Proceedings of the 8th Annual Applied Ergonomics Conference. New Orleans, LA, 2005.

UIVerify - a Web-Based Tool for Verification and Automatic Generation of User Interfaces

Smadar Shiffman
NASA Ames Research Center/QSS
Moffett Field, CA 94305
shiffman@email.arc.nasa.gov

Asaf Degani
NASA Ames research Center
Moffett Field, CA 94305
adegani@mail.arc.nasa.gov

Michael Heymann
Department of Computer Science
Israel Institute of Technology
heymann@cs.technion.ac.i

ABSTRACT

In this poster, we describe a web-based tool for verification and automatic generation of user interfaces. The verification component of the tool accepts as input a model of a machine and a model of its interface, and checks that the interface is adequate (correct). The generation component of the tool accepts a model of a given machine and the user's task, and then generates a correct and succinct interface. This write-up will demonstrate the usefulness of the tool by verifying the correctness of a user interface to a flight-control system. The poster will include two more examples of using the tool: verification of the interface to an espresso machine, and automatic generation of a succinct interface to a large hypothetical machine.

Keywords

Formal methods, verification, interface design, interface analysis, automation, autopilot, flight control systems

INTRODUCTION

Computers and automated devices play a pivotal role in our modern society. Almost every technological system is driven by computers; efficient user interaction is a key element for making these systems efficient and, in the case of high-risk systems such as aircrafts, also safe. One of the factors frequently cited in aviation incident and accident reports as being responsible for faulty system operation is the fact that the interface between the user and the machine is inadequate, and at times misleading [1]. The problem, however, is not only unique to aviation—it can also be found in many other human-machine systems, from household devices to consumer electronics, automotive, and medical systems [2]. The difficulty in designing adequate interfaces is only bound to grow in the future, as computer systems are becoming more sophisticated and complex.

For an interface to be adequate, it must first and foremost be correct, so that the user always knows the current configuration of the machine (e.g. mode) and can predict the next mode. In addition, we strive for interfaces that are succinct, so that they do not overload the user with superfluous information. In this poster, we describe UIVerify—a web-based tool that allows designers of computer-based systems to analyze user interfaces. UIVerify includes a component for verifying the correctness of an interface, and a component for generating correct and succinct interfaces. The methodologies behind the tools are based on published work on verification and generation of user interfaces [3, 4].

VERIFICATION OF INTERFACES

Formal verification of user interfaces entails checking whether the user can interact with the underlying machine correctly and reliably in order to achieve a specified set of tasks. The verification process detects three types of user-interface inadequacies that are based on the criteria set by Degani and Heymann [4]. The first inadequacy—the existence of *error states*—occurs when the user interface indicates that the machine is in one mode, when in fact the machine is in another. Interfaces with error states lead to faulty interaction and operational errors. The second inadequacy—the existence of *restricting states*—occurs when the user is unaware that certain user interactions can trigger additional mode changes in the machine. Interfaces with restricting states tend to surprise and confuse users. The third inadequacy—the existence of *augmenting states*—occurs when the user is told that a certain mode is available, when in fact the machine does not have this mode, or access to certain modes is disabled. Interfaces with augmenting states puzzle users and have contributed to operational errors.

At the heart of the interface design problem lies the fact that in any human-machine system, two concurrent processes are at constantly at play: the machine and its (internal) behavior, and the information provided to the user (about the behavior of the underlying machine) via the interface. Every interface is an abstraction, or simplification, of the underlying machine behavior [3]. The objective of the verification process is to systematically and comprehensively validate that the abstracted interface is correct. One way to view how the verification process detects user-interface inadequacies is by constructing a composite model that incorporates the underlying behavior of the machine (*machine model*) and the user-interface information (*user model*). In this composition, called the *composite model*, we combine corresponding user-interface states and machine states into state-pairs. Next, the verification process simulates an activation of the composite model, where the user-interface model and the machine model evolve concurrently in a synchronized manner. If the process cannot detect instances of error states, restricting states, or

augmenting states, the user-interface model, according to these three verification criteria, is considered correct.

GENERATION OF INTERFACES

A second component of UIVerify is a tool for automatic generation of interfaces. For a given machine (and description of the user's task), the tool generates the simplest (e.g., minimal) interface possible. The idea is to provide the user with a correct interface that does not include any superfluous information; in other words, the objective is to de-clutter the display as much as possible. The interface generation methodology incorporates a variant of a reduction algorithm [5] to produce an abstraction of the machine's model that included only the details that are necessary for correct user interaction [3, 4, 6].

THE UIVerify WEB-BASED TOOL

UIVerify includes both the verification and generation components, and comprises a backend, which runs the verification and generation processes, and a front end, which communicates with the analyst using a web browser. The verification and generation packages are coded in GNU C++. Through the browser, the user of the system (which we will call from here on the analyst), inputs the machine model, the user model, and the correspondence between these two models. The analyst inputs the models by uploading the description from a text file or by manually typing the model description into a form. The tool displays each loaded model, both in textual form and in graph form, for analyst confirmation. The tool generates the graph form of the models with GraphViz [7, 8]. Upon the analyst's request, Java™ servlets activate the verification and generation processes through Java™ system calls. The servlets are synchronized to allow for correct simultaneous activation of the tool via multiple client browsers. The results are displayed within the web browser. The front-end pages are served from a SUN™ workstation using an Apache web server.

USING THE TOOL

The poster will demonstrate how to work with UIVerify by showing three examples. The first example shows how the tool detects an error state in the interface to an espresso machine. The second example shows how the tool detects an error state in a flight-control system. The third example shows how the tool generates a simplified interface for a large hypothetical machine model. Here, in this description, we only show the process of inputting the data (models) of the flight control system and the results of the verification as conducted by the tool.

Machine Model

Figure 1 shows the behavior of the automatic flight control system (machine model). It is a finite state machine description of how the autopilot works. There are several states/modes (e.g., CAPTURE, VERTICAL SPEED) and transitions in-between (e.g., *engage change level*). The analyst enters the model in Figure 1 into the tool as a set of fragments, or tuples, which include the BEGINNING STATE, *transition*, and END STATE. For example, the tuple: CAPTURE, *set altitude ahead of capture start*, and VERTICAL SPEED (to altitude setting) is one fragment. For the model in Figure 1, there are 18 such tuples that account for all the states and transitions in the model. In addition, the analyst also indicates the specification class for each state. For example, the specification class for the state VERTICAL SPEED (to altitude setting) is **ARMED FOR CAPTURE**. Figure 2 shows the confirmation screen for all the inputs that were entered by the analyst.

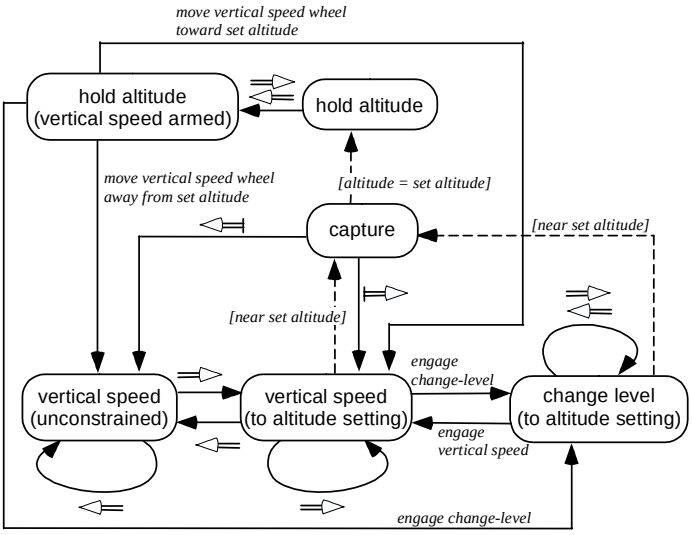

Figure 1 Machine model of automatic flight control. The symbol ⟹ indicates the event 'set altitude *ahead* of current aircraft altitude'; the symbol ⟾ indicates 'set altitude *ahead* of capture start'; the symbol ⟸ indicates 'set altitude *behind* current aircraft altitude'; the symbol ⟽ indicates 'set altitude *behind* capture start.'

User Model

Figure 3 is the user model. It shows the information provided to the pilot about the system (through the interface and in the flight manual). The analyst enters the user model description as tuples (in the same way as the machine model). The interface (and hence also the user model) is an abstracted description of the underlying machine model. As can be seen in Figure 3, the user model is simplified in the sense that all altitude-setting events are related to the current aircraft altitude (e.g., the transition from CAPTURE to VERTICAL SPEED (unconstrained)). In the machine model, however, the description is more refined such that it also includes the subtlety of '*setting altitude to ahead/behind capture start*.'

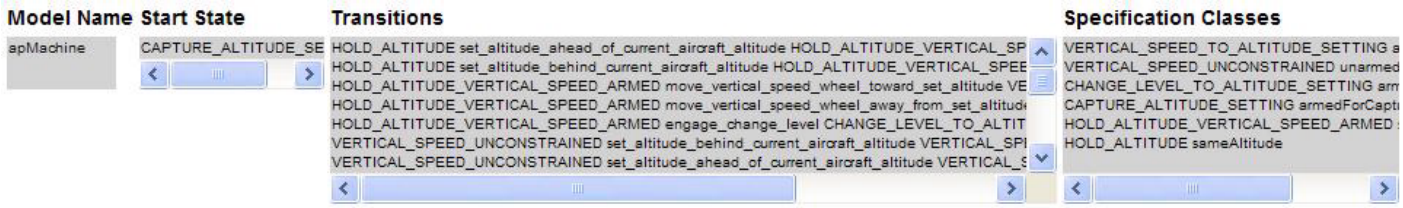

Figure 2 The input-confirmation screen for the machine model

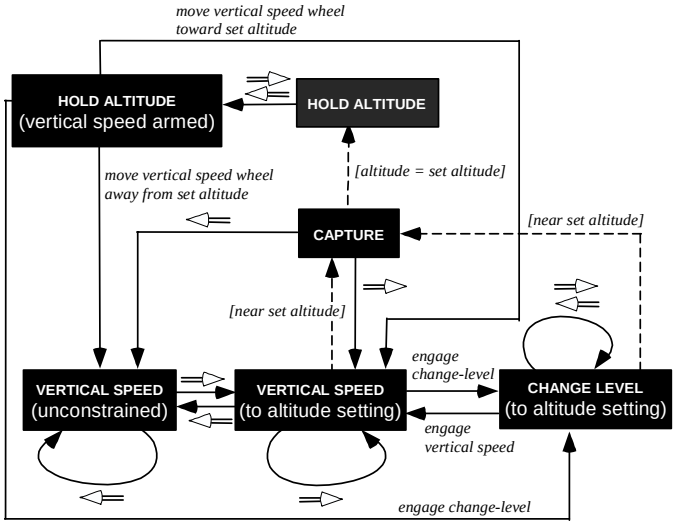

Figure 3 User model of automatic flight control. The symbol ⟹ indicates the event 'set altitude *ahead* of current aircraft altitude'; the symbol ⟸ indicates 'set altitude *behind* current aircraft altitude.'

Event Correspondence

By now we have entered both the machine model and then the user model into the tool. The third step is to enter the correspondence between the events in the machine model vs. the events in the user model. The correspondence between machine-model events to user-model events is shown in Table 1. We can see that the events *set altitude behind capture start* and *set altitude ahead of capture start* are simplified in the user model. Figure 4 shows the confirmation screen for the event mapping.

Table 1 Correspondence of machine-model events with user-model events

Machine-model events	User-model events
• move vertical speed wheel *toward* set altitude	• move vertical speed wheel *toward* set altitude
• move vertical speed wheel *away* from set altitude	• move vertical speed wheel *away* from set altitude
• engage change level	• engage change level
• engage vertical speed	• engage vertical speed
• near set altitude	• near set altitude
• altitude equals set altitude	• altitude equals set altitude
• set altitude *ahead* of current aircraft altitude ⟹	• set altitude *ahead* of current aircraft altitude ⟹
• set altitude *behind* current aircraft altitude ⟸	• set altitude *behind* current aircraft altitude ⟸
• set altitude *behind* capture start ⟸|	• set altitude *behind* current aircraft altitude ⟸
• set altitude *ahead* of capture start |⟹	• set altitude *behind* current aircraft altitude ⟸
• set altitude *ahead* of capture start |⟹	• set altitude *ahead* of current aircraft altitude ⟹

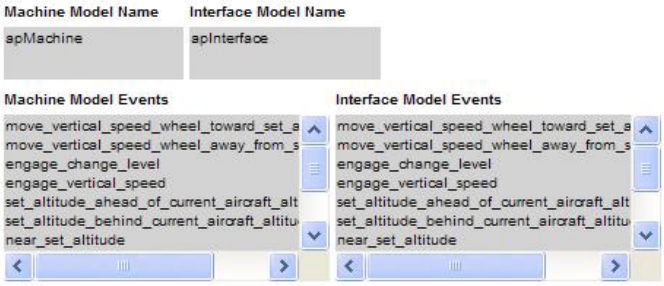

Figure 4 The confirmation screen for the correspondence between machine-model and user-model events.

At this point, the tool is ready to verify the interface. The computation time for a model with 19 states takes about 1 second and the results show that UIVerify detects an *error state* between machine-model state VERTICAL SPEED (to altitude setting) and user-model-state VERTICAL SPEED (unconstrained) (see Figure 5).

To visualize and better understand this error state inadequacy that was detected by the tool, let's take the machine model, the user model, and build a composite model (Figure 6). Figure 6(a) is a portion of the machine model, showing the consequences of changing the altitude while in capture mode. Figure 6(b) is a portion of the user model, also showing the consequences of changing altitude while in capture mode. (Recall that the simplification in the user model was performed on the events leading out of the capture mode). Now look at the composite model of Figure 6 (c): For changing the altitude to above the current aircraft altitude, the simplification works just fine. But for changing the altitude to below the current aircraft altitude, the simplification creates an error state: Based on this display and knowledge, the pilot assumes that the aircraft will always go into unconstrained climb, when in fact, it may sometimes capture the altitude. This discrepancy will always happen when the new altitude setting is below the capture start. Naturally, if the pilot cannot discriminate whether the airplane will continued to climb indefinitely in vertical speed (unconstrained), or go into vertical speed (to altitude setting) and then capture the newly set altitude—the interface is indeed incorrect [9].

RELATED RESEARCH AND CONCLUSIONS

Several researchers demonstrated how formal methods can be used for analyzing user interfaces and identifying design deficiencies [10-12]. Rushby, et al., use model checking techniques to perform an iterative search for inconsistencies within a combined rule-based representation of machine and

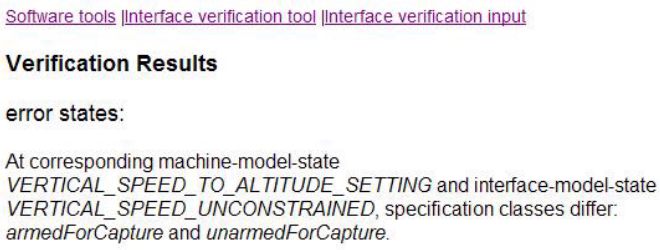

Figure 5 Verification results for flight-control example

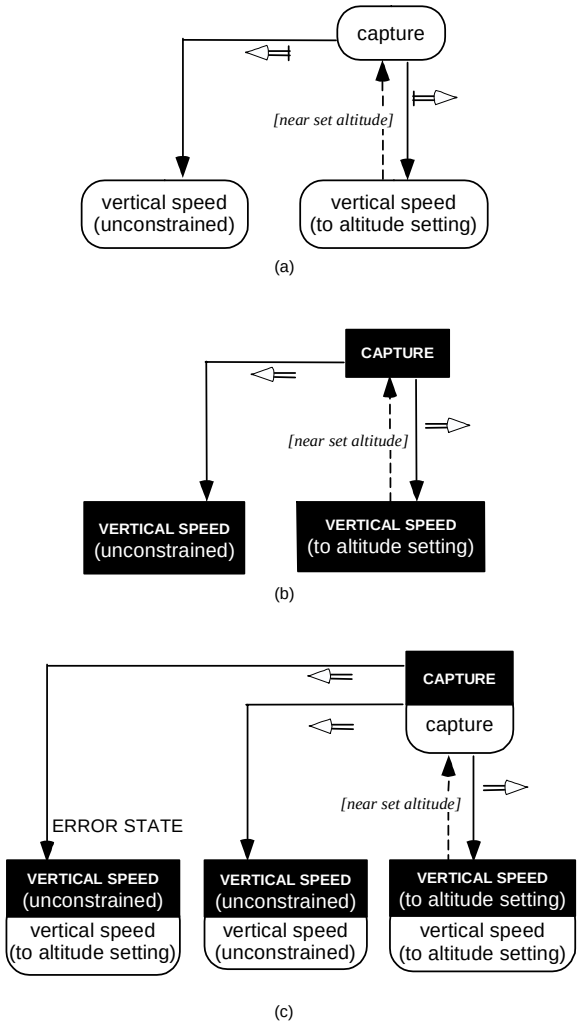

Figure 6 Creating a composite model: (a) machine model, (b) user model, and (c) composite model.

user models, and allow alteration of either the machine or the user model in-between iterations. The Degani and Heymann method, which is the approach and methodology behind UIVerify, employs the use of separate descriptions for the machine and the interface, focuses on the synchronization between the two concurrent models, and provides criteria (error state, restricting state, augmenting states) for verification. The UIVerify tool is applicable for user interfaces that have many discrete modes. As for control systems, where it is important to also take into account continuous variables (such as time, speed, flight path angle, etc.), it is possible in most cases to use the methods described in [13] so as to convert the (hybrid) system into a finite state machine representation and then use UIVerify to perform the verification. As for automatically generating correct and succinct interfaces, the method of Heymann and Degani and its implementation in UIVerify is unique in its capability to determine the simplest interface possible. The UIVerify tool, which is currently in a proof of concept phase, is available for use at http://uiverify.arc.nasa.gov.

REFERENCES

1. Leveson, N.G. and Palmer, E., Designing automation to reduce operator errors. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Orlando, FL. October, 1997.
2. Degani, A., (2004). *Taming HAL, Designing Interfaces Beyond 2001*. Palgrave Macmillan, New York.
3. Degani, A. and Heymann, M., (2002). Formal verification of human-automation interaction. *Human Factors*, Vol. 44.1, pp 28-43.
4. Heymann, M. and Degani, A., (2002). On abstractions and simplifications in the design of human-automation interfaces. *NASA Technical Memorandum* 2002-21397, Mofett Field, CA.
5. Paull, M.C. and Unger, S.H. (1959). Minimizing the number of states in incompletely specified sequential switching functions. *Institute of Radio Engineers-- Transactions on Electronic Computers*, 1959, pp. 356-367.
6. Degani, A. and Heymann, M., Meyer, G., and Shafto, M., (2002). Some formal aspects of human-automation interaction. *NASA Technical Memorandum* 2000-209600, Mofett Field, CA.
7. North, S.C. and Koutsofios, E., Application of Graph Visualization, in *Proceedings of Graphics Interface*, Banff, Alberta, Canada, 1994, pp. 235-245.
8. Gansner, E.R. and North, S.C., An open graph visualization system and its applications to software engineering. Software - Practice and Experience (SPE), Vol. 30.11, 2000, pp. 1203-1233.
9. Degani, A., Heymann, M., (2000). Pilot-autopilot interaction: A formal perspective. In Abbott, K., Speyer, J.J., Boy, G., eds.: *Proceedings of the International Conference on Human-Computer Interaction in Aeronautics: HCI-Aero 2000*, Toulouse, France, pp. 157-168.
10. Palanque, P. and Bastide, R., (1994). Petri-net based design of user-driven interfaces using the interactive cooperative objects formalism, *in Proceedings of Design, Specification and Verification of Interactive Systems*, Springer Verlag, pp. 383-400.
11. Rushby, J., Using model checking to help discover mode confusions and other automation surprises, in *Proceedings of the Workshop on Human Error, Safety, and System Development (HESSD)*, Li`ege, Belgium, 1999.
12. Doherty, G., Campos, J. C. and Harrison, M.D., Representational reasoning and verification, *Formal Aspects of Computing*, No. 3, 2000, pp. 1-23.
13. Oishi M., Tomlin, C. and Degani, A. (2003). Discrete abstraction of hybrid systems: verification of safety and application to user-interfaces. *NASA Technical Memorandum* 2003-212803, Mofett Field, CA.