

Using Task Analytic Models to Visualize Model Checker Counterexamples

Matthew L. Bolton and Ellen J. Bass
Department of Systems and Information Engineering
University of Virginia
Charlottesville, VA
mlb4b@virginia.edu, ejb4n@virginia.edu

Abstract—Model checking is a type of automated formal verification that searches a system model’s entire state space in order to mathematically prove that the system does or does not meet desired properties. An output of most model checkers is a counterexample: an execution trace illustrating exactly how a specification was violated. In most analysis environments, this output is a list of the model variables and their values at each step in the execution trace. We have developed a language for modeling human task behavior and an automated method which translates instantiated models into a formal system model implemented in the language of the Symbolic Analysis Laboratory (SAL). This allows us to use model checking formal verification to evaluate human-automation interaction. In this paper we present an operational concept and design showing how our task modeling visual notation and system modeling architecture can be exploited to visualize counterexamples produced by SAL. We illustrate the use of our design with a model related to the operation of an automobile with a simple cruise control.

Index Terms—Human-automation interaction, task analysis, model checking, formal methods, counterexample visualization.

I. INTRODUCTION

Failures in complex, safety-critical systems are often due to system components interacting in unexpected ways. Model checking is an automated approach to formal verification which can be used to find these types of failures by exhaustively searching a system’s state space in order to determine if desired properties or specifications (usually written in temporal logic) hold [1]. If there is no violation, then the specification has been formally proven to be valid for the model. If there is a violation, an execution trace is produced (a counterexample). This counterexample depicts a model’s state corresponding to a specification violation along with a list of the incremental model states that led up to the violation.

Model checking has been used successfully to find interaction errors in computer hardware and software applications. However, one source of failures in complex, safety critical systems is the interaction between the human operator and other system components.

When designing for human-automation interaction (HAI), human factors engineers generally do not use formal models (see [2] for a review of the ways formal methods have been used to analyze HAI), but rather task analytic methods to represent human behaviors. Task analytic models represent descriptive and normative human behavior as sequences of activities with respect to the fulfillment of goals [3], [4].

We have developed a method [5] to evaluate HAI formally for systems where human tasks can be represented using task analytic behavior models. The method utilizes a formal modeling architectural framework which models human-automation interactive systems in terms of human missions (i.e. goals), human task behavior, human-device interfaces, device automation, and environment (see [6]). In our analysis framework, human task models are created using an intermediary language called Enhanced Operator Function Model (EOFM) [7], [8], an XML-based, generic human task modeling language based on the Operator Function Model (OFM) [9], [10]. EOFMs are hierarchical and heterarchical representations of goal driven, high-level, conceptual activities that decompose into lower level activities, and finally, atomic actions (discrete, observable, cognitive, or perceptual human behavior). EOFMs express task knowledge by explicitly specifying the conditions under which human operator activities can be undertaken: what must be true before they can execute (preconditions), when they can repeat (repeat conditions), and when they have completed (completion conditions). Each activity is composed of one or more other activities or one or more actions, represented as a decomposition in a hierarchy. A decomposition operator specifies the temporal relationships between and the cardinality of the decomposed activities or actions (when they can execute relative to each other and how many can execute).

EOFMs can be represented visually as a tree-like graph (see examples in Figs. 1–3). Actions are rectangles and activities are rounded rectangles. An activity’s decomposition is presented as an arrow, labeled with the decomposition operator, that points to a large rounded rectangle containing the decomposed activities or actions. In the work presented here, three decomposition operators are used: (a) *ord* (all activities or actions in the decomposition must execute in the order they appear); (b) *or_seq* (one or more of the activities or actions in the decomposition must execute); and (c) *xor* (exactly one activity or action in the decomposition must execute). Conditions on activities are represented as shapes or arrows (annotated with the condition logic) connected to the activity that they constrain. A precondition is represented as a yellow, downward-pointing triangle; a completion condition is a magenta, upward-pointing triangle; and a repeat condition is an arrow recursively pointing to the top of the activity. More details can be found in [7] and [8].

The EOFM language is used to represent the structure of human task analytic models. The details of how an instantiated EOFM executes based on the represented structure is controlled by the language's formal semantics [8]. Thus, in order to be included in formal verification analyses, instantiated EOFMs must be converted into a formal notation which implements the represented human task based on its semantic interpretation. In our method, EOFM's formal semantics are used to translate instantiated EOFM task models into the language of the Symbolic Analysis Laboratory (SAL) [11]: a framework for combining different tools to calculate properties of concurrent systems which includes a state of the art symbolic model checker called SAL-SMC. SAL-SMC is used to perform formal verifications on the complete system model. Each step in a SAL produced counterexample is presented as a list of variable values sorted by variable name, with no indication of the relationships between variables. Further, the translation process divorces the formal representation of the human task behavior from its original EOFM instantiation. Because there has been little work investigating formal verification of HAI, there is no precedent for how to visualize task analytic behavior from model checker counterexamples. This paper reviews the literature that does exist on counterexample visualization. Concepts from existing techniques are incorporated into a design in which the HAI architectural framework and the visual notation supported by the EOFM are used to create a counterexample visualization to help analysts evaluate the role of HAI in specification violations.

II. COUNTEREXAMPLE VISUALIZATION TECHNIQUES

Tables, state diagrams, process and sequence diagrams, and domain or application dependent representations have been used to display counterexamples.

A. Variable Tables

Tables depict variable values in each counterexample step. Table representations are supported by some model checkers such as the Cadence SMV [12]. These tables may sometimes highlight changes in variables to make it easier to compare variable values between steps [13], [14]. They can also allow analysts to hide variables and steps from the display [15].

B. State Diagrams

Some visualizations illustrate counterexample steps using state diagrams. Tools such as UPPAAL2k [16] and STATEMATE [17] represent models using the statecharts visual formalism [18] and allow counterexamples to be visually inspected by highlighting states and/or transitions active at each step.

Some tools [19]–[21] allow representations of a model's state space to be interactively explored to find witnesses or counterexamples. In these representations, a subset of the systems states are depicted as nodes in a directed graph with model specified transitions between them. The presented state space subset is calculated from user specifiable proof strategies [19], [21] or statistical measures [20]. Nodes can be interactively examined in order to get a complete view of their

variable values, and the tools provide feedback about whether temporal logic properties evaluate to true. Human analysts can expand their search of the state space by interactively selecting paths or applying new proof strategies. The tools facilitate this task by highlighting paths statistically likely to produce a counterexample [20] or label each graph edge with the proof steps used to reach the target states [19].

C. Process and Sequence Diagrams

With process and sequence diagrams, variables are either grouped into domain relevant designations (such as processes or threads for software) or represented by themselves [14], [16], [21]–[26]. These designations are listed together along an axis. Time or execution order is represented along a perpendicular axis. A mark (such as a dot or box) in this visualization indicates a change in a designation's state at the given time or step. It can often be inspected to obtain state variable values. Marks connected at a given step or time by a line or arrow can either show synchronized changes between the designations [23] or information passing [22]. Color can be used to convey the truth value of temporal logic properties [23], [24] at each time or step.

D. Domain and Application Dependent Representations

Representations specific to a particular domain or application, such as timing plots for models of computer hardware [27], [28], have been employed. Models animating human interaction with visual prototypes of human-device interfaces for counterexamples and formal model simulation traces have also been explored [14], [29].

E. Common Features

These visualization share common features that support evaluation including:

1) *Encapsulation of Variables or States*: The sequence diagrams and application dependent representations allow model state variables to be encapsulated in higher level designations that are familiar to the analyst. These also sort model variables and provide other formatting enhancements.

2) *Interactive Detail Refinement*: By allowing analysts to interactively query for information not currently displayed, visualizations support encapsulation without removing access to detailed information.

3) *Highlighting Analysis Relevant Changes*: The discussed visualizations make variable changes salient at several levels: individual variables (tables, sequence diagrams, and state diagram animations); designations; and system properties specified in temporal logic. Sequence diagrams and tables show coordination between changes in state between different variables or designations. Some sequence diagram and state diagrams provide visual feedback about the value of analysis-relevant temporal logic properties.

III. OPERATIONAL CONCEPT AND DESIGN

An analyst performing formal verifications of HAI is interested in determining how human behavior may contribute to a violation of a system specification. The analyst wants

to examine the conditions under which the human operator performs actions and what the impact of those actions are on the other elements of the system. To support analysts performing such a verification using SAL, our architectural framework, and our EOFM task modeling language; we are developing a counterexample visualization that draws from the common features of other visualization techniques and EOFM's visual notation. Example designs can be seen in Figs. 1-3. These are discussed in greater depth in Section IV.

A. Encapsulation of Variables or States

The visualization encapsulates variable states based on the elements of the architectural framework (mission, task behavior, human-device interface, device automation, and environment) and EOFM's visual notation. High level designations of these are organized in a table with designations as the rows and counterexample steps as the columns¹.

B. Interactive Detail Refinement

The visualization allows an analyst to interactively inspect the values of encapsulated variables. An analyst can traverse the table by either using navigation buttons (previous: ◀ and next: ▶) or by scrolling through the table cells and selecting a specific step¹. Selecting a cell moves a cursor (↓) that points to the associated step's column and presents a detailed view of the model variable values in a separate, detailed view. This view presents the individual variable values from the system model elements for the given step in separate labeled columns. It also displays the execution state of the human task task behavior model using EOFM's visual notation².

The formal semantics of the EOFM language [8] specify that each activity and action in an instantiated EOFM has three possible execution states: *Ready* (waiting to execute), *Executing*, and *Done* (finished executing). When the visualizer presents the execution state of the human operator's task behavior model, it renders the entire graph for the goal level (root) activity that is executing. It color codes each of the activities and actions in this structure to indicate its execution state: white for *Ready*, green for *Executing*, and grey for *Done*.

C. Highlighting Analysis Relevant Changes

The visualization highlights changes in variable values in both the high level encapsulated designations and the individual variables. At the high level, each table cell's color indicates whether or not there has been a change in one of the associated designation's variables since the previous step: white for no change, and yellow for a change. In our architecture, human task behavior can only produce a change in the other elements of the system architecture if a human action is performed. Thus, a counterexample step with a human action is indicated with an 'X' in its corresponding table cell. In the detailed view, variables whose values changed from the previous step are highlighted in yellow. An activity or action whose execution

state has changed since the previous step is presented with a yellow highlight around its border³.

D. Implementation

Our counterexample visualization prototype is implemented in Microsoft Visio and Visual Basic for Applications. With an instantiated EOFM and a text file containing a SAL produced counterexample as input, the software identifies variables that represent the human task behavior model. The analyst indicates under which architectural designation the remaining variables belong: mission, human-device interface, device automation, or environment. The software then renders each detailed view as a separate page in a Visio document. The high-level encapsulation table (labeled "Navigation Form") is presented as a dialog box that is presented over each page.

IV. EXAMPLE

The following example illustrates the visualization using a formal model of a human operator driving towards a traffic light in a car equipped with cruise control. The driver's goal is to drive at his/her desired (mission) speed while safely responding to traffic lights and avoiding merging traffic. Traffic can merge from a ramp intersecting the road before the traffic light. For modeling purposes, the relative distance between the car and the light is: Very Very Far, Very Far, Far, Merging, Close, Very Close, and At Intersection.

The driver can drive at one of 3 speeds (Slow, Moderate, or Fast). Increasing the gas (pressing the gas pedal) causes the car to accelerate to the next faster speed and decreasing the gas decelerates the car to the next slower speed. The driver can release the gas pedal and the car decelerates until it stops or the driver can press the break and the car stops more quickly. The driver can enable or disable the cruise control using available buttons. Pressing the break also disables the cruise control. The cruise control will maintain the car's current speed unless the driver increases the gas, where the driver controls the cars speed above the cruise speed.

The formal system model architecture [6] includes the human-device interface (the pedal and the car's indicated speed), device automation (the car's speed and acceleration), operational environment (the color of the light, its relative position to the car, and whether or not there is merging traffic), and human mission (the driver's desired speed). It also contains the human task behavior which is translated from an instantiated EOFM for the driving tasks: driving at the desired speed (adjusting the car's speed and cruise control), avoiding merging traffic (accelerating or slowing down to avoid the traffic), and responding to the light (waiting until very close to the light and breaking, or rolling to a stop from further away). A full specification of this model can be found in [8].

SAL-SMC was used to formally verify that the driver will never run a red light (reach the intersection when the light is red with the car not being stopped). This is specified in linear

¹See the "navigation form" in Figs. 1-3.

²Everything outside of the "navigation form" in Figs. 1-3.

³Examples of highlighting can be seen in Figs. 1-3.

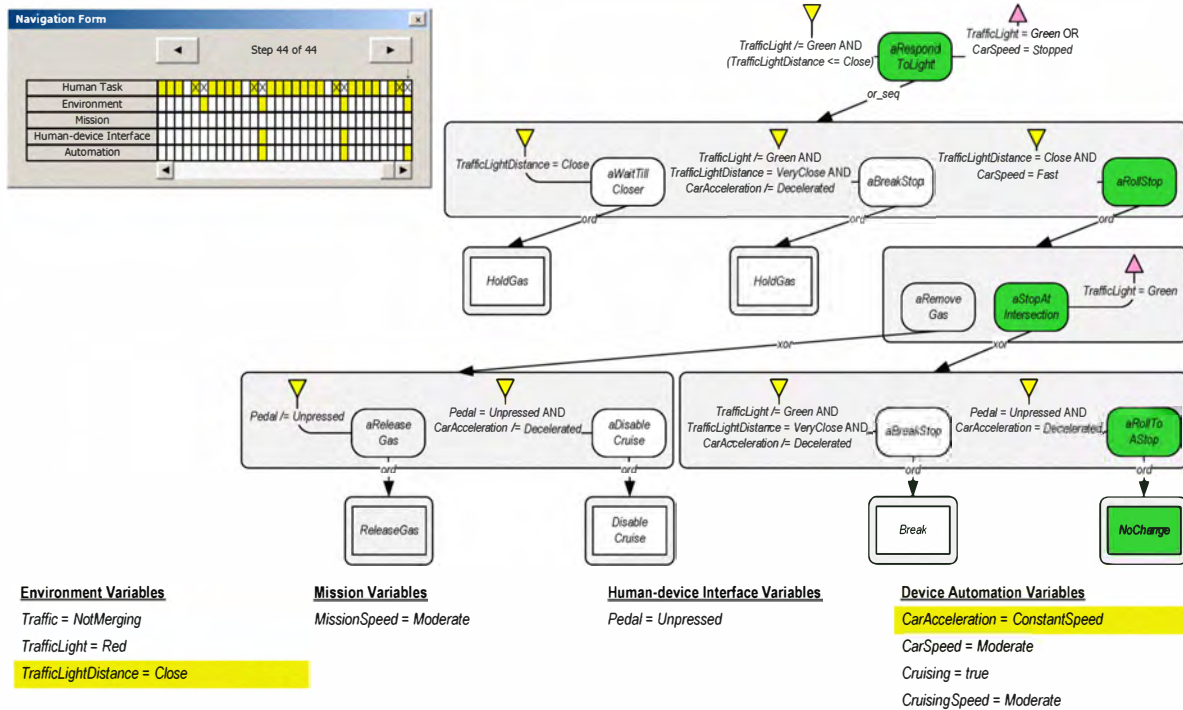


Fig. 1. Step 44 from the visualization of the counterexample. The car has reached the intersection while moving at a constant, moderate speed while the traffic light is red, a violation of (1). The operator has attempted to perform a roll stop by keeping his/her foot off of the gas pedal. However, this has not impacted the car's speed because the cruise control is enabled and the car is at its cruise speed.

temporal logic as:

$$G \neg \left(\begin{array}{l} \text{TrafficLightDistance} = \text{AtIntersection} \\ \wedge \text{TrafficLight} = \text{Red} \wedge \text{Car} \neq \text{Stopped} \end{array} \right) \quad (1)$$

When checked against the formal system model, this specification produced a counterexample. The violation occurs in the last step of the counterexample (step 44; Fig. 1): where the driver is attempting to roll to a stop at the traffic light. The driver is not pressing the gas pedal, but the car is moving at a moderate, constant speed (not decelerating) with the cruise control enabled. Here, the driver has attempted to roll to a stop without disabling the cruise control.

The driver had previously attempted to slow the car by releasing the gas pedal (*aReleaseGas* and *aRemoveGas*) rather than disable the cruise control (Fig. 1). By stepping through the counterexample with the navigation form, we see that at step 33 (Fig. 2), the driver should disable the cruise control as part of performing a roll stop (*aDisableCruise* via *aRemoveGas* and *aRollStop*). However, even though cruise is enabled (*Cruising = true* under 'Device Automation Variables'), the driver is pressing the gas pedal. Thus he/she performs the activity for releasing the gas pedal (*aReleaseGas*) instead.

The navigation form shows when the last human action was performed (the 'X' at step 26). The associated detailed view (accessed by clicking on the 'X') reveals that while cruise was enabled, the driver pressed the gas pedal (*IncreaseGas*) to let merging traffic in behind (Fig. 3). Thus, the visualization shows that this violation occurs when the driver accelerates to

go ahead of merging traffic after enabling the cruise control and then attempts to roll to a stop at a traffic light.

V. DISCUSSION

When using our HAI architectural framework and our EOFM task modeling language, the counterexample visualization supports the interpretation of a SAL produced counterexample by providing a means of determining how human task behavior interacts with other system architectural elements leading up to a specification violation. It accomplishes this by exploiting the visual notation of the EOFM and useful features of other visualization techniques: (a) It encapsulates variables into designations from the architectural framework and presents task model data using the EOFM visual notation; (b) Counterexample steps can be interactively inspected to display detailed human task behavior and other model variable state information; and (c) It highlights changes in the high level encapsulation view, the detailed view of the variables, and the human task behavior execution state.

While the application presented here is simple for illustrative purposes, our visualization has been used successfully with a number of other applications including a pain medication pump, a radiation therapy machine, and an aircraft on approach [2]. Because both the abstracted encapsulation of variables (the "navigation form") and the detailed display of variable values are similar to table visualizations, they should scale for systems with larger numbers of variables [13]. However, because human task behavior is being represented diagrammatically, it may not scale well for applications with large task

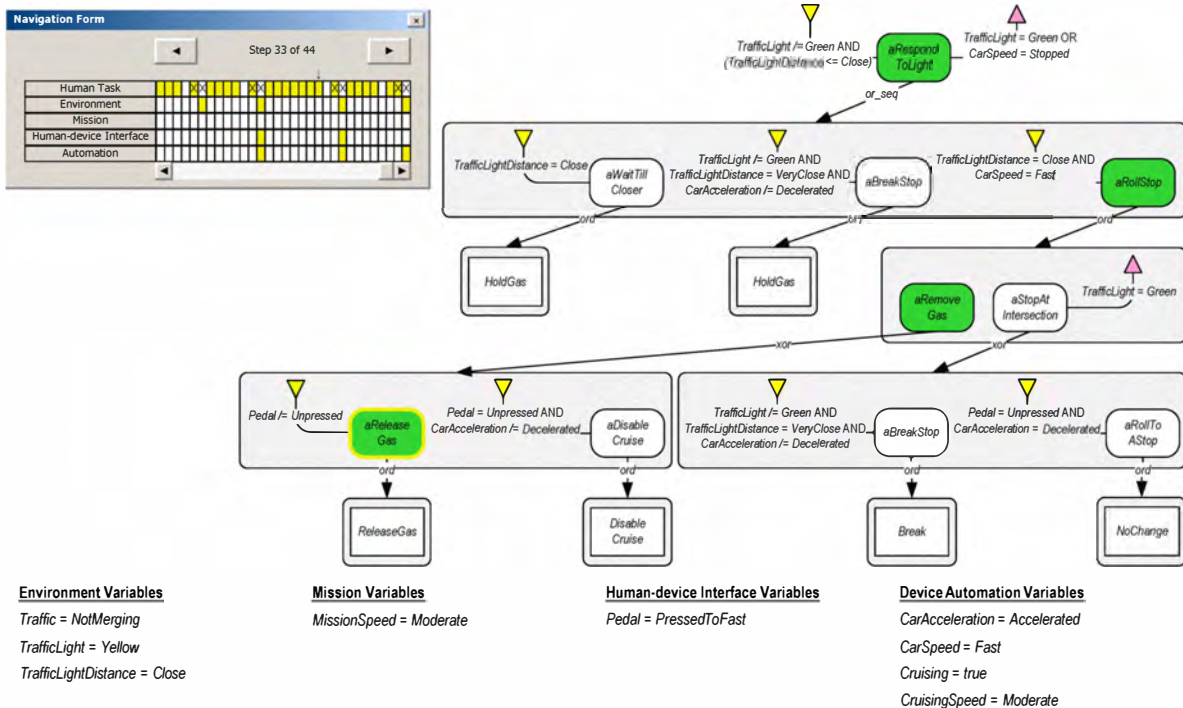


Fig. 2. Step 33 from the visualization of the counterexample. The car is close to the light and the light is yellow. Thus the driver is attempting to respond to the light by performing a roll stop. To accomplish this task, he/she “removes” gas from the car by performing the activity for releasing the gas pedal.

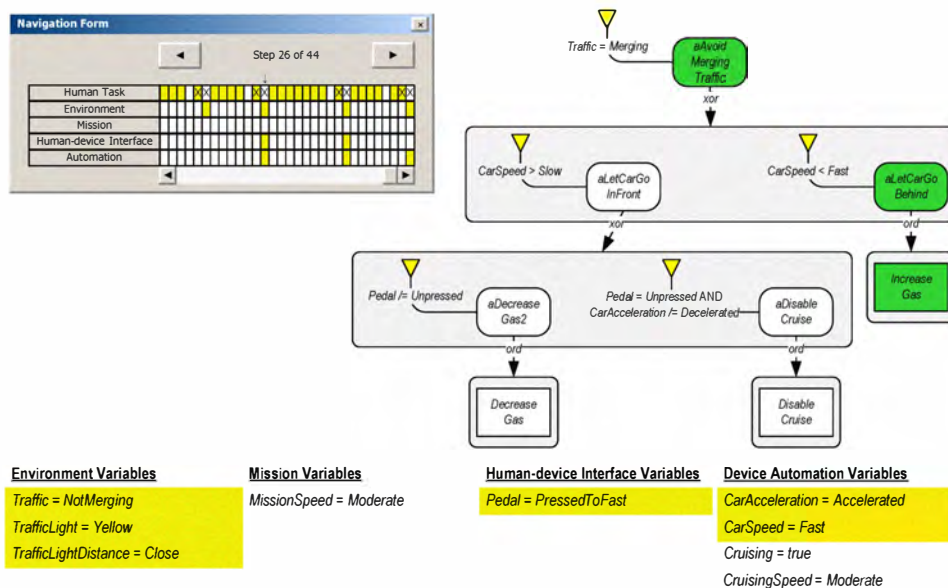


Fig. 3. Step 26 from the visualization of the counterexample. The driver has just performed the action for increasing the gas to avoid merging traffic by letting the merging traffic pass behind. This has caused the gas pedal to be pressed to the “fast” position, with the car accelerating to the fast speed, allowing the traffic to merge, and the car to go to the “Close” interval. The traffic light has also turned yellow.

structures or systems with multiple human operators. Future work should attempt to identify how our visualization scales to larger applications and what applications it is suited for.

There are other features that could be incorporated into our visualization. Several existing visualizations [19]–[21], [23], [24] provide feedback about how logically expressed properties

evaluate at each step, often with color coding. While our visualization displays the conditions associated with task behavior activities, it does not provide any visual indication of how they evaluate (*true* or *false*). Additionally, our visualization provides no feedback about the evaluation of temporal logic properties (like that used to produce the counterexample). Future work

should investigate how to incorporate visual feedback about property evaluations into our visualization.

Interactive statespace exploration is supported by [19]–[21]. SAL-SMC counterexamples represent a single execution path through the model and thus do not facilitate additional model exploration. Future work should investigate how statespace exploration could be made to work with our visualization.

Our visualization was explicitly designed to help analysts interpret counterexamples. However, such a visualization may be useful for debugging models during development. Future work should determine what the requirements are for debugging and related visualization tools.

Analysts may wish to compare different counterexamples to diagnose potentially related specification violations. Future work should investigate what the requirements are for such a feature and potentially adapt the visualization to support it.

Because there are no existing visualizations designed explicitly to help analysts interpret counterexamples utilizing task analytic models, there are no analogous technologies to compare with ours. Our visualization uses the EOFM's visual notation to display the state of task analytic behavior models coupled with concepts employed by variable tables and sequence diagrams. This would suggest that our visualization is better suited to analyses with our method [5] than with SAL's default counterexample display, variable tables, or sequence diagrams. Future work should conduct human subject experiments to determine if this is the case.

ACKNOWLEDGEMENT

This work was supported in part by Grant Number T15LM009462 from the National Library of Medicine (NLM) and Research Grant Agreement UVA-03-01, sub-award 6073-VA and 2723-VA from the National Institute of Aerospace (NIA). The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIA, NASA, NLM, or NIH. The authors would like to thank Radu I. Siminiceanu of the NIA and Ben Di Vito of the NASA Langley Research Center for their technical help and support.

REFERENCES

- [1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge: MIT Press, 1999.
- [2] M. L. Bolton, "Using task analytic behavior modeling, erroneous human behavior generation, and formal methods to evaluate the role of human-automation interaction in system failure," Ph.D. dissertation, University of Virginia, Charlottesville, 2010.
- [3] B. Kirwan and L. K. Ainsworth, *A Guide to Task Analysis*. London: Taylor and Francis, 1992.
- [4] J. M. Schraagen, S. F. Chipman, and V. L. Shalin, *Cognitive Task Analysis*. Philadelphia: Lawrence Erlbaum Associates, Inc., 2000.
- [5] M. L. Bolton and E. J. Bass, "A method for the formal verification of human interactive systems," in *Proceedings of the 53rd Annual Meeting of the Human Factors and Ergonomics Society*. Santa Monica: Human Factors and Ergonomics Society, 2009, pp. 764–768.
- [6] —, "Formally verifying human-automation interaction as part of a system model: Limitations and tradeoffs," *Innovations in Systems and Software Engineering: A NASA Journal*, 2010, doi:10.1007/s11334-010-0129-9.
- [7] —, "Enhanced operator function model: A generic human task behavior modeling language," in *Proceedings of the IEEE International Conference on Systems Man and Cybernetics*. Piscataway: IEEE, 2009, pp. 2983–2990.

- [8] M. L. Bolton, R. I. Siminiceanu, and E. J. Bass, "A systematic approach to model checking human-automation interaction using task-analytic models," under review.
- [9] C. M. Mitchell and R. A. Miller, "A discrete control model of operator function: A methodology for information display design," *IEEE Transactions on Systems Man Cybernetics Part A: Systems and Humans*, vol. 16, no. 3, pp. 343–357, 1986.
- [10] D. A. Thurman, A. R. Chappell, and C. M. Mitchell, "An enhanced architecture for OFMspert: A domain-independent system for intent inferring," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. Piscataway: IEEE, 1998.
- [11] L. De Moura, S. Owre, and N. Shankar, "The SAL language manual," Computer Science Laboratory, SRI International, Menlo Park, Tech. Rep. CSL-01-01, 2003.
- [12] K. McMillan, "The cadence smv model checker," <http://www.kenmcml.com/smv.html>.
- [13] K. Loer and M. D. Harrison, "An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation," *Automated Software Engineering*, vol. 13, no. 4, pp. 469–496, 2006.
- [14] J. C. Campos and M. D. Harrison, "Interaction engineering using the ivy tool," in *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York: ACM, 2009, pp. 35–44.
- [15] M. Kermelis, "Towards an improved understanding of model-checking traces by visualisation," Master's thesis, University of York, York, 2003.
- [16] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannot, K. G. Larsen, M. O. Miller, P. Pettersson, C. Weise, and W. Yi, "UPPAAL - now, next, and future," in *Proceedings of the 4th Summer School Modeling and Verification of Parallel Processes*. Berlin: Springer, 2001, pp. 99–124.
- [17] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, 1990.
- [18] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.
- [19] M. Chechik and A. Gurfinkel, "A framework for counterexample generation and exploration," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 429–445, 2007.
- [20] H. Aljazzar and S. Leue, "Debugging of dependability models using interactive visualization of counterexamples," in *Proceedings of the 5th International Conference on Quantitative Evaluation of Systems*. Los Alamitos: IEEE Computer Society, 2008, pp. 189–198.
- [21] Y. Dong, C. R. Ramakrishnan, and S. A. Smolka, "Evidence Explorer: A tool for exploring model-checking proofs," in *Proceedings of the 15th International Conference on Computer Aided Verification*. Berlin: Springer, 2003, pp. 215–218.
- [22] C. Artho, K. Havelund, and S. Honiden, "Visualization of concurrent program executions," in *Proceedings of the 31st Annual International Computer Software and Applications Conference*. Piscataway: IEEE, July 2007, pp. 541–546.
- [23] P. Kemper and C. Tepper, "Trace analysis—gain insight through model-checking and cycle reduction," Dortmund University of Technology, Dortmund, Tech. Rep. 06007, 2006.
- [24] R. Simmons and C. Pecheur, "Automating model checking for autonomous systems," in *Proceedings of the AAAI Spring Symposium on Real-Time Autonomous Systems*. Menlo Park: AAAI Press, pp. CD–ROM.
- [25] G. J. Holzmann, *The Spin Model Checker, Primer and Reference Manual*. Reading: Addison-Wesley, 2003.
- [26] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran, "MOCHA: Modularity in model checking," in *Proceedings of the 10th International Conference on Computer Aided Verification*. Berlin: Springer, 1998, pp. 521–525.
- [27] S. Jeon, "Verification of function block diagram through verilog translation," Master's thesis, Korea Advanced Institute for Science and Technology, Daejeon, 2007.
- [28] P. C. Tae Jung Kim, "smv2vcd," Pittsburgh, December 2001, <http://www.cs.cmu.edu/~modelcheck/smv2vcd.html>.
- [29] Y. Ait-Ameur and M. Baron, "Formal and experimental validation approaches in HCI systems design based on a shared event B model," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 6, pp. 547–563, 2006.