Formal Verification of Medical Device User Interfaces Using PVS

Paolo Masci[1,⋆], Yi Zhang[2], Paul Jones[2],
Paul Curzon[1], and Harold Thimbleby[3]

[1]School of Electronic Engineering and Computer Science
Queen Mary University of London, United Kingdom
{paolo.masci,pc}@eecs.qmul.ac.uk

[2]Center for Device and Radiological Health,
U.S. Food and Drug Administration, Silver Spring, Maryland, USA
{yi.zhang2,paul.jones}@fda.hhs.gov

[3]FIT Lab, Future Interaction Technology Laboratory
Swansea University, United Kingdom
harold@thimbleby.net

Abstract. We present a formal verification approach for detecting design issues related to user interaction, with a focus on user interface of medical devices. The approach makes a novel use of configuration diagrams proposed by Rushby to formally verify important human factors properties of user interface implementation. In particular, it first translates the software implementation of user interface into an equivalent formal specification, from which a behavioral model is constructed using theorem proving; human factors properties are then verified against the behavioral model; lastly, a comprehensive set of test inputs are produced by exploring the behavioral model, which can be used to challenge the real interface implementation and to ensure that the issues detected in the behavior model do apply to the implementation.
We have prototyped the approach based on the PVS proof system, and applied it to analyze the user interface of a real medical device. The analysis detected several interaction design issues in the device, which may potentially lead to severe consequences.

Keywords: Software verification; Medical devices; User interfaces.

1 Introduction

In many countries, manufacturers of medical devices are required to assure reasonable safety and effectiveness of software in their devices; they have to provide adequate evidence to support this before their device can be placed on the market [1]. When considering the safety of a medical device, human factors issues

⋆ Corresponding author.

that include the human-device interface are critical. We refer to the part of a device that the user receives information from and provides information to as *the user interface*. Software in the device that contributes to the behavior of this interface we refer to as *user interface software*. User interface software defines the way in which a device supports user actions (e.g., the effect of clicking a $\boxed{\text{Start}}$ button) and provides feedback (e.g., rendering error messages on the device's display) in response to events.

The development of user interface software, or more generally, the interaction design of medical devices, is not standardized in the industry. Instead, each device manufacturer crafts its own device interaction design. A number of reports (such as [27]) have asserted that manufacturers typically address human factors issues within their user interface software in an ad hoc manner, rather than using rigorous design and evaluation techniques. Part of the reason lies in the fact that human factors specialists are usually involved too late in the software development process, if at all. These specialists typically base their analysis upon methods like heuristic evaluation [10], which require the availability of a fairly complete user interface prototype. As a result, it is often too late and too expensive to find and correct an interaction design flaw. Software engineers, on the other hand, do not have effective means to identify human factors related flaws in a software implementation, if such flaws are inherited from system-level design and defined in software requirements and design specifications.

The reality described above, as well as the fact that many manufactures reuse legacy code to develop new devices, makes it necessary to verify interaction design flaws after a user interface is implemented. However, dosing so can be expensive and time-consuming. It is more desirable and cost-effective if such flaws can be detected and weeded out early on (e.g. at the design stage). Rigorous development techniques, such as model-based design [13,22], can help to achieve this objective, if integrated into the development life-cycle.

In this paper, we focus on user interface software in medical devices, and present a formal approach for detecting design issues in such software. The approach translates the source-code implementation of user interface software into a formal specification. Theorem proving is then used to generate from this specification a behavioral model of the software. This model captures the control structure and behavior of the software related to handling user interactions. During this process, theorem proving is also used to prove that important human factors principles are satisfied by (all reachable states of) the model, or otherwise to detect potential interaction design issues. The behavioral model generated is also exhaustively explored to derive a suite of test input sequences that can expose the detected interaction design issues, if any, in the implementation of the user interface software.

The contributions of the paper are as follows. (i) We present a formal approach to generate and verify behavioral models of user interface software. The approach is based on a novel use of configuration diagrams [23]. (ii) We describe a case study based on a real medical infusion pump. The presented approach is demonstrated within PVS [20] for a C++ implementation of the device user interface software. Our approach was successful in detecting multiple interaction design issues from the implementation of the user interface software of the subject pump, many of which could potentially cause severe consequences.

The reason that we chose infusion pumps as a representative class of medical devices for study is because many infusion pumps suffer from poor human factors design. In fact, 87 models of infusion pumps were recalled in the US alone between 2005 and 2009. Human factors issues were among the primary causes for these recalls [6].

The present work builds on our previous research on the verification of medical device user interfaces [11, 14–16, 22] and on user interface prototyping [19]. These previous efforts have demonstrated that formal methods can be used to identify human factors issues in reverse-engineered models of medical devices. This paper presents an approach that continues our previous work, and extends rigorous analysis to source code implementations of real user interfaces.

2 Example results from formal source code analysis

To better illustrate the usefulness of our approach, we first explain the results of applying it to analyze the user interface implementation of a real infusion pump. In this case study, the details of which are introduced in section 4, our approach detected four interaction issues listed below. These issues cause the pump to either overlook user errors or interpret input numbers in an erroneous way. In either situation, unexpected numbers may be used to configure the pump, which can potentially cause serious clinical consequences (e.g., a lethal dose of drug is infused to the patient, because the amount of drug to be infused is mistakenly configured as an extremely large number).

Valid input key sequences are incorrectly registered without the user's awareness. The pump mistakenly discards the decimal point in input key sequences for fractional numbers between [100.1, 1200). For example, the input key sequence $\boxed{1}\boxed{0}\boxed{0}\boxed{\bullet}\boxed{1}$ is registered as 1001 without any warning or error message. This issue arises because of a constraint imposed in a routine of the pump's software: numbers above or equal to 100 cannot have a fractional part. Due to this constraint, the pump erroneously ignores the decimal point in the key sequence $\boxed{1}\boxed{0}\boxed{0}\boxed{\bullet}\boxed{1}$, and registers it as 1001. This issue opens the possibility that a user commits a missing decimal point error and accidentally inputs a value ten times larger than the intended one (an out-by-ten error).

Inappropriate feedback is given to the user for error conditions. The pump produces an inappropriate error message for fractional numbers between [120.1, 1200). For example, the pump rejects the input key sequence $\boxed{2}\boxed{0}\boxed{0}$ $\boxed{\bullet}\boxed{1}$ with the error message *"HIGH"* even if the range of accepted values is (0, 1200]. The reason for this issue is because the pump erroneously ignores the decimal point in the key sequence and registers the number as 2001, which is beyond the permitted range. What the pump should have reported is a message like *"The input value 200.1 should not have a fractional part"*. Even though the pump rejects the key sequence for $\boxed{2}\boxed{0}\boxed{0}\boxed{\bullet}\boxed{1}$, it accepts key sequences for integers on either side of 200.1. Without appropriate feedback, the user might not understand why keying a number within the range limits supported by the device is rejected, and could erroneously reach the conclusion that the device is malfunctioning.

Ill-formed input key sequences are silently accepted without the user's awareness. For instance, the sequence $\boxed{9}\boxed{\bullet}\boxed{9}\boxed{\bullet}\boxed{1}$ is accepted and registered as 9.91 with the second decimal point silently discarded. This invalid input sequence might be the result of a user error in reality. For example, the user intends to input the value of 99.1, but due to issues like inattention, he/she presses an unnecessary $\boxed{\bullet}$ between two $\boxed{9}$ keys. Accepting such invalid key sequences could allow user errors to go undetected. The safe and correct way of handling such invalid sequences is to halt user interaction and return a warning message.

Digits after decimal point silently discarded without the user's awareness. For instance, the pump mistakenly registers the input key sequence $\boxed{1}\boxed{0}\boxed{\bullet}\boxed{0}\boxed{9}$ as 10, as opposed to the intended 10.09. The reason for this issue is because the pump software automatically limits the accuracy of numbers to one decimal digit for values between [10, 100).

Notably, we used input sequences like the above to challenge another infusion pump from a different manufacturer. Similar design issues were observed for the same input sequences. This suggests that such design flaws may be common to different implementations of user interface software. Therefore, fixing defects presented in this paper can result in significant improvement in the safety of infusion pumps [29], and possibly other devices that incorporate interactive data entry software (such as ventilators and radiation therapy systems).

3 The approach

Our approach, as depicted in figure 1, starts with translating the source code of user interface software of medical devices into a formal specification acceptable to the PVS theorem prover. A behavioral model is then extracted, in a mechanized manner, from the formal specification using PVS and configuration diagrams. Theorem proving is also applied to the behavioral model to verify its compliance to human factor design principles. Lastly, the behavioral model is exhaustively explored to generate a suite of test key sequences that expose interaction design issues of the original device.

3.1 From C++ code to PVS specifications

PVS is a well known industrial-level theorem prover that enables mechanized verification of potentially infinite-state systems. It is based on a typed higher-order logic, and its specification language has many features similar to those of C++. These similarities between the two languages make it possible to devise a set of guidelines for translating (a subset of) C++ programs into PVS specifications, with the semantics of the original C++ programs preserved.

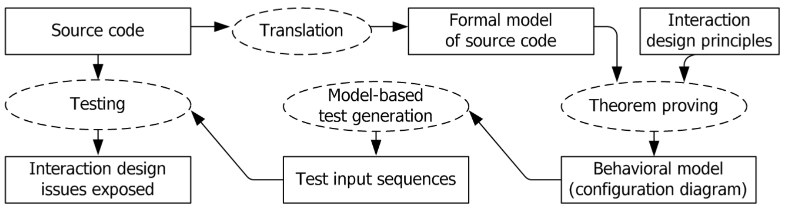
Source code
Translation
Formal model of source code
Interaction design principles
Testing
Model-based test generation
Theorem proving
Interaction design issues exposed
Test input sequences
Behavioral model (configuration diagram)

Fig. 1: Overview of our approach for verifying user interface software

Our approach adopts the following guidelines to manually translate C++ programs into PVS specifications. These guidelines provide a systematic approach for the translation:

– Conditional and iterative statements in C++ are straightforwardly translated to their counterparts in the PVS specification language;
– Computation in C++, which is typically defined as instructions modifying the values of variables of objects, is emulated in PVS with the assistance of a record type, namely `state`. In type `state`, each field is defined to record the value of a member variable in C++. Thus, computation over C++ variables can be translated as updating the fields of `state` accordingly. Type `state` is then passed to all PVS functions for reference and update;
– C++ functions are emulated in PVS as higher-order functions with the same function arguments, while local variables in C++ functions are emulated using the PVS `LET-IN` construct that binds expressions to local names;
– Class inheritance in C++ is translated by introducing a field in the structure that translates (the state variables of) the base class.

Data types in C++, such as float and integer, can be mimicked in PVS using subtyping [25], a PVS language mechanism that restricts the data domain of types. For instance, the subtype {`x: real | x >= FLOAT_MIN AND x <= FLOAT_MAX`} checks if a real-typed variable has value within the range from `FLOAT_MIN` to `FLOAT_MAX`. In many cases, subtyping is sufficient to check whether a behavioral model correctly captures all boundary conditions encountered by the C++ implementation. Furthermore, PVS includes a standard library that emulates C++ data types such as lists and strings, as well as common C++ library functions such as `strcmp`.

It is worth pointing out that, the translation of C++ programs benefits from the strong type-checking mechanism in PVS. That is, if data types declared in the PVS specifications are consistent with those in the C++ code, PVS can assist in detecting type errors in the C++ code. With appropriate subtypes, it is also possible to conduct more sophisticated type checking using PVS to detect common coding errors in C++, such as null pointer dereferences, use-before-def errors, and out-of-bound array accesses.

Currently, the translation of C++ programs in our approach considers only basic C++ constructs. The translation of complex C++ features, such as passing function parameters by reference, is left for future work, as it is not needed for our case study.

3.2 Generation of behavioral models from PVS specifications

Safe user interface design for medical devices needs to comply with important human factors principles, such as consistency of actions, feedback, mode clarity, and ability to undo. As shown in [12], such principles can be formalized as properties that must always be satisfied by a device. Our approach formalizes such principles as invariants that the behavioral model of user interface software in medical devices must satisfy.

We use in a novel way configuration diagrams, first proposed in [23], to extract the behavioral model from the PVS specification of user interface software, and to prove invariants of interest against the model. The intuition of configuration diagrams is that, proving an invariant G can be facilitated by using a strengthening invariant A, where A is given as a disjunction of properties $A = A_1 \vee \cdots \vee A_k$. Then, instead of proving G, the proof is done on $G \wedge A$, or, equivalently, $(G \wedge A_1) \vee \cdots \vee (G \wedge A_k)$. Properties A_i need not to be invariants, which makes them easier to define. Sub-properties $C_i = G \wedge A_i$ are referred to as *configurations.*

All configurations encountered during the analysis can be organized as a configuration diagram, which is a labeled graph where each node corresponds to a configuration, each edge represents a possible transition between configurations, and the labels marked on the edges denote conditions that enable transitions.

Our approach follows the following mechanized process, also presented in [23], to construct configuration diagrams for PVS specifications:

1. Invent a configuration C_1; Use the theorem prover to verify that C_1 is reachable from the initial state and C_1 satisfies the property being verified.
2. Identify the conditions that trigger outgoing transitions from C_1, and use the theorem prover to check if the disjunction of these conditions is *true.* This ensures that all possible cases are covered.
3. For each condition identified in (2), use the theorem prover to perform a symbolic execution for one step from C_1. This returns a new configuration C_2, an already existing configuration, or a variant of an existing one. If new configurations are obtained, check them against the property being verified.
4. Repeat steps (2) and (3) until no new configuration is encountered.

An example of using configuration diagrams to extract and verify behavioral models can be found in sub-sections 4.3 and 4.4.

3.3 Generation of test input sequences

In many modern medical devices user interaction is carried out by clicking buttons. Test cases to (the user interface of) these devices can therefore be given in the form of a sequence of key presses that the user performs to operate the devices. The effectiveness of using input key sequences to analyze the user interface of medical devices has been demonstrated in [5], where key sequences reflecting arbitrary user strategies were generated to assess the sensitivity of infusion pumps to unnoticed key slip errors.

In our approach, however, key sequences are generated from configuration diagrams, and used as test cases to challenge the real implementation of user interface software. That is, an analyst can watch the execution of the implementation based on the generated key sequences, so as to confirm whether or not it actually possesses the design issues detected in its behavioral model.

To generate key sequences from a configuration diagram, our approach traverses the diagram and identifies user actions associated with its transitions. Formally, a *walk* in a configuration diagram is a sequence $n_0 \xrightarrow{e_{01}} n_1 \xrightarrow{e_{12}} n_2 \ldots$, where n_i is a node in the diagram, and e_{ij} is an edge connecting node n_i to n_j. By collecting user actions (key presses in our case) marked on each edge e_{ij} in a walk, one can produce a sequence of key presses that can be used as a test case.

3.4 Discussion

Most of the model construction and proof tasks in our approach are automated by PVS and *grind*, a powerful decision procedure included in PVS, which repeatedly applies definition expansion, propositional simplification, and decision support to assist the analysis [26]. Human intervention is required only for two purposes: 1) guide PVS to prune irrelevant details away from the analysis, in order to avoid case-explosion and keep the generated configuration diagram compact; and 2) guide PVS to decompose theorems into sub-theorems. More specifically, the analyst needs to select or modify control conditions of the behavioral model suggested by PVS. PVS then checks if the selected or modified ones cover all possible model execution paths.

It should be noted that, even though human intervention demands skills and expertise with PVS, the level of human involvement required by our approach does promote active thinking for the analyst, giving her/him deep insights into the software's control structure and behavior. Because of this active involvement, it is possible to identify (the root cause of) issues and their fixes before the analysis is complete [23].

Lastly, the key point of generating useful key sequences, as in traditional software test generation, is to ensure that the key sequences derived from the configuration diagram achieve full coverage of the diagram. This ensures that the generated key sequences represent all possible user interactions that user interface software may encounter. Our approach currently realizes the generation of test sequences based on manual browsing of configuration diagrams. But it can certainly be extended with effective model based test generation techniques (e.g. [28]), to automate the exploration of (large-scale) configuration diagrams and the generation of comprehensive test key sequences from them.

4 Case study: analyzing a real-world infusion pump

To evaluate the effectiveness of our approach, we applied it to the user interface implementation of a real infusion pump[1]. It should be noted that, in the study

[1] The identity of the pump is concealed for confidentiality reasons, even though it is no longer marketed in US. Also, the information presented in this section is obfuscated.

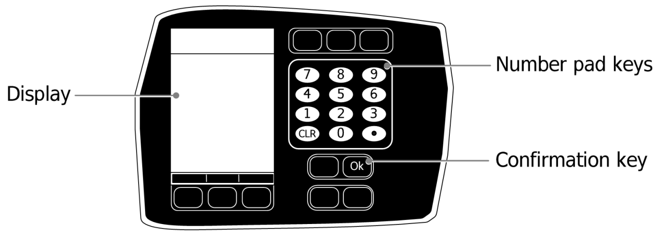

Fig. 2: Layout of the infusion pump user interface under study

we had access to the source code of the user interface software, but we did not have access to the design documentation of the pump, nor the library objects its implementation referenced. Admittedly, the absence of library code may cause inaccuracy of verification (e.g., design issues are falsely detected or omitted). Fortunately, the design issues detected in this study, as reported in section 2, were confirmed as genuine and caused by the subject implementation.

4.1 Overview of the user interface under study

Figure 2 illustrates the general layout of the user interface considered in the study. Keys relating to the data entry system are labeled, while the others are left blank for simplicity. By understanding the pump implementation, we comprehended its behavior, which is summarized as follows.

Digit keys. During data entry, the software accepts one key press at a time and calculates new values to be rendered on the display according to the following rules: (i) if a decimal point key has not been registered, then the new value is obtained by adding ten times the current displayed value and the value associated with the digit key clicked. For instance, if the display is 1 and a click on $\boxed{7}$ is registered, then the new value is $10 \times 1 + 7 = 17$; (ii) if a decimal point key has been registered, the value is obtained by adding the current displayed value and the value associated with the clicked digit times $10^{-(\text{decimalDigits } + \text{ } 1)}$, where decimalDigits is the current number of decimals of the displayed value. Thus if the display is 17. and a click on $\boxed{2}$ is registered, the new value is $17 + 2 \times 10^{-1} = 17.2$; (iii) the display is updated to the calculated value only if:

– The new value is in the range 0–1200;
– The maximum decimal precision of the new value does not exceed
  - 2 decimal digits if the new value is less than 10; or
  - 1 decimal digit if the new value is within [10, 100); or
  - 0 decimal digits if the new value is equal to or greater than 100.

A key that causes the calculated value to violate the above constraints puts the software into an error mode, in which user interaction is halted, and a warning message is displayed.

Decimal point key. The pump registers decimal points only when the current displayed value is less than 100 and a decimal point has not been previously registered. Otherwise, the decimal point key click is discarded.

Clear key. If the software is not in the error mode, the initial state is restored (i.e., the displayed value is reset to 0); otherwise, the error mode is cleared and the most recent valid state is restored.

4.2 Translation of the C++ implementation

The portion of the implementation under study was a C++ class, the body of which consists of approximately 2,000 lines of code. This class defines the pump's behavior of handling key presses on the number pad, and managing feedback rendered on its display.

The first step of analysis was to translate the C++ class into PVS specifications, in which the guidelines given in section 3.2 were followed.

Listing 1.1: PVS specification of the software's state variables

```
state: TYPE = [# display: {s: string | s`length < DISP_BUFF_SIZE},
                 dispval: float,
                 pointRegistered: bool,
                 decimalDigits  : {i: int | i >= 0 AND i <= 2}
                 errorMode      : bool #]
```

State variables. A record type, *state*, is defined to correspond to (the structure of) the C++ class in the implementation. Listing 1.1 illustrates the definition of *state*, in which every field is defined for one member variable of the C++ class. In particular, the *display* field stores the string to be rendered on the display; the *dispval* field is a float number that stores the current legal value registered by the pump; *pointRegistered* is a Boolean field that indicates whether or not the decimal point has been registered; the *decimalDigits* field records the number of decimal digits of the currently registered value; and *errorMode* is a Boolean that is set to true when the software is in the error mode. The predicate subtype associated with the *display* field is used to restrict the string length, while the subtype for *decimalDigits* is to enforce constraints on the number of decimal digits. Both of these subtypes are consistent with the constraints imposed by the original code.

Listing 1.2: PVS specification of decimal point

```
pointClicked(st: state): state =
 if(NOT errorMode(st) & NOT pointRegistered(st) & dispval(st) < 100)
 then st WITH [ pointRegistered := TRUE,
                display := strcat(display(st), ".") ] else st endif
```

Decimal point. Function *pointClicked*, as shown in listing 1.2, translates the code that handles decimal point clicks. It takes the software's current state (*st*) as parameter, and updates the device's display by invoking *strcat* (a simulation of the counterpart C++ function) to concatenate the pieces to be displayed. A PVS's WITH construct is used to update two fields of *st* when it is not in the error mode; or leave *st* unchanged otherwise.

Digit keys. Function *digitClicked* translates the code that handles digit keys. The parameter *key* of type KEY_CODE specifies the identifier of the key (each key is given a unique identifier whose value corresponds to the key label). Listing 1.3 provides the definition of *digitClicked*, where a LET-IN construct is used to create local bindings to simulate local variables used in the implementation. When a digit key is clicked, the new display value is computed and stored in variable *tmp* (line 3 in Listing 1.3). If the new value meets the range and precision constraints, the display and other relevant state variables are updated with this value (lines 5-12 and 16-18); otherwise a warning message is displayed (lines 14-15). Function *sprintf* is called to reproduce the behavior of the corresponding C++ function, which outputs the string to be displayed.

Listing 1.3: PVS specification of digit keys

```
1  digitClicked(key: KEY_CODE)(st: state): state =
2  if(NOT errorMode(st)) then LET
3      tmp: double = dispval(st),
4      (tmp, st) = if(dotRegistered(st)) then
5          if(decimalDigits(st) < MAX_DECIMAL_DIGITS
6               & ((tmp < 100 & decimalDigits(st) = 0)
7                  OR (tmp < 10 & decimalDigits(st) = 1))) then LET
8             PPdecimalDigits = decimalDigits(st) + 1,
9             tmp = tmp + key * pow10(-1 * PPdecimalDigits)   IN
10            (tmp, st WITH [ decimalDigits := PPdecimalDigits ])
11         else (tmp, st) endif
12     else (tmp * 10 + key, st) endif IN
13     if(tmp > MAX_VALUE)
14     then st WITH [ errorMode := true,
15                    display := strcpy(display(st),message(TOO_HIGH))]
16     else st WITH [ dispval := tmp,
17                    display := sprintf(display(st), "%*.*f", 0,
18                          decimalDigits(st),tmp)] endif else st endif
```

Clear key. Function *clearClicked*, shown in Listing 1.4, translates the code segment that handles the Clear key clicks. When a click on the Clear key is detected and the software is not in the error mode, *clearClicked* restores the initial state. Otherwise, it clears the error by setting *errorMode* to false, and updates the display with the last legal value stored in *dispval*.

Listing 1.4: PVS specification of clear key

```
clearClicked(st: state): state =
  if(NOT errorMode(st))
  then st WITH [ dispval := 0, display := "0",
                 pointRegistered := false, decimalDigits := 0  ]
  else st WITH [ errorMode := false,
                 display := sprintf(display(st), "%*.*f", 0,
                              decimalDigits(st),dispval(st))] endif
```

4.3 Verification using configuration diagrams

The human factors principles that we attempted to verify against the pump implementation included: **consistency**, asserting that the same user actions (in this case, key clicks) should produce the same results in logically equivalent

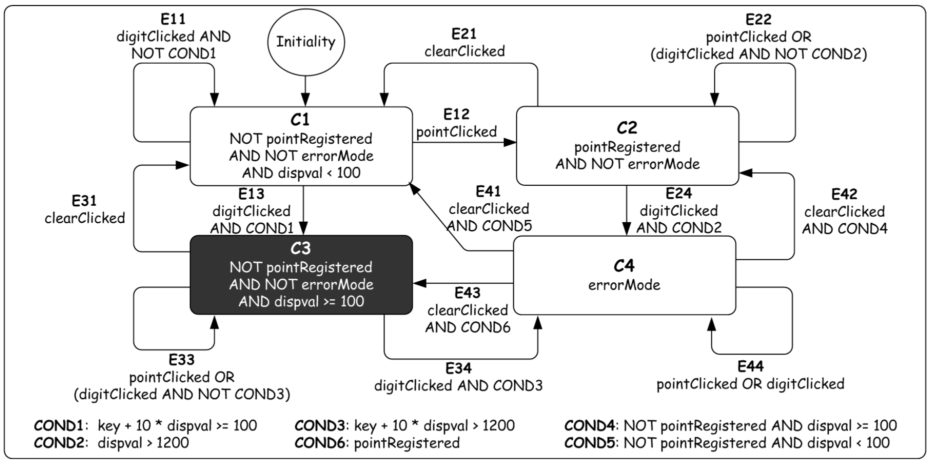

Fig. 3: Configuration diagram regarding the *consistency* of decimal point clicks

situations; and **feedback**, which ensures that the user is provided with sufficient information on what actions have been done and what result has been achieved.

Given different aspects of the pump's behavior, these two principles can be instantiated differently. Take the handling of decimal point clicks for example. We instantiated these two principles, for this specific aspect of the pump's behavior, as predicate `decimal_point_pred` (see Listing 1.5)[2]. This predicate essentially asserts that, no matter what current state (*st*) the pump has, when the decimal point key is clicked, the pump should enter into a new state *st_prime*, in which either the decimal point is registered (variable *pointRegistered* is set true), or the error mode is triggered (*errorMode* is true).

Listing 1.5: Predicate decimal_point_pred in PVS

```
decimal_point_pred(st: state): bool =
    LET st_prime = pointClicked(st)
     IN (pointRegistered(st_prime) OR errorMode(st_prime))
```

Predicate `decimal_point_pred` defines a safe way to manipulate decimal point clicks. Based on this predicate, a behavioral model was constructed for the infusion pump under study, by applying the procedure presented in section 3.2 to the PVS translation of its implementation. Simultaneously, the proof that the pump satisfies `decimal_point_pred` was accomplished within the PVS theorem prover by checking this predicate against all reachable states of the behavioral model under all possible input key sequences.

The behavioral model illustrated in figure 3, in the form of a configuration diagram, was constructed as the result of our analysis effort. After proving twenty theorems during the model construction process, we verified that the infusion

[2] Instantiation of the principles with respect to other aspects of the pump's behavior can be carried out similarly.

pump violates predicate `decimal_point_pred` (an example of such violation is shown in section 4.4). Please refer to section 2 for an explanation of the verification results, and to [17] for more details on the generation of the configuration diagram and the proof process.

4.4 Generation of test input sequences

As discussed in section 3.3, a comprehensive set of key sequences can be generated as test cases to the device implementation by exploring all *walks* in its configuration diagram.

Consider generating test cases from the configuration diagram in figure 3. At the beginning, the pump satisfies C_1: the decimal point is not registered; its user interface is not in the error mode; the display value is less than 100. This is visualized in the diagram as an edge from a default node **Initiality** to C_1.

Outgoing edges from C_1 are labeled with the combination of conditions and user actions that can lead the pump into a new configuration. Note that only conditions and user actions related to the verification of desired properties are considered. For example, only the following combinations can trigger the pump to exit from configuration C_1: a decimal point is pressed (E_{12} in figure 3); or, a digit key is pressed when COND1 holds (E_{13} in figure 3), where COND1 asserts that the new display value is greater than or equal to 100.

The trace $C_1 \xrightarrow{E_{13}} C_3 \xrightarrow{E_{33}} C_3 \xrightarrow{E_{33}} C_3$ represents a walk in this configuration diagram. This walk stands for a class of possible user interaction scenarios, one of which can be: start from C_1 when the display value is 10; key $\boxed{0}$ is pressed, and the model moves to C_3 as a digit key is pressed and COND1 is satisfied; key $\boxed{\bullet}$ is pressed, and the model stays in C_3. Lastly, key $\boxed{1}$ is pressed.

An example of sequence of key presses that can be extracted from the above example walk is $\boxed{1}\boxed{0}\boxed{0}\boxed{\bullet}\boxed{1}$, which exposes an interaction design flaw: the pump silently discards the decimal point. In particular, when the prefix $\boxed{1}\boxed{0}\boxed{0}$ $\boxed{\bullet}$ of this sequence is fed to the pump, the model will stay in configuration C_3, in which predicate *pointRegistered* is false indicating that the decimal point is not registered, and predicate *errorMode* is also false indicating that no warning message is provided to the user.

Following the above process, we generated test cases that exposed the interaction design flaws presented in section 2. These test cases were used to check the infusion pump under study, and confirmed that the detected design flaws did exist in its implementation.

5 Related work

The work presented in the paper is based on configuration diagrams, originally introduced by Rushby to verify safety properties of potentially infinite-state systems [23]. For such systems, formal verification requires either a direct proof through deductive mechanized methods (e.g., theorem proving), or justification of an abstraction that downscales the system so that it can be verified through exhaustive state exploration (using model checking for example). In contrast, our approach uses configuration diagrams in a novel way to identify interaction design issues in software. In particular, we use configuration diagrams to extract and verify a behavioral model of the software specifying how the software manages the interactions with the user.

Several approaches have been proposed to use model checking to verify user interface implementations[3]. For example, Rushby [24] used model checkers Murϕ and SAL to verify mode confusion in a cockpit; Rukšėnas et al [21] used SAL to identify post-completion errors in infusion pumps; Campos and Harrison used IVY/NuSMV to analyze infusion pumps against properties such as consistency, visibility, and feedback [4, 11]; and in our own work, we used SAL and Event-B/Rodin to analyze the data entry system of infusion pumps for their predictability [15,16] and other safety properties identified by FDA [22].

The main limitation of using model checking to analyze user interface design/implementations lies in that, one has to wisely balance the complexity of the models constructed for user interface and the fidelity of these models to the original design/implementation. On one hand, the constructed models cannot be too complex to be analyzable (within reasonable time cost) [3, 9, 12]. This is why abstraction has to be used to eliminate irrelevant details away from the models. On the other hand, it is often difficult to find appropriate types of abstraction, so as to preserve necessary details of the user interface for verification. Therefore, model checkers often use too coarse abstraction to extract models from the real design/implementation, resulting in excessive spurious counterexamples (i.e., counterexamples representing behaviors that do not exist in the real design/implementation) to be reported.

Even though counterexample guided techniques, such as [2,7,8], can be used to guide model checkers to refine and optimize the abstraction, such techniques still demand significant effort from the analysts to first decide if a counterexample is genuine or spurious. Unfortunately, with respect to analyzing user interface software for its human factors properties, no general solution has been proposed to assist analysts in making such decisions.

In contrast to model checking driven approaches, our approach defines a general method for model construction based on theorem proving and configuration diagrams. It avoids the difficulty of finding an appropriate level of abstraction that ensures the accuracy and fidelity of the constructed behavioral models. However, the behavioral models constructed by our approach can also be verified by model checkers for their human factors properties.

6 Conclusions

A rigorous and effective approach for formally verifying the source code implementation of user interface software in medical devices has been presented.

[3] It is worth noting that model-checking can be used in the design phase as a "high-level debugger" of designs. However, this requires a different approach to modeling, such as that illustrated in [22].

The case study shows that this approach can detect interaction design issues in real implementations that might lead to critical safety consequences. These issues exist because of a combination of design features in user interface software, each of which is not problematic individually. Interestingly, we fed the test cases generated by the approach to another infusion pump made by a different manufacturer, and observed similar design issues.

The case study presented only formally analyzed a portion of the software implementation of the subject infusion pump. As a result, only part of the configuration diagram was developed, and only part of the proofs generated by PVS were formally proved. However, even with this partially completed formal analysis, real issues were identified. This suggests that our approach has the potential to assess and improve the quality and safety of user interface software in medical devices even before their complete implementation is available.

Once human factors properties are assured using PVS, the specification can be used to rapidly prototype a new user interface design in which the identified interaction design issues have been addressed. In fact, PVS provides a component called PVSio-web [19] that helps developers to define the layout of a user interface; and a component called PVSio [18] that enables interactive execution of specifications defining the behavior of the user interface, and a ground evaluator that automatically compiles these specifications into executable code.

Acknowledgments. We thank Michael Harrison for his invaluable suggestions and comments on the paper and the support of CHI+MED (Computer-Human Interaction for Medical Devices, EPSRC research grant [EP/G059063/1]).

References

1. AAMI Medical Device Software Committee. Medical device software risk management. *AAMI Tech. Rep. TIR32:2004*, 2004.
2. T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS'04*, pages 388–403, 2004.
3. M.L. Bolton and E.J. Bass. Formally verifying human-automation interaction as part of a system model: Limitations and tradeoffs. *Innovations in Systems and Software Engineering*, 6(3):219–231, 2010.
4. J.C. Campos and M.D. Harrison. Modelling and analysing the interactive behaviour of an infusion pump. *Electronic Communications of the EASST*, 2011.
5. A. Cauchi, A. Gimblett, H Thimbleby, P. Curzon, and P. Masci. Safer 5-key number entry user interfaces using differential formal analysis. In *BCS-HCI*, 2012.
6. Center for Devices and Radiological Health, US Food and Drug Administration. *White Paper: Infusion Pump Improvement Initiative*, 2010.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV00*, pages 154–169, 2000.
8. M.B. Dwyer, O. Tkachuk, W. Visser, et al. Analyzing interaction orderings with model checking. In *ASE2004*, pages 154–163. IEEE Computer Society, 2004.
9. G.E. Gelman, K.M. Feigh, and J. Rushby. Example of a complementary use of model checking and agent-based simulation. In *SMC2013*. IEEE, 2013.
10. G. Ginsburg. Human factors engineering: A tool for medical device evaluation in hospital procurement decision-making. *Journal of Bio. Informatics*, 38(3), 2005.

11. M.D. Harrison, J.C. Campos, and P. Masci. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering*, pages 1–17, 2013.
12. M.D. Harrison, P. Masci, J.C. Campos, and P. Curzon. Automated theorem proving for the systematic analysis of interactive systems. In *FMIS2013*, 2013.
13. R. Jetley, S. Purushothaman Iyer, and P.L. Jones. A formal methods approach to medical device review. *Computer*, 39(4):61–67, 2006.
14. P. Masci, P. Curzon, M.D. Harrison, A. Ayoub, I. Lee, and H. Thimbleby. Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. *EICS2013. ACM Digital Library*, 2013.
15. P. Masci, R. Rukšėnas, P. Oladimeji, A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H. Thimbleby. On formalising interactive number entry on infusion pumps. *Electronic Communications of the EASST*, 45, 2011.
16. P. Masci, R. Rukšėnas, P. Oladimeji, A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H. Thimbleby. The benefits of formalising design guidelines: a case study on the predictability of drug infusion pumps. *Innovations in Systems and Software Engineering*, pages 1–21, 2013.
17. P. Masci, Y. Zhang, P. Curzon, M.D. Harrison, P. Jones, and H. Thimbleby. Verification of software for medical devices in PVS. *CHI+MED Tech. Rep., http://www.chi-med.ac.uk/researchers/bibdetail.php?docID=656*, 2013.
18. C. Munoz. Rapid prototyping in PVS. *National Institute of Aerospace, Hampton, VA, USA, Tech. Rep. NIA*, 3, 2003.
19. P. Oladimeji, P. Masci, P. Curzon, and H. Thimbleby. PVSio-web: A tool for rapid prototyping device user interfaces in PVS. In *FMIS2013*, 2013.
20. S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV96*, pages 411–414. Springer, 1996.
21. R. Rukšėnas, P. Curzon, A.E. Blandford, and J. Back. Combining human error verification and timing analysis: A case study on an infusion pump. *Formal Aspects of Computing*, in press, 2013.
22. R. Rukšėnas, P. Masci, M.D. Harrison, and P. Curzon. Developing and verifying user interface requirements for infusion pumps: A refinement approach. In *FMIS2013*, 2013.
23. J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In *CAV00*, pages 508–520. Springer, 2000.
24. J. Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering & System Safety*, 75(2):167–177, 2002.
25. N. Shankar and S. Owre. Principles and pragmatics of subtyping in PVS. In *Recent Trends in Algebraic Development Techniques*, pages 37–52. Springer, 2000.
26. N. Shankar, S. Owre, J. Rushby, and D. Stringer-Calvert. PVS prover guide. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1:11–12, 2001.
27. M.F. Story. The FDA perspective on human factors in medical device software Development. In *IQPC Software Design for Medical Devices Europe*, 2012.
28. H. Thimbleby. *Press on: Principles of Interaction Programming.* Mit Press, 2007.
29. H. Thimbleby and P. Cairns. Reducing number entry errors: solving a widespread, serious problem. *Journal of the Royal Society Interface*, 7(51):1429–1439, 2010.