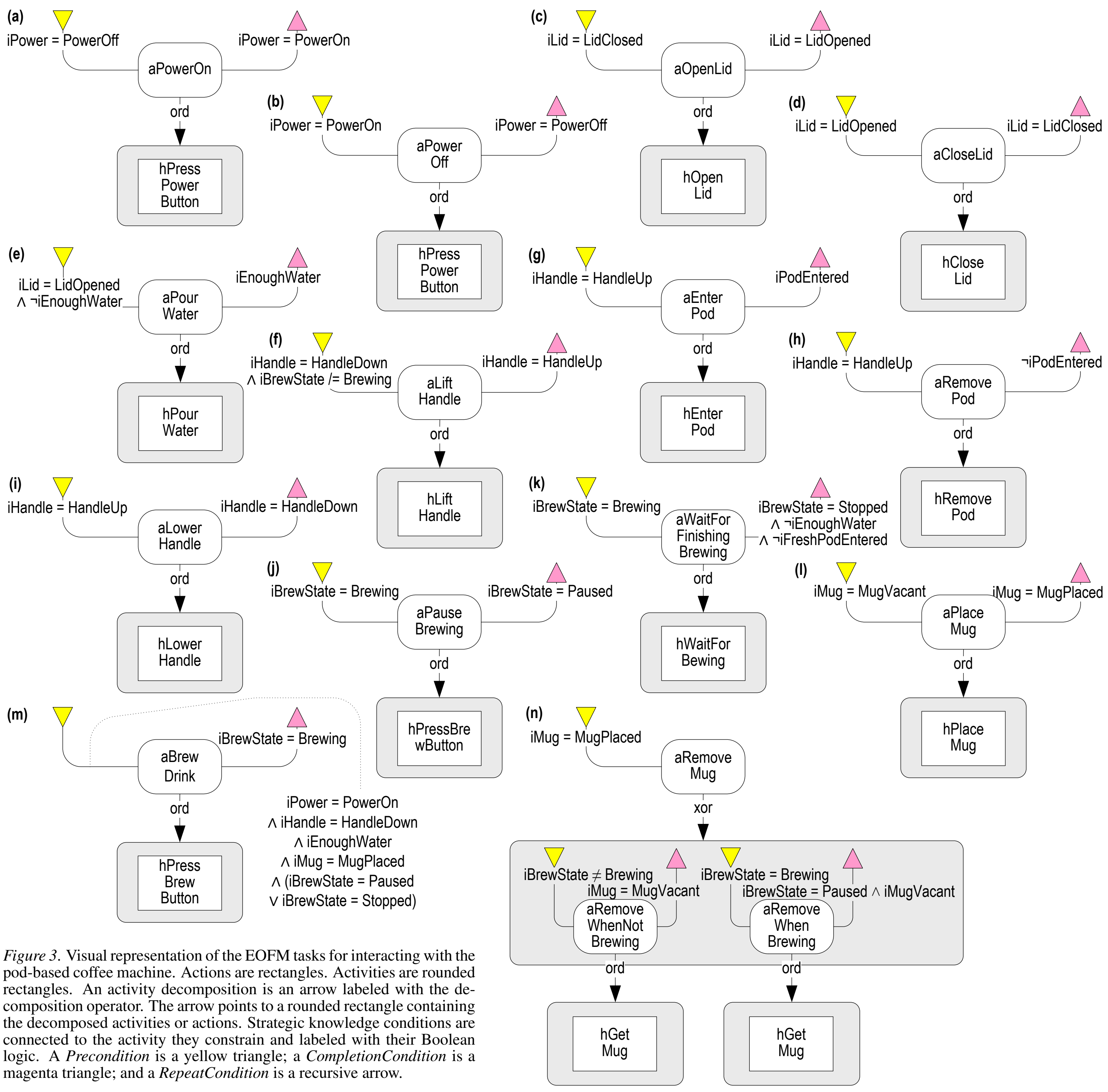

Figure 3. Visual representation of the EOFM tasks for interacting with the pod-based coffee machine. Actions are rectangles. Activities are rounded rectangles. An activity decomposition is an arrow labeled with the decomposition operator. The arrow points to a rounded rectangle containing the decomposed activities or actions. Strategic knowledge conditions are connected to the activity they constrain and labeled with their Boolean logic. A *Precondition* is a yellow triangle; a *CompletionCondition* is a magenta triangle; and a *RepeatCondition* is a recursive arrow.

brewing even though all of the necessary steps were complete. All of these constitute serious implementation issues. All but the last are relatively straightforward. The last one, where the machine failed to brew, appears to occur because the implementation of the machine uses the lifting of the handle to load water into an internal compartment to enable brewing. Thus, if there is not enough water in the machine when the handle is lifted, the machine will not brew. This is an interesting problem because, when the automation contains hidden states or modes, the human may not be able to keep track of the state of the machine, producing mode confusion (Sarter & Woods, 1995).

In discovering the hidden mode as well as the mug overflow, we came to the realization (an incidentally discovered usability issue) that there was a problem with undoability in the machine. If excessive water is loaded into the internal reservoir by lifting the handle, there is no direct way to undo this action short of brewing the coffee. Thus, manually executing the test cases had utility beyond merely identifying discrepancies between the model and its implementation.

While the presented method was useful in our application, there is room for improvement in future work. For example, the presented work only considered two different coverage criteria. Because EOFM's architecture can contain system elements beyond human tasks and interfaces (Bolton & Bass, 2010), coverage criteria could be asserted across additional system elements. Future work should investigate which coverage criteria provide analysts with the most useful information. Additionally, EOFM has a variant called EOFMC (EOFM with Communica-

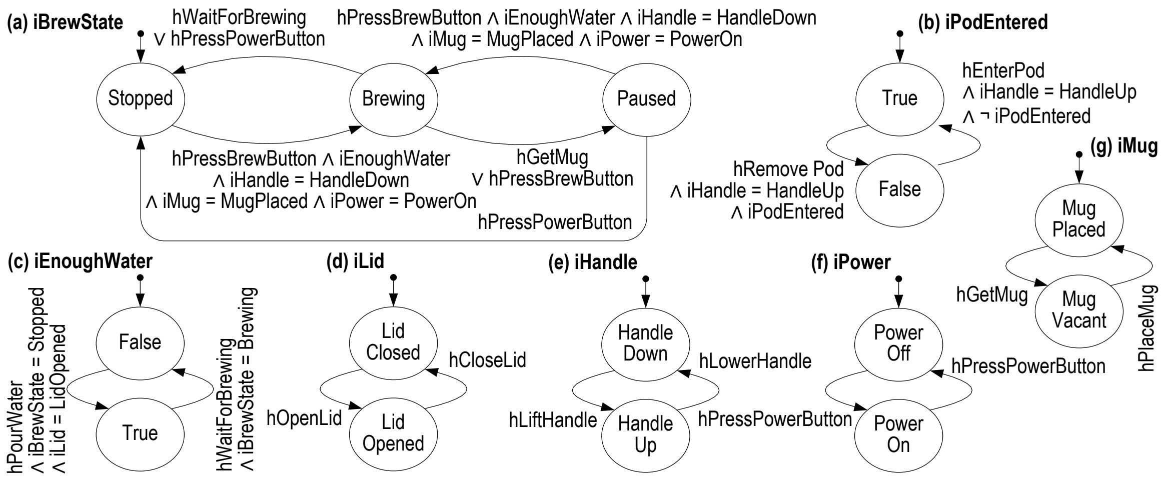
(a) iBrewState
hWaitForBrewing ∨ hPressPowerButton
hPressBrewButton ∧ iEnoughWater ∧ iHandle = HandleDown ∧ iMug = MugPlaced ∧ iPower = PowerOn
Stopped
Brewing
Paused
hPressBrewButton ∧ iEnoughWater ∧ iHandle = HandleDown ∧ iMug = MugPlaced ∧ iPower = PowerOn
hGetMug ∨ hPressBrewButton
hPressPowerButton
(b) iPodEntered
True
False
hEnterPod ∧ iHandle = HandleUp ∧ ¬ iPodEntered
hRemove Pod ∧ iHandle = HandleUp ∧ iPodEntered
(g) iMug
Mug Placed
Mug Vacant
hGetMug
hPlaceMug
(c) iEnoughWater
False
True
hPourWater ∧ iBrewState = Stopped ∧ iLid = LidOpened
hWaitForBrewing ∧ iBrewState = Brewing
(d) iLid
Lid Closed
Lid Opened
hCloseLid
hOpenLid
(e) iHandle
Handle Down
Handle Up
hLowerHandle
hLiftHandle
hPressPowerButton
(f) iPower
Power Off
Power On
hPressPowerButton

Figure 4. The human-machine interface for the pod-based coffee machine as a collection of seven synchronously composed, concurrent state machines. Note that ovals represent states and arrows between ovals represent transitions. Arrows starting with a dot indicate an initial state.

Table 1. *Steps Where Problems Occurred when Executing Test Cases Generated to Satisfy Different Coverage Criteria*

Interface Test	Activity Test	Problem
17, 207, 273, 397	133	Could not remove mug because the handle is up.
266, 906, 1156	124	Could not place mug because the handle is up.
471, 729	67	Previously adding water while the handle was up resulted in excess water being added and water overflowing the mug during brewing.
603, 1036	184	Removing the mug did not pause brewing.
1027	238	Water was added after the handle was raised, a pod was added, and the handle was lowered. This resulted in the brew button having no response.

tions) (Bass et al., 2011) that allows human-human communication and coordination to be incorporated into large task and formal system models both with and without miscommunication generation (Bolton, 2015). Future work should investigate how human-human communication and coordinated can be incorporated into our test generation method. Finally, Our test case generation method only considered normative human behavior. However, other test generation methods (Barbosa et al., 2011; Campos et al., 2016) have included generated human error in tests. This allows testers to assess the impact of erroneous human behavior on actual system performance. EOFM supports a number of different approaches for generating erroneous human behavior (Bolton, 2017b). Future work should investigate how the erroneous behavior generation methods supported by EOFM can be incorporated into test case generation.

REFERENCES

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Barbosa, A., Paiva, A. C., & Campos, J. C. (2011). Test case generation from mutated task models. In *Proceedings of the 3rd acm sigchi symposium on engineering interactive computing systems* (pp. 175–184).

Bass, E. J., Bolton, M. L., Feigh, K., Griffith, D., Gunter, E., Mansky, W., & Rushby, J. (2011). Toward a multi-method approach to formalizing human-automation interaction and human-human communications. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics* (pp. 1817–1824). Piscataway: IEEE.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *2007 future of software engineering* (pp. 85–103). IEEE Computer Society.

Bolton, M. L. (2015). Model checking human–human communication protocols using task models and miscommunication generation. *Journal of Aerospace Information Systems*.

Bolton, M. L. (2017a). Novel developments in formal methods for human factors engineering. In *Proceedings of the human factors and ergonomics society annual meeting* (Vol. 61, pp. 715–717). Los Angeles: Sage.

Bolton, M. L. (2017b). A task-based taxonomy of erroneous human behavior. *International Journal of Human-Computer Studies*, *108*, 105–121.

Bolton, M. L., & Bass, E. J. (2010). Formally verifying human-automation interaction as part of a system model: Limitations and tradeoffs. *Innovations in Systems and Software Engineering*, *6*(3), 219–231.

Bolton, M. L., Bass, E. J., & Siminiceanu, R. I. (2013). Using formal verification to evaluate human-automation interaction in safety critical systems, a review. *IEEE Transactions on Systems, Man and Cybernetics: Systems*, *43*(3), 488–503.

Bolton, M. L., Jimenez, N., van Paassen, M. M., & Trujillo, M. (2014). Automatically generating specification properties from task models for the formal verification of human-automation interaction. *IEEE Transactions on Human-Machine Systems*, *44*, 561–575.

Bolton, M. L., Siminiceanu, R. I., & Bass, E. J. (2011). A systematic approach to model checking human-automation interaction using task-analytic models. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, *41*(5), 961–976.

Bolton, M. L., Zheng, X., Molinaro, K., Houser, A., & Li, M. (2017). Improving the scalability of formal human-automation interaction verification analyses that use task-analytic models. *Innovations in Systems and Software Engineering*, *13*(1), 1–17.

Bowen, J., & Reeves, S. (2009). UI-Design Driven Model-Based Testing. *ECEASST*.

Campos, J. C., Fayollas, C., Martinie, C., & Navarre, D. (2016). Systematic automation of scenario-based testing of user interfaces. *EICS*.

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking*. Cambridge: MIT Press.

d'Ausbourg, B. (1998). Using model checking for the automatic validation of user interfaces systems. In *Design, specification and verification of interactive systems' 98* (pp. 242–260). Springer.

De Moura, L., Owre, S., & Shankar, N. (2003). *The SAL language manual* (Tech. Rep. No. CSL-01-01). Menlo Park: Computer Science Laboratory, SRI International.

Fraser, G., Wotawa, F., & Ammann, P. E. (2009). Testing with model checkers: A survey. *Software Testing, Verification and Reliability*, *19*(3), 215–261.

Hamon, G., de Moura, L., & Rushby, J. (2004). Generating efficient test sets with a model checker. In *Proceedings of the second international conference on software engineering and formal methods* (pp. 261–270). IEEE.

Kirwan, B., & Ainsworth, L. K. (1992). *A guide to task analysis*. London: Taylor and Francis.

Li, M., Wei, J., Zheng, X., & Bolton, M. L. (2017). A Formal Machine–Learning Approach to Generating Human–Machine Interfaces From Task Models. *IEEE Transactions on Human-Machine Systems*, 1–12.

Memon, A. M., Pollack, M. E., & Lou Soffa, M. (2001). Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Trans. Software Eng.*, *27*(2), 144–155.

Sarter, N. B., & Woods, D. D. (1995). How in the world did we ever get into that mode? Mode error and awareness in supervisory control. *Human Factors*, *37*(1), 5–19.

Vieira, M., Leduc, J., Hasling, W. M., Subramanyan, R., & Kazmeier, J. (2006). Automation of GUI Testing Using a Model-driven Approach. *AST*, 9.