Formal Verification of Safety-Critical User Interfaces: A Space System Case Study

Manuel Sousa[1] and **José Creissac Campos**[1] and **Miriam Alves**[2] and **Michael D. Harrison**[3]

[1] Depart. Informática, Universidade do Minho & HASLab/INESC TEC, Campus de Gualtar, 4710-057 Braga, Portugal
[2] Instituto de Aeronáutica e Espaço, São José dos Campos, Brasil
[3] Newcastle University, Newcastle upon Tyne & Queen Mary, University of London, Mile End Road, London, UK
jose.campos@di.uminho.pt

Abstract

Safe operation of safety critical systems depends on appropriate interactions between the human operator and the computer system. Specification of such safety-critical systems is fundamental to enable exhaustive and automated analysis of operator system interaction. In this paper we present a structured, comprehensive and computer-aided approach to formally specify and verify user interfaces based on model checking techniques.

Introduction

Safe operation of safety-critical systems depends upon appropriate interaction between the human operator and the system. In a study by MacKenzie (1994), the vast majority of computer-related accidental deaths studied (92%) could be related to failures in such interaction. While it is common to attribute such accidents to "user error", in practice error is the consequence of a combination of factors, and the role played by the user interface should be carefully considered (Leveson 1995). Operators can, in many cases, solve more problems that they create (Nichols 1973).

Specifying user interfaces for safety-critical applications involves more than merely describing the graphical elements. Elements of behavior that are related to the interaction are crucial too. The user interface defines a "dialog" that determines how the operator communicates with the system, which functions are available, and which information is provided at each moment. It is this dialog, and the effect of users' actions on the system, that must be captured in the specification. Furthermore, this specification must be amenable to exhaustive and, preferably, automated analysis.

This last requirement rules out specifications written in natural language which are notoriously ambiguous and not readily amenable to analysis. The use of an appropriate formal language for modeling the user interfaces has the benefit of a precise mathematical meaning, allowing the elaboration of a specification that is complete, consistent, and unambiguous. It also helps ensure accurate communication between all parties involved as well as providing a certain level of precision through its execution. Computer-aided automated analyses also raise confidence in the completeness and consistency of the user interfaces.

The application of formal techniques for specification and verification to complex system's user interfaces may address different goals. These range from validating the user interface's design before its implementation, through to the system's qualification and acceptance phase, where it is important to guarantee that the user interfaces are appropriate for the system's safety. These aspects are sometimes hard to assure with conventional testing. More recently the ability to validate requirements early on in the complex systems lifecycle is becoming increasingly important to organizations that implement capability-based acquisition. Government organizations, for instance, play the role of smart buyers whose job is to acquire a set of capabilities. This makes the task of assuring that the capabilities are correctly translated into system specifications more vital to the government.

Although there is an extensive literature reporting results in the use of formal methods for modeling and verification of safety-critical space systems (Leveson et al. 1994; Schneider et al. 1998; Havelund, Lowry, and Penix 2001; Glück and Holzmann 2002; Alves et al. 2011), there is a lack of references reporting the use of such methods for formally specifying and verifying systems' user interfaces. Broadening the scope, references can be found that address user interface analysis in safety critical domains, such us avionics (Campos and Harrison 2001; Rushby 2002), or the medical field (Elder and Knight 1995; Campos and Harrison 2009; Masci et al. 2011; Harrison, Campos, and Masci 2013). However, while some of the references address real devices (or components of devices), most tend to be relatively small devices, which raises issues of scalability.

In this paper we present a structured, comprehensive and computer-aided approach to formally specify and verify user interfaces. The approach is based on model checking techniques and resorts to the IVY workbench (Campos and Harrison 2008; 2009), a model-based tool for the analysis of interactive systems. A case study of a safety-critical system used within the aerospace domain was conducted, where a set of user interfaces were formally modeled, based on the User Operation Manual, and verified. Besides of formally verifying the user interface against critical properties, the study case was intended to evaluate two main points: the proposed approach and its tool-based support for this kind of systems; and the expressiveness of the formal language of
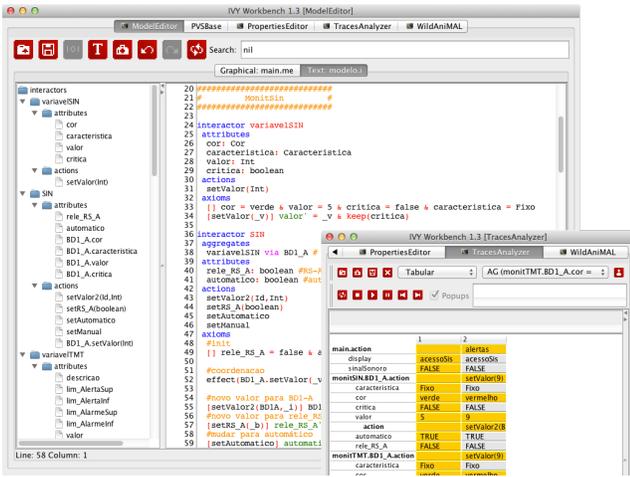
Figure 1: The IVY tool

Figure 2: State-based representation.

choice. The results to date showed the applicability of such approach for the system under study, and its potential to be used in other critical areas. The study case also signaled a potential of applying such approach for user interface verification and subsequent acceptance of a third-party contractor.

The IVY workbench

The IVY tool (see Figure 1) supports the modeling and verification of interactive systems. The approach is based on the development of models of the interactive device, and on their verification through model checking against properties that encode assumptions about the device's behavior. When verification fails, the counter examples produced by the verification process act as scenarios for analysis.

The tool provides support for three main activities: building a model; expressing and verifying properties; and analyzing the results of the verification process. Each of these will now be briefly described.

Building models   Models are developed in the MAL (Modal Action Logic) Interactors language. A detailed description of the language is out of the scope of the paper, the interested reader is referred to (Campos and Harrison 2001) for more details on the language. In brief, a MAL interactor is defined by: a set of typed **attributes** that define the interactor's state; a **mapping** of the attributes to some presentation medium; a set of **actions** that define operations on the interactor's state; a set of **axioms** written in MAL that define the semantics of the actions.

MAL axioms can be divided into three main groups: **Propositional axioms** define invariants that must always be true of the state of the interactor; **Modal axioms** define the effect of an action on the state of the interactor, they can be seen as production rules that define a state machine over the state of the interactor; **Deontic axioms** define under which conditions an action is permitted or obligatory.

A model is constructed by composing interactors in a hierarchical structure. In this way, a model can always be represented by a state mach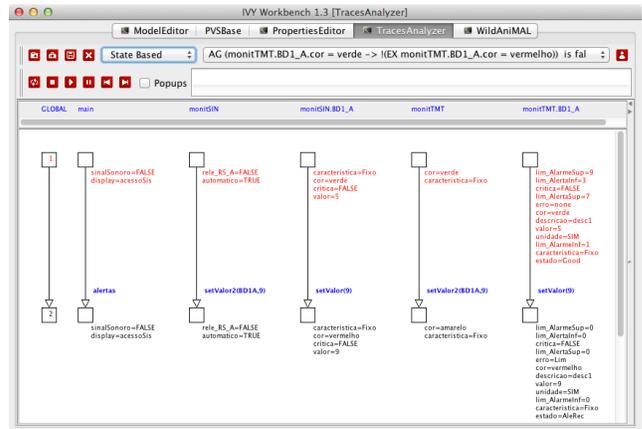ine, where the states are defined by the attribute values and the transitions are labeled by the actions that cause changes to the attributes.

Expressing and verifying properties   Properties for verification are written in CTL (Computational Tree Logic) (Clarke, Emerson, and Sistla 1986). Properties express assumptions about the expected behavior of the system.

Patterns are used to help the writing of the properties. The IVY user can select, from a number of patterns, the one that best suits the analysis needs, and instantiate it with actions and attributes from the model. The correct CTL property for verification is generated by the properties editor.

The verification step is performed by the NuSMV model checker. To make the verification possible, MAL interactor models are compiled into equivalent SMV models.

Analysing results   A visualization component facilitates analysis of the results by means of visual representations and trace analysis mechanisms. A tabular representation (see inset in Figure 1) uses columns to represent states and lines for actions and attributes (the yellow/darker background color on some cells is used to highlight changes to the values of actions/attributes).

In a state-based representation (see Figure 2) each interactor is represented in a column which shows the states that the interactor goes through. Actions are shown as labels in the arrows between two consecutive states. A further representation uses UML 2.0 Activity Diagrams.

The Testing and Preparation System (TPGS)

The TPGS is a space ground legacy system that has been in use to support space mission preparation for more than 15 years in the Brazilian Space Launcher Program. The software components of TPGS are often updated as a result of new mission requirements, different rocket configurations, and new operating systems releases. The hardware is occasionally upgraded as a results of the obsolescence of the TPGS's equipment. Main contractors are in charge of these activities. Our mid-term goal in this case is to provide a set of user interface requirements for correct mission-safety system operation as the system evolves as a result of these up-
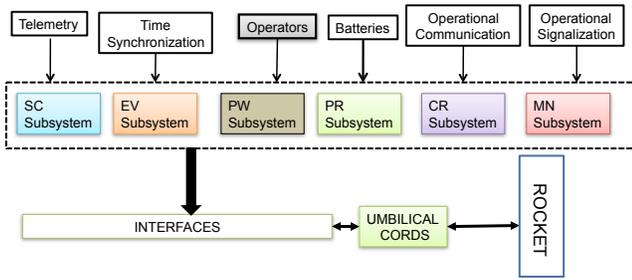
Figure 3: The TPGS's macro architecture

Figure 4: The models's macro architecture

dates. These requirements are formally modeled and verified against critical properties using the IVY tool. Once the user interface requirements are formally verified, they can be used as a requirement baseline (an oracle) for the TPGS during the acceptance phase.

TPGS is a safety-critical system that is composed of very specific and customized hardware and software components. These components are responsible for the ground control, testing and preparation of a satellite launch rocket before it is launched. It includes six different subsystems that provide specific functionality during the rocket's testing and preparation for launching. Figure 3 shows the macro architecture of TPGS.

The functional requirements for the TPGS were established separately for each subsystem in a 250-page document. The rocket's preparation process for the flight involves thoroughly checking specific parts of the rocket's electrical network. The operators of each subsystem must follow a comprehensive preparation checklist that includes procedures and interactions with the rocket's hardware and software in real-time. The preparation and testing process is mainly based on the user interface's inputs and outputs. As each TPGS's subsystem has a specific set of critical procedures to be accomplished, the user interfaces must provide an acceptable level of trustworthiness.

Considering the user interface complexity of the TPGS, we decided to select the EV subsystem to apply our approach. This subsystem is responsible for flight events, sequence testing and the activation of the rocket's security mechanical devices during the automatic sequencing for launching. We intended to address two issues: (a) determine whether the expressiveness of the language was enough to model the type of user interfaces used in the aerospace field (and study the best approach to model them); (b) determine the viability of analyzing and formally verifying user interfaces of systems as complex as these (and study the level of detail that could be achieved while maintaining the feasibility of the analysis).

Model

We do not provide a detailed account of the model herein (detailed descriptions of MAL models can be found in, for example, (Campos and Harrison 2008; 2009; Harrison, Campos, and Masci 2013)). Here we are interested in how the model was generated from the operator manual.
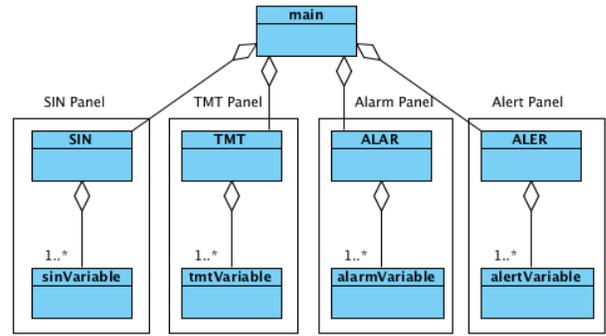
Structure of the model

The first decision to make was concerned with the structure of the model. According to the manual, the subsystem consists of several screens, with some screens containing tens of graphical elements. This structure was replicated in the model, making use of the object-oriented features of the MAL interactors language. The model becomes a tree-like composition of interactors (organized through aggregation relations), where each interactor represents an entity in the interface (e.g. a panel or a control representing the state of a variable in one of the panels).

Figure 4 shows the architecture of the model. A *main* interactor coordinates a number of other interactors, each corresponding to a different panel in the user interface. Each of these panels aggregates variables representing the information that they display. It must be noted that panels *Alarm* and *Alert* are special cases. They present a summary view of the variables in an alarm (resp., alert) condition in the other panels.

Navigation between screens

The manual describes the interfaces of the EV subsystem as comprising a total of twenty three screens. Only one screen can be displayed at a time. Which screen is being displayed is represented in the *main* Interactor using the variable *display* with type *Screens* (the set of possible screens):

```
types
  Screens = {Principal, Sinotico, alar, ...}
interactor main
  attributes
    [vis] display: Screens
```

To change between screens the user interface provides a set of buttons. Actions representing each of the buttons (hence, the selection of each screen) were defined. The actions are all defined in the same way: when an action is triggered, for example *sin*, the value of variable *display* changes to represent the appropriate screen (in the case below, the *Sinotico* screen):

```
  actions
    sin tmt alarmes alertas ...
  axioms
    [sin] display'=Sinotico
    ...
```

As some navigation buttons do not always appear on the screen, some restrictions have to be made to represent this behavior. The action *sin* for example, is not available on the access screen, when the system is blocked, and on the exit screen. To represent that, we need to add a permission statement, saying when the action *sin* can be triggered:

```
per(sin) -> !(display in {acessoSis,
    acessoSev, Bloqueio, Escolha, Final})
```

All actions related to navigating the user interface are modeled similarly. To finalize the modeling of the navigation we only need to define the first screen of the system. This is done with an initialization axiom:

```
[] display=acessoSis
```

Modeling the screens

Modeling navigation between screens was relatively straightforward, the next step was to model the screens. Since each of the screens presents information about a specific aspect of the system (particular cases are the alerts and warning screens that show lists of variables in alert/alarm), the first step was to model the behavior of a single variable. Then, modeling a screen consisted in defining the aggregation of a relevant number of variables (with any additional actions and control logic).

Variables   The manual describes variables on the screen as having a value which can change over time, a color, and a display characteristic (fixed or blinking). The value of a variable changes according to some underlying system properties (e.g. temperature of some specific device). Variables can be critical or non-critical, depending on which system property they represent. The *TMT* screen has the more complex variables, featuring additional characteristics such as working limits, description, and measurement units. Going outside the specified limits generates alerts or alarms. These alerts and alarms must then be acknowledged by the operator.

To capture all the behavior described we designed the *tmtVariable* interactor. The value displayed, and other characteristics such as the color and display characteristic used are captured by attributes of the interactor. Changing the variable is achieved through the *setValue(int)* action, where the action changes the value and sets the appropriate color and blinking characteristic. Axioms defining it take the general shape (where priming is used to reference the value of an attribute in the state after the action has happened):

```
[setValue(_v)] (conditions on _v) ->
        value' = _v & color' = ??
      & characteristic' = ??
```

To illustrate the approach taken, consider the following extract from the manual (translated from the portuguese original), which describes when a variable should be displayed in blinking yellow (i.e. *value'=yellow & characteristic'=blink*):

Blinking yellow: For a critical variable, when the current value of the variable is in non acknowledged alert (value within the alert range), there is no acknowledged alarm in the variable, and the previous criterion [non acknowledged alarm criterion] is not satisfied. If over

the same critical variable an acknowledged alarm exists, then Fixed Red prevails. For a non critical variable, when the current value of the variable is in non acknowledged alarm (value within the alarm range)

Representing the fact that a variable is critical or not by a boolean attribute named *critical*, and the inferior/superior alert and alarm limits by *infAlertLim*/*supAlertLim* and *infAlarmLim*/*supAlarmLim*, respectively, we can model the condition expressed in the first sentence of the quotation above as:

```
critical
& ( (_v>=infAlarmLim & _v<infAlertLim)
  |(_v<=supAlarmLim & _v>supAlertLim))
& alarmState!=AlaRec & alarmState!=AlaNRec)
```

where *AlaRec* is the acknowledged alarm state and *AlaNRec* is the non acknowledge alarm state.

Similarly, the last sentence can be represented by the expression:

```
!critical
& ((_v < infAlarmLim) | (_v > supAlarmLim))
```

By this means the axiom described in Figure 5 was developed. The axioms include two additional attributes: the type of error, and the variable's unit. The *keep* operator is used to express that the value of the listed attributes are kept unchanged. The value of an attribute will change nondeterministically, unless this is ruled out by the *keep* operator or the value is explicitly set by the model axiom.

Note that the middle sentence was not directly included in this axiom. Instead, because the sentence refers to the prevalence of a different condition over this one, a new axiom (for the condition *critical & alarmState = AlaRec*) is added to the model. In this case the axiom specifies setting the color to red and the blinking characteristic to fixed.

Screens   A model of the TMT screen can now be created by aggregating instances of *tmtVariable*. The TMT screen has a table with more than 30 variables. Its function is to allow monitoring the variables, and as such, there are no actions for the user to perform in the screen. However, the access button to the screen (as for all the other screens in the system) has a color and a blinking characteristic, as the variables do. These are represented directly on the panel. Here is the structure of the Interactor:

```
interactor TMT
 aggregates
  tmtVariable via BD1_A
  tmtVariable via BD1_B
  ...
 attributes
  characteristic: Characteristic
  color: Color
 actions
  setValue2(Id,Int)
```

The means of defining *color* and *characteristic* is similar to that of the variables, but in this case their values depend on the colors and characteristics of the variables in the screen. For example, the button is red and blinking when at least one critical variable of the screen is in a non-acknowledged alarm state. These constraints are expressed as invariants. In the case of two variables we can write the following invariant to express the blinking red condition above:

```
[setValue(_v)] (  (  (critical & (  (_v >= infAlarmLim & _v < infAlertLim)
                                  | (_v <= supAlarmLim & _v > supAlertLim)))
                  | (!critical & ((_v < infAlarmLim) | (_v > supAlarmLim))))
            & (alarmState != AlaRec & alarmState != AlaNRec))
        ->  value = _v & color = yellow & error = Lim & alertState = AleNRec
          & characteristic = Blink
          & keep(supAlertLim,infAlertLim,supAlarmLim,infAlarmLim,
                                        unit,critical,alarmState)
```

Figure 5: Axiom for the blinking yellow case

```
BD1_A.alarmState = AlaNRec -> (color = red
      & characteristic = blink)
BD1_B.alarmState = AlaNRec -> (color = red
      & characteristic = blink)
```

In the case of the *setValue2* action, we need to synchronize it with *BD1_A.setValue* and *BD1_B.setValue*. We can do this by adding constraints to express, indeed force the requirement that when a variable sets its value the screen performs the same action (the effect operator asserts the occurrence of an action):

```
effect(BD1_A.setValue(_v)) ->
                effect(setValue2(BD1A,_v))
effect(BD1_B.setValue(_v)) ->
                effect(setValue2(BD1B,_v))
per(setValue2(BD1A,_v)) ->
                effect(BD1_A.setValue(_v))
per(setValue2(BD1B,_v)) ->
                effect(BD1_B.setValue(_v))
```

The first two implications state that when the *BD1_A* (resp., *BD1_B*) variable sets its value the setting of the variable is visible on the screen too (i.e., *setValue2* happens when any of the variables are set). The third and fourth implications state that *setValue2* for *BD1_A* (resp., *BD1_B*) is only permitted when *BD1_A* (*BD1_B*) is set (i.e., *setValue2* cannot happen unless the set of a variable happens).

For the sake of simplicity the axioms above are for a screen with only two variables. Adding more variables amounts to adding more axioms, or adding conditions to the existing ones. The modeling principle, however, is the same. Note also, that in the final version of the model some of these axioms were combined to simplify the model.

Adding more screens Other screens were modeled similarly, each with its specificities. The SIN screen, besides variables, features safety relays. An indicator in each relay indicates whether it can be released or not. These relays were modeled as booleans directly in the *SIN* Interactor. Additionally, the SIN screen can be set to automatic mode.

The alarms and alerts screens have a list of alarms and errors, respectively, that can be acknowledged by the operator. The same approach used for the *TMT* screen was followed: defining alerts or alarms variables, and aggregating them to create the screens' models (the *ALER* and *ALAR* interactors.

Coordination between the screens We create the model of the subsystem by aggregating the four interactors in the *main* interactor, and adding coordination invariants. Note that since the Alarm and Alert screens/interactor support the acknowledgment of alarms/alerts, and these must be coordinated with the TMT and SIN screens, an acknowledge action was added to the *TMT* and *SIN* interactors.

Analysis

A number of different properties were checked of the model. Our experience shows that these typically fall into two categories. Properties intended to validate the model (i.e., to provide confidence that no modelling mistakes have been made), and properties intended to verify the model against requirements. The first category of properties typically takes the shape of stating that some goal of the system cannot be achieved. For example, in order to get counterexamples, showing that variables can be in alert, we wrote the inverse property. In this case that for all states where the variable *BD1_A* is in normal state (its color is green), there is no next state where the variable is in alarm (color red):

```
AG(monitTMT.BD1_A.colour = green ->
        !EX (monitTMT.BD1_A.colour = red))
```

As expected, the property was model checked as false, and a counterexample was generated (see Figure 2). The counterexample highlighted a situation where the *BD1_A* variable was red, but under an acknowledged alert condition. According to our understanding of what had been modeled, such a situation should not be possible. Analysing the model we found that it did not define what happens to a non critical alert, which was the case in the counterexample. Because of that, the behavior of the system under that particular condition was not being defined in the model. Consequently the model checker was considering any possible behavior as legal, which meant that variables were in practice being changed randomly.

Initially we assumed this was a problem with the model, but analyzing the operation manual we realized that it does not provide the absent information. Hence we had found a gap in the definition of the alerts and alarms in the operation manual of the system. After discussing this with an operator, it was concluded that indeed the manual did not treat these conditions accurately. No other error was found in relation to the other properties we have tested to date.

An additional concern is related to the performance of the verification. We experimented with the use of invariants *vs.* guards in the definition of how button colors and blinking characteristics change as the variables' values change. This involved describing the relation between the variables' value and their color and other characteristics, statically versus expressing this relation in the modal axioms as described above. We found that invariants make the model checker

consume more memory during the verification, as well as taking more time to complete the analysis. Hence, we decided that, particularly in relation to the action *setValue* of the *tmtVariable* Interactor, we would use guards instead.

In this paper we have described the simpler versions of the models (considering very few variables). In the complete study we added more detail to the model to show that this is just a case of adding more variables and the corresponding axioms (models were produced with up to 42 variables per screen). The described approach adds no particular complexity in terms of writing the models (even if some axioms can become rather large). However, the addition of new variables substantially increases verification time. For example, with one variable, one property took approximately one to two minutes. With two variables the same property took approximately six minutes. With more variables the time needed for verification grows quickly. The increasing verification time is not constant, so we can expect a considerable time variation for the verification of the complete system.

A particularly interesting aspect of this work relates to the nature of the model and how it was developed. As already stated the model was developed from the operator manual, while interaction with the IAE's team was kept to a minimum. The model then represents the information provided to operators, more than the actual implementation. Proving properties can then be understood as investigating the quality of the provided information. Indeed we were able to identify an instance where not enough information is provided. This illustrates how the approach can be used independently from the "client", to obtain relevant verification results based on the information provided.

Conclusions

Formal verification techniques have an important role in guaranteeing the safety of safety-critical interactive systems. The model checking based approaches used in this paper have the rigor, and the tool support needed to perform exhaustive and automatic verification of system designs.

This paper has described how the IVY workbench was applied to an existing aerospace system. Our experience has shown that the proposed approach provides a practical and feasible way to systematically specify and automatically verify, the user interfaces of complex systems. In most of the cases, such verification activities are restricted to inspections of documents and test results analysis, where the interpretation of the results can still be quite subjective and the test scenarios may not cover all the possible combinations of actions that can take place. The support of a computer-aided tool was fundamental to accomplish this activity and the overall approach can represent a step forward to improve the system's dependability.

References

Alves, M. C. B.; Drusinsky, D.; Michael, J. B.; and Shing, M. 2011. Formal validation and verification of space flight software using statechart-assertions and runtime execution monitoring. In *Proc. 6th IEEE Intl. Systems of Systems Conf.*, 155–160.

Campos, J. C., and Harrison, M. D. 2001. Model checking interactor specifications. *Autom. Softw. Eng.* 8(3-4):275–310.

Campos, J. C., and Harrison, M. D. 2008. Systematic analysis of control panel interfaces using formal tools. In *Interactive Systems*, volume 5136 of *Lecture Notes in Computer Science*, 72–85. Springer-Verlag.

Campos, J. C., and Harrison, M. D. 2009. Interaction engineering using the IVY tool. In *Engineering Interactive Computing Systems (EICS 2009)*, 35–44. ACM.

Clarke, E. M.; Emerson, E. A.; and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8(2):244–263.

Elder, M. C., and Knight, J. C. 1995. Specifying user interfaces for safety-critical medical systems. In *2nd Annual Intl. Symp. Medical Robotics and Computer Assisted Surgery*, 148–155. Wiley-Liss.

Glück, P. R., and Holzmann, G. J. 2002. Using spin model checking for flight software verification. In *Proc. IEEE Aerospace Conference*, 105–113.

Harrison, M.; Campos, J.; and Masci, P. 2013. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering*. doi:10.1007/s11334-013-0201-3.

Havelund, K.; Lowry, M.; and Penix, J. 2001. Formal analysis of a space craft controller using spin. *IEEE Transactions on Software Engineering* 27(8):749–765.

Leveson, N. G.; Heimdahl, M. P. E.; Hildreth, H.; and Reese, J. D. 1994. Requirements specification for process-control systems. *IEEE Trans. Soft. Eng.* 20(9):684–706.

Leveson, N. 1995. *Safeware: System Safety and Computers.* Addison-Wesley Publishing Company, Inc.

MacKenzie, D. 1994. Computer-related accidental death: an empirical exploration. *Sci. Publ. Policy* 21(4):233–248.

Masci, P.; Ruksenas, R.; Oladimeji, P.; Cauchi, A.; Gimblett, A.; Li, Y.; Curzon, P.; and Thimbleby, H. 2011. On formalising interactive number entry on infusion pumps. *Electronic Communications of the EASST* 45.

Nichols, D. 1973. *Mishap Analysis: an Improved Approach to Air Craft Accident Prevention.* Air War College research report. Air War College, Air University.

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

Schneider, F.; Easterbrook, S. M.; Callahan, J. R.; and Holzmann, G. J. 1998. Validating requirements for fault tolerant systems using model checking. In *Proc. 3rd Interntional Conference on Requirements Engineering*, 4–13.