The final publication is available at Springer via https://doi.org/10.1007/978-3-030-04771-9_21

Formal modelling as a component of user centred design

Michael D. Harrison[1][0000−0002−5567−9650], Paolo Masci[2][0000−0002−0667−7763], and José Creissac Campos[2][0000−0001−9163−580X]

[1] School of Computing, Newcastle University, Urban Sciences Building, Newcastle upon Tyne, UK michael.harrison@ncl.ac.uk

[2] HASLab/INESC-TEC and Dept. Informatics/University of Minho, Campus de Gualtar, Braga, Portugal
paolo.masci@inesctec.pt, jose.campos@di.uminho.pt

Abstract. User centred design approaches typically focus understanding on context and producing sketch designs. These sketches are often non functional (e.g., paper) prototypes. They provide a means of exploring candidate design possibilities using techniques such as cooperative evaluation. This paper describes a further step in the process using formal analysis techniques. The sketch design of a device is enhanced into a specification that is then analysed using formal techniques, thus providing a systematic approach to checking plausibility and consistency during early design stages. Once analysed, a further prototype is constructed using an executable form of the specification, providing the next candidate for evaluation with potential users. The technique is illustrated through an example based on a pill dispenser.

1 Introduction

User centred design approaches are designed to satisfy Gould and Lewis's guiding principles [7]: (i) to focus on user tasks early and throughout the design process; (ii) to measure usability empirically; (iii) to design and test iteratively. A variety of techniques exist that satisfy these principles to a greater or lesser extent. Contextual design [1] and scenario based design [5] are examples. Contextual design aims to understand the context through observation to identify what would help improve the situation in which the proposed design is to be used. This process involves focussing on user needs and tasks, asking questions of the following type: *"Who enters patient, medicine and prescription details in the medical device, and where do these activities happen? How and where are reminders produced and how does the patient access the dose?"*. Contextual design and scenario based design techniques use *scenarios* that capture typical or exceptional situations in which a possible design would be used.

The *sketch designs* developed as a result of this process are often initially non-functional (for example it could be a simple PowerPoint presentation or a paper storyboard). They are developed and evaluated by letting end users interact with the sketch design in selected scenarios of use. Think aloud techniques such as

cooperative evaluation [13] are typically used to collect feedback that is useful to improve the design and judge whether a further iteration would be appropriate.

The focus of this paper is *the nature of the sketch design and the process of development of the final design*. This paper briefly explores integration of the informal, though structured, approach typical of contextual design and scenario based design with formal techniques. An example based on an automated pill box for dispensing drugs to patients at specific times is used throughout the paper to present the approach.

In the following sections, first we describe how the initial sketch design was developed (Section 3.1). Then, an enhanced design is presented that fills various gaps observed of the initial design (Section 3.2). This design is checked for plausibility (Section 4) and against use-related requirements. The first set of use-related requirements are designed to check the consistency of the actions offered by the refined design (Section 5.1). Requirements also consider the reversibility of scrolling behaviour (Section 5.2). The iteration of the design (Section 6) is then described with a discussion of comparable approaches and further work (Section 7).

Contribution. Contributions are:

1. an illustration and discussion of how existing formal tools could be used as part of a user centred design process;
2. a case study using a pill dispenser design as focus.

2 The approach

The aim is to integrate the formal modelling process with user centred design. We do this through five steps.

Step 1: An initial interactive sketch design is created that demonstrates the different screens of the system, either as a storyboard or a non functional prototype. This provides the first candidate for evaluation with potential end users.

Step 2: Once evaluated, a revised design is created based on a formal model developed from the prototype. This initial formal specification includes details of modes, actions, and fields of each screen. This model is assessed for *plausibility*. The specification is plausible if it exhibits the designer's intention. This can be demonstrated through exploration of the executable form of the specification and also achieved by demonstrating that the design exhibits intended functional properties.

Step 3: The formal specification is iterated as a result of the assessment of step 2. The new version of the specification is analysed using formal verification technologies to explore inconsistencies and gaps in the proposed design.

Step 4: An executable formal specification, developed as a result of the analysis of step 3, provides the next candidate for evaluation with potential users.

Step 5: The process (steps 2-4) is repeated.

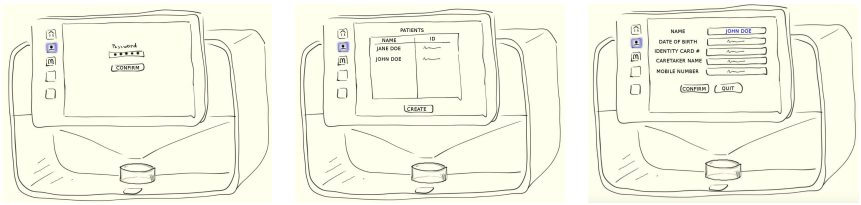
The steps of the method use the following tools provided by the PVSio-web toolkit: PVS [16] to develop and analyse the model; PVSio [15] to check its plausibility; the PVSio-web Storyboard Editor [18] to develop the initial sketch; the PVSio-web Prototype Builder [12] to produce an interactive prototype based on the PVS model. PVSio-web is a web-based environment that enables the creation of interactive prototypes based on executable PVS specifications. The toolkit supports the creation of both *storyboard-based prototypes* based on mockup pictures of different screens of the system under development, and *high-fidelity prototypes* that can closely resemble the visual appearance and behaviour of a final product. The interactive prototype, so constructed, can be evaluated with end-users. The PVS language builds on higher-order logic, and provides an extensive library of constructs for representing complex system behaviours and datatypes. PVSio is a tool that extracts Common Lisp code from PVS executable specifications. This makes it possible to test the functionality of PVS specifications by evaluating ground expressions representing user actions performed on the system state.

3 Designing a pill dispenser

A pill dispenser is a medical device that provides doses of drugs to patients at specified times. While such devices are often designed for individual use, the considered example was designed to be used by groups of patients, perhaps in a care home common room or a hospital ward. The proposed initial design suggests a device that caters for the multiple and complex requirements of patients. While this service was initially sketched by engineers, it would be expected that, as part of a user centred design, the initial design would have been informed by developing an understanding of the context in which the design is to be situated. The device as envisaged in early sketches alerts the patient when medicine is due and the patient responds and obtains their dose using a thumb print to ensure they are receiving the medication intended for them. The device maintains a database of patients who have been subscribed to the system as well as a database of medicines. The pill dispenser supports "columns" of pills from which the patient can obtain their required dose.

3.1 The starting point: the sketch design

A video was provided to the authors of an early prototype of the device. A storyboard was produced from this video (see Figure 1). The display designs were sketched and transitions between displays indicated. The initial prototype was the sketch design. A state transition diagram described the transitions between displays (see Figure 2). PVSio-web uses a graphical state transition language (emucharts a simplified version of Statecharts [9]) to describe the flow of the storyboards. The sketch screens are linked to nodes of the emuchart which can be translated automatically into PVS as illustrated in Listing 1.1.



(a) Password screen (b) Patient list screen (c) Patient details screen

Fig. 1: Example display images produced for the initial prototype.

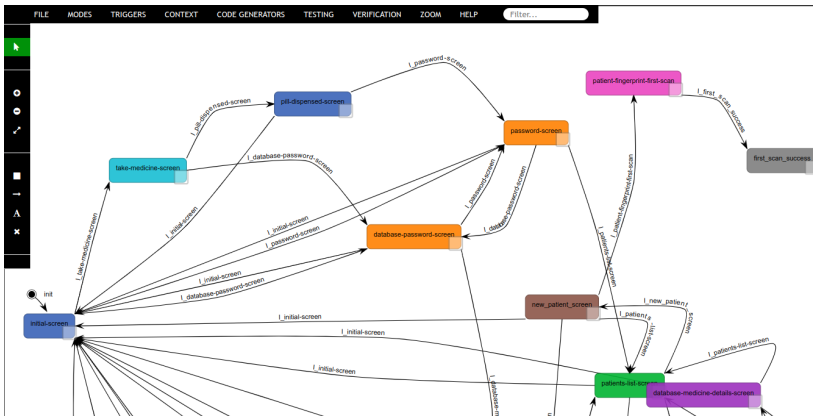


Fig. 2: The initial sketch: state transition diagram

The video illustrated some but not all of the features of the design. Figure 3 indicates a phase of the creation of the interactive storyboard in PVSio-web. Green areas on the left hand side of the sketch design represent interactive buttons that can be used to navigate to a different screen. The full list of screens used to develop the sketch design is shown at the bottom-left corner of the figure.

The initial sketch design indicated three user pathways. *The first pathway* allows entry, or modification, of patient details. This requires use of a password and involves the nurse or carer responsible for setting up patient details. New patient information can be entered, including the patient's thumb print for validation purposes, or a list of existing patients in the database is displayed which can be scrolled up or down to allow access to all patients for selection. Details of the patients can be changed. Each patient has up to five prescriptions (in this version of the prototype). A prescription can be added or removed and includes details of time and frequency of each drug prescribed. *The second pathway*, also protected by password, allows access by carers or nurses or doctors who are able to enter details of medicines. This pathway allows entry or editing of medicines. In a similar way the medicine pathway allows medicines to be listed or displayed and modified.

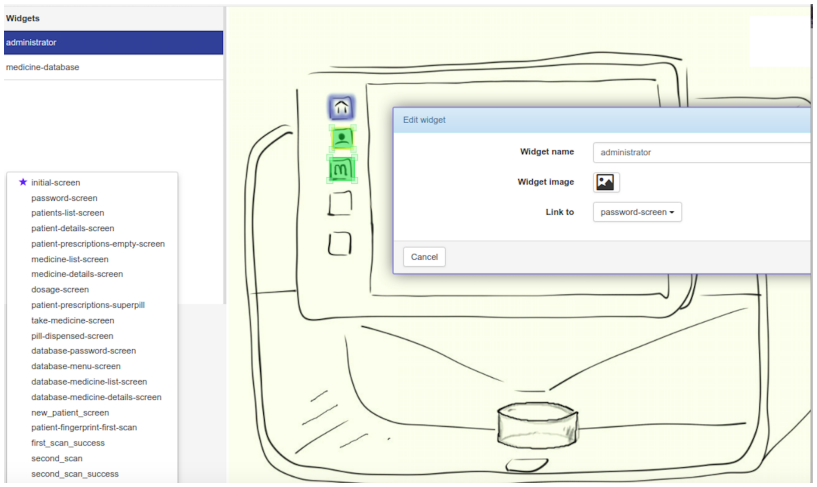


Fig. 3: Phase of the creation of the initial sketch design using PVSio-web.

```
pill_dispenser: THEORY BEGIN
  %-- operating modes
  Mode: TYPE = { initial_screen, password_screen, patients_list_screen, ...}
  %-- state attributes
  State: TYPE = [# mode: Mode #]
  %-- init function
  init: State = (# mode := initial_screen #)
  %-- transition functions
  per_password_screen(st: State): bool = (mode(st) = initial_screen)
    OR (mode(st) = pill_dispensed_screen)
    OR (mode(st) = database_password_screen)
  password_screen(st: (per_password_screen)): State =
    COND
     mode(st) = initial_screen
      -> LET st = leave(initial_screen)(st)
          IN enter(password_screen)(st),
     mode(st) = pill_dispensed_screen
      -> LET st = leave(pill_dispensed_screen)(st)
          IN enter(password_screen)(st),
     mode(st) = database_password_screen
      -> LET st = leave(database_password_screen)(st)
          IN enter(password_screen)(st)
    ENDCOND
  %-- ...more transition functions omitted
END pill_dispenser
```

Listing 1.1: PVS specification generated from the emuchart

The third pathway was not provided in the video and further information would be required to complete it. As is common in design approaches of this kind the current version of the design is partial. Further iteration will flesh out the details of the design. This pathway identifies and alerts the patient who is required to take their medicine. The patient's thumbprint is required to access the dose. The paper focuses on the two pathways that were illustrated in the original video.

3.2 The enhanced model

Additional details about modes, actions and field types present in each screen are now added to the initial model and the model is restructured. The full specification of the illustrated case study may be found at our repository[3]. This model of the design is based on actions that are invoked when the user presses a control that is visible on the display. The revised model provides more detail of the interaction: which actions are available; which fields must be completed before an action can be completed. Actions of selecting and entering fields are included as well as, in some cases, concrete examples of information in the pillbox (e.g., patient names, content of prescription charts, and so on). The pill dispenser screen is assumed to be a touch screen but for present purposes the details of how the pressing takes place is not a concern. These actions cause transitions between modes. Further transitions are caused by selecting fields and entering values. These transitions do not change mode. They add to the set of fields that have been entered and also add the values entered to temporary records of patients, their prescriptions or medicines (depending on mode). Modes, actions and fields are represented using `mode_type` (lines 1-2 of Listing 1.2), `actions_type` (line 3) and `fields_type` (line 5).The availability or visibility of actions and fields is made explicit using boolean functions, for example (in the case of actions) `available_actions_type` (line 4). The state of the device is represented by a type `state`, a fragment of the definition is illustrated in line 7-12.

```
1  mode_type: TYPE = { initial, pwd, db_pwd, db_menu, patient_list,
2                      db_med_list, new_patient_details, ... }
3  actions_type : TYPE = { key1, key2, key3, confirm, create, ... }
4  available_actions_type : TYPE = [ actions_type -> boolean ]
5  fields_type : TYPE = { password, dob, dosage, id_card, mob, carer, ... }
6  fields_set: TYPE = [ fields_type -> boolean ]
7  state: TYPE = [# mode: mode_type,
8                   vis_field: fields_set,
9                   sel_field: fields_set,
10                  ent_field: fields_set,
11                  action: available_actions_type,
12                  ... #]
```

Listing 1.2: Types used in the model of the first sketch

A PVS function is now illustrated that specifies the behaviour of the pill dispenser when the operator enters new patient details (Listing 1.3). The specification of the function includes identification of the actions that are visible (lines 5-7), the fields that are visible (lines 16-18), the fields that are selected (none in this case, see line 19) and the field that is entered (line 20). Within the mode, specified by the `mode` attribute of `state`, fields can be entered (as discussed below, and see definition of `enter` in Listing 1.5). In the initial state of this mode, described here, only one field is represented as entered, namely the patient name. For reasons of simplicity the name is taken to be generated automatically in this initial model (see definition of `np` in line 2).

Each time a field is entered the temporary patient record (`temp_patient`) is updated. In the initial transition (`new_patient_details_screen`) all fields

[3] http://hcispecs.di.uminho.pt/m/8

are empty since this is a new patient except for the patient name. The temporary patient record (`temp_patient`, lines 8-11) is also set with the patient name (`p_name`, line 8) set to `np`, and the rest of the patient record set to null. Further temporary elements are set to null: `temp_script` (which identifies prescriptions associated with the patient) and `temp_med` (which is used when setting up the record for a medicine).

```
1  new_patient_details_screen(st: state): state =
2    LET np = next_pid(st`p_max)
3     IN clear_screen(st) WITH
4        [ mode := new_patient_details,
5          action := LAMBDA(x: actions_type):
6                      (x = key1) OR (x = key2) OR (x = key3) OR
7                        (x = confirm) OR (x = quit),
8          temp_patient := (# p_name := np,
9                             p_fields :=  LAMBDA(x: fields_type): FALSE,
10                            scripts_index := s_null,
11                            scripts := LAMBDA (s: s_index): nil_script #),
12         temp_script := nil_script,
13         temp_med := nil_med,
14         m_current := m_null,
15         p_current := np,
16         vis_field := LAMBDA(x: fields_type):
17                        (x = name) OR (x = dob) OR (x = id_card) OR
18                          (x = carer) OR (x = mob),
19         sel_field := LAMBDA(x: fields_type): FALSE,
20         ent_field := LAMBDA(x: fields_type): x = name ]
```

Listing 1.3: The specification of the new patient details screen

The patient database (Listing 1.4), is specified by type `patient_db_type`. This type describes a list of patient records. Patient records include fields associated with date of birth, carer and so on as well as the prescriptions that are associated with them (scripts). There is a limit to the number of scripts that can be associated with a patient as defined by type `s_index`.

```
1 list_script_type: TYPE = [s_index -> script_type]
2 patient_type: TYPE = [# p_name : p_index,
3                         p_fields : fields_set,
4                         scripts_index: s_index,
5                         scripts: list_script_type #]
6 patient_db_type: TYPE = list[patient_type]
```

Listing 1.4: Patient database types

Patient fields can be entered in mode `new_patient_details`. Entry of a field requires two pre-conditions. The field must be visible. Hence in Listing 1.3 (lines 16-18), `name`, `dob`, `id_card`, `carer` and `mob` are fields that are visible. A field must also be selected (only one field is selected at a time and selection is lost when the field is entered). Hence in Listing 1.3 line 19, no fields are selected. Two actions `select` and `enter` specify selection and entry of fields. Selection also specifies selection of actions.

The function `enter` is illustrated in Listing 1.5. Entering a field first (Listing 1.5) checks that the field is selected (line 2 of Listing 1.5). It then updates temporary database fields. These are: `temp_script` (lines 4-6), `temp_patient` (lines 7-9) and `temp_med` (lines 10-12). These updates depend on whether the mode relates to entry of values to these temporary records. In all cases the

entered field is added to the set of entered fields (line 13), and the selection of the field necessary prior to entry is set to false (line 14).

```
1  enter(f: fields_type, st: state): state =
2        IF sel_field(st)(f)
3        THEN st WITH [
4              temp_script := IF per_enter_patient_script(f, st)
5                              THEN enter_script_field(f, st`temp_script)
6                              ELSE st`temp_script ENDIF,
7              temp_patient := IF per_enter_patient_field(f,st)
8                               THEN enter_patient_field(f, st`temp_patient)
9                               ELSE st`temp_patient ENDIF,
10             temp_med := IF per_enter_med_field(f, st)
11                         THEN enter_med_field(f, st`temp_med)
12                         ELSE st`temp_med ENDIF,
13             ent_field := LAMBDA(x: fields_type): x = f OR st`ent_field(x),
14             sel_field := LAMBDA(x: fields_type): FALSE ]
15       ELSE st ENDIF
```

Listing 1.5: Entering a field

4 Plausibility

Once the specification has been developed, and before further analysis of the implications of the design, it is clearly necessary to be assured that the model is a plausible reflection of the envisaged design. This checking process is iterative. The design is developed by fleshing out interaction detail and adding functionality. It is not conventional formal refinement because at each step the design is in flux, open to change as a result of evaluation and discussion with potential users. The plausibility of the specification of the design is explored in two ways. First, PVSio is used to explore grounded versions of the specified functions. This allows a form of direct interaction with the model to exercise the available actions and observe their effect on the state of the system. It makes it possible to explore some situations in which actions do not have the expected behaviour. Inevitably this second approach, using PVSio, does not allow exhaustive analysis in the sense that model checking (see for example [2]) does. The goal at this stage however is to establish a first impression about the model and flush out any obvious problems, before more exhaustive analysis is carried out. Second, PVS theorems are constructed to demonstrate that actions change state as expected. Here the aim is to demonstrate that for all states (not just the states generated through execution of the ground functions), the behaviour of actions is as expected.

The use of PVSio, to explore plausibility is now considered in more detail.

4.1 Using *PVSio* to check plausibility

PVSio [15] makes it possible to *test* the model. Although the model is of a half-formed sketch design testing can be sufficient to check that the model meets the designer's intentions. An example of how PVSio can be used to check plausibility now follows. The following shows the last steps of a sequence that builds a database of patients. The sequence shows the last few actions of a much longer

sequence including selecting (i.e., pressing) key2, selecting the password field, entering the password, and then pressing confirm. The sequence that produces the state `editmdpnp2` constructs the elements of the database. In fact the analysis of the specification involved checking that each step of the sequence had the desired effect.

```
susdmdnp2minus: state = LET st = editmdnp2,
                            st = select(key2, st),
                            st = select(password, st),
                            st = enter(password, st)
                        IN act(confirm, st)
```

PVSio shows the effect of this long sequence on the state of the pill dispenser (see Listing 1.6). There is only space to show a small part of the state that is produced. At the end of the action sequence the mode is `patient list` (line 1 in Listing 1.6). This mode shows a list of up to five (five is the limit for the screen) patients. There are no visible fields associated with this mode, but there are visible actions: key1, key2, key3 and create (line 3). The state attribute `patient_id_line` shows the list of patients (identified by `p_name`) that are visible in the list (line 4).

The listing also shows one element of the patient database (`patients_db`) with `p_name` equal to 1 (lines 5-24). This patient entry shows that fields `dob`, `id_card`, `mob` and `carer` have been entered as well as the prescriptions that have been entered. The patient entry allows for five prescriptions. Only elements l(0) and l(1) have been entered.

```
1  (# mode := patient_list,
2     vis_field := {  }, sel_field := {  }, ent_field := {  },
3     action := { key1, key2, key3, create },
4     patient_id_line := { l(4):=5, l(3):=4, l(2):=3, l(1):=2, l(0):=1 },
5     patients_db := (:
6      (# p_name := 1,
7         p_fields := { dob id_card mob carer  }, scripts_index := 2,
8         scripts := { l(4) := (# med_name := 0, s_fields := {  },
9                                 s_period := period_null,
10                                quant := 0, t1 := 0, t2 := 0 #)
11                    l(3) := (# med_name := 0, s_fields := {  },
12                                s_period := period_null,
13                                quant := 0, t1 := 0, t2 := 0 #)
14                    l(2) := (# med_name := 0, s_fields := {  },
15                                s_period := period_null,
16                                quant := 0, t1 := 0, t2 := 0 #)
17                    l(1) := (# med_name := 2,
18                                s_fields := { dosage prescription },
19                                s_period := daily,
20                                quant := 5, t1 := 7, t2 := 0 #)
21                    l(0) := (# med_name := 1,
22                                s_fields := { dosage prescription },
23                                s_period := bidaily,
24                                quant := 3, t1 := 3, t2 := 5 #) } #),
25    (# p_name := 2, %... details omitted #) } #),
26    %-- ...further entries and structures omitted #)
```

Listing 1.6: Displaying the effect of a sequence

The information provided by PVSio therefore makes it possible to check the effect of sequences of actions. It is possible to use sequences of this kind to demonstrate that in a particular context, as defined in a sequence of ground functions, an

action (or sequence of actions) will have a desired effect. An example of the sort of sequence that was explored in checking plausibility was to demonstrate that, after the execution of a sequence for creating more than five patients in the database (five is the limit of patients that can be shown in the screen), scrolling down the patient list in the relevant mode, followed by scrolling it up, produced the same display as before the scrolling actions were taken. Listing 1.7 shows the sequence that was explored. The context for the exploration is the state `susdmdnp2minus` produced by the sequence mentioned above. The sequence of actions considered is Listing 1.7. The result of performing the sequence is shown in Listing 1.6. The `patient_id_line` is unchanged and the line indexed by 0 points to `p_name = 1`.

PVSio therefore makes it possible to test the model to check that the behaviour coincides with the expected behaviour insofar as it is represented in the sketch model. In the next section we consider template properties [10] that are designed to check use related properties of the emerging design. In Section 5.2 we prove that the patient list scrolling actions are inverses of each other.

```
scrolldscrollu: state = LET st = susdmdnp2minus,
                            st = scroll_down_patient_list(st)
                        IN scroll_up_patient_list(st)
```

Listing 1.7: Adding scroll down followed by scroll up

4.2 From plausibility checks to plausibility theorems

PVS theorems can be used to demonstrate that the model has consistent and desirable behaviour thus providing confidence in its plausibility. An example illustrates the process. Consider for example the mode that allows the entry of patient details. Based on our understanding, actions key1, key3, confirm, prescriptions and quit are visible inviting the user to take one of these actions. The PVS theorem (see Listing 1.8) aims to prove that these actions have the desired effect, producing the relevant mode displays and updating the patient database appropriately. Thus it can be demonstrated that one step behaviours are consistent with those suggested of the sketch design for all states of the pill dispenser. As an example, consider quit. The sketch indicates that the action takes the pill dispenser to a mode where a list of patients, taken from the patient database, is shown (see line 26) and takes no further action.

On the other hand pressing confirm also updates the patient database (if any changes have been made in the patient details screen) and produces the patient list screen (see lines 7-18). Furthermore the sketch indicates that the confirm action is only permitted if all the relevant fields have been selected and entered. In this case the patient database is updated with the temporary patient record (`st2'temp_patient`) using the function `p_insert` which inserts the patient into the database (lines 16-18). The database is ordered and the insertion either replaces an existing record or inserts the record in the right place in the list. In the case of prescriptions the database is updated with the temporary patient and a transition is made to the current list of prescriptions for that patient.

A collection of PVS theorems like `check212` demonstrates that expected transitions take place and have been verified using the PVS theorem proving assistant.

```
check212: THEOREM FORALL (st: state):
 ((p_current(st) < p_max(st)) AND (p_max(st) < plimit))
    IMPLIES
  LET st1 = patient_details_screen(st)
   IN ((select(key1, st1) = init_screen(st)) AND
       (select(key3, st1) = db_menu_screen(st)) AND
       %-- set up the state for the confirm action
       (LET st2 = enter(name, select(name, st1)) IN
       (LET st2 = enter(dob, select(dob, st2)) IN
       (LET st2 = enter(id_card, select(id_card, st2)) IN
       (LET st2 = enter(carer, select(carer, st2)) IN
       (LET st2 = enter(mob, select(mob, st2)) IN
       %-- the effect of the confirm action
         (select(confirm, st2) =
             patient_list_screen(st2 WITH [
               patients_db := p_insert(st2`p_current,
                                       st2`temp_patient,
                                       st2`patients_db) ])))))) AND
       %-- the effect of the prescriptions action (sets up scripts list)
       (select(prescriptions, st1) =
        LET tp = p_find(st`p_current, st`patients_db),
            stx = st WITH [ patients_db := p_insert(st`p_current, tp,
                                                    st`patients_db),
                            temp_patient := tp ]
          IN script_list_screen(stx)) AND
       (select(quit, st1) = patient_list_screen(st)))
```

Listing 1.8: Plausible actions from the patient details screen

5 Proving properties

Once a plausible model has been developed it is possible to do further exploration. This includes user evaluation of a realistic prototype, but it also makes it possible to analyse the behaviour of the modelled prototype against use-centred requirements [10]. These requirements may include safety requirements that are used in the software safety analysis required by the regulator. This step therefore enables a more exhaustive analysis of the emerging design than would be possible with the functional prototype typically used in use centred design. It also supports software engineering of the system using a spiral model, and the mapping of a requirements specification including user centred requirements [17]. The approach is demonstrated by considering two use centred requirements: consistency and reversibility.

5.1 Consistency

Action Consistency

$$\forall a \in Act, s \in S, m \in MS : \\ guard(s, m) \wedge \\ pre\_filter(s, m) \varphi post\_filter(a(s), m) \quad (1)$$

The action consistency property is formulated as a property of either a single action, or of a group of actions (we will refer to them as Act) which may exhibit similar behaviours. A relation $\varphi : C \times C$ connects a filtered state, before an action occurs (captured by $pre\_filter : S \times MS \rightarrow C$), with a filtered state after the action (captured by $post\_filter : S \times MS \rightarrow C$).

There are many properties of the model of the sketch design that relate to its consistency. It is relatively common that actions are inconsistent in some detail. Consider, for example `quit` as represented in the storyboard. A first consideration of prototype material indicates that quit consistently changes mode without changes to either the patient or meds database. The action consistency template can be instantiated to a theorem that makes this assumption. The theorem fails to be true because there is a special case during the patient's thumb print registration sequence when `quit` is used to exit the sequence and the patient database is changed. The consistency template (1) instantiation is reformulated to include a guard that excludes this feature. The formulation of the theorem is as follows: it uses a simple guard (`mode(st) /= creation_success`), and the filters extract the attributes that specify the patient database and the medicine database:

```
quit_consistency_thm: THEOREM FORALL (st: state):
  mode(st) /= creation_success
    IMPLIES
  LET st1 = select(quit, st)
   IN (st`meds_db = st1`meds_db AND st`patients_db = st1`patients_db)
```

5.2 Reversibility

When testing plausibility using PVSio we considered the reversibility of scroll actions. The testing that was done inevitably considered only specific states of the patient database that generated the patient listing (see Section **??**). A general reversibility property, which proves this requirement for all states, is identified in the reversibility template as follows. This template is formulated for a group of actions $Act \subset S \rightarrow S$ using $guard : S \rightarrow \mathbf{B}$, and a $filter : S \rightarrow C$ relevant to the entry mode. For each $a \in Act$, there corresponds a $b \in Act$ such that:

Reversibility

$$\forall\, s \in S : guard(s) \Rightarrow \\ filter(b(a(s)) = filter(s)) \qquad (2)$$

This template can be used to prove that scrolling actions have required characteristics. Consider two actions `scroll_up_patient_list` and `scroll_up_med_list` and their inverses `scroll_down_patient_list` and `scroll_down_med_list`. The guards require that respective list screens are visible. The theorem is expressed using a function that instantiates the template and is proved using structural induction. Structural induction assumes that the property is true of a state and then proves that as a consequence it is true of any state that can be reached by the actions supported of the device. The verification of the theorem as formulated succeeds, i.e., the formulated property is true of the design.

Fig. 4: Pillbox prototype based on concept design image.

```
%-- reversibility of scroll actions
confirm_ud_scroll_fn(st: state): boolean =
 (mode(st) = patient_list
   IMPLIES scroll_down_patient_list(scroll_up_patient_list(st)) = st)
  AND (mode(st) = db_med_list
        IMPLIES scroll_down_med_list(scroll_up_med_list(st)) = st)
%-- reversibility theorem, formulated using structural induction
confirm_ud_scroll_thm: THEOREM
  FORALL (pre, post: state):
    init?(pre) IMPLIES confirm_ud_scroll_fn(pre)
      AND (state_transitions(pre, post) AND
            confirm_ud_scroll_fn(pre) IMPLIES confirm_ud_scroll_fn(post))
```

6 Iterating the prototype

Once properties are proved of this version of the PVS model, a further prototype can be developed for co-operative evaluation with end users. The visual appearance of the prototype is based on a concept design image created, for example, using a photo-editing tool. PVSio-web is then used to create hotspot areas over the picture and link them to the PVS model. Hotspots over buttons represent input widgets of the prototype, and they are linked to transition functions defined in the PVS model. Hotspot areas over display elements are used to render the value of state variables so that the visual appearance of the prototype closely resembles that of the real system in the corresponding states.

Figure 4 shows a screenshot of the developed prototype. It uses 17 widgets to model different elements in the various screens of the pillbox. Listing 1.9 shows a snippet of JavaScript code used for creating the *home* button of the prototype. `TouchscreenButton` is the widget constructor. The `new` operator is used to create a new object of type TouchscreenButton. The created widget is stored in a variable `key1`. The *first argument* of the constructor is a string defining the widget identifier. The PVSio-web toolkit uses this string as a basis for deriving the name of the transition function in the PVS model to be linked to the widget. The full name of the transition function is constructed by concatenating the user action that activates the widget with the widget identifier.

For example, when the user clicks on the button, the transition function that will be evaluated is `act(key1, st)`. The *second argument* is a structure defining the coordinates and size of the widget. This is necessary to create an interactive overlay area of the correct size for the image used as a basis for the visual appearance of the prototype, and to position the interactive area in the correct place, that is the left side of the screen. The *third argument* provides information about the callback function to be invoked for refreshing the visual appearance of the prototype when the evaluation of the transition function associated with the button generates a new system state, as well as information on the visual appearance of the touchscreen button (label, colour, font).

The visual aspect of all widgets is refreshed each time the PVS specification is evaluated in PVSio. The evaluation of the specification occurs either when the user interacts with an input widget (e.g., presses a button), or periodically (if the device has internal timers that are ticking).

```
var key1 = new TouchscreenButton("key1", {
   top: 216, left: 230, height: 64, width: 64
 }, {
   softLabel: "home",
   backgroundColor: "green",
   fontsize: 16,
   callback: render
});
```

Listing 1.9: Creation of a touchscreen button using PVSio-web.

This refined version of the prototype benefits from improved look and feel. The results of the evaluation with end users is then used to iterate the design process.

7 Related work and Conclusions

While there is relatively little literature concerned with development techniques that combine informal representations of design with formal models, there are many activities that combine different formal descriptions of visual, functional and task elements. Bowen and Reeves [4] explore the relation between display and functional models. Their work also focuses on specifications of sketch designs and aims to enable analysis of these designs. We are not, however, aware of development of executable versions of their models. Haesen and others [8]

integrate models and informal design knowledge. Their focus is also the role of formal task models and abstract user interfaces in user centred design. They use personas, scenarios and related task models in their models. Graphical models of storyboards are produced along with constraints on these models. Bolton and others [3], Paterno and others [14] and Fields [6] combine task and functional models. Palanque and others [11] combine visual, functional and task elements.

An important challenge in developing the approach described in this paper was not to reduce the value of user centred design. A criticism often levelled at formal techniques is that they can have the effect of limiting the scope of the analysis, ignoring important broader issues. We believe that our analysis, as an adjunct to the techniques and approaches of user centred centred design, responds to these criticisms. A further concern is that the effort and knowledge involved in producing the models and performing the analysis are not cost effective. It is true that these are techniques that are not typically found in the toolkit of a development team, particularly the small teams that often design and implement medical devices. However the safety of medical devices, in particular, is crucial and a thorough analysis of usability issues is a key contribution ensuring safety.

An important future dimension of our work, currently under development, is to simplify and automate some of these processes. Tools for presenting and instantiating property templates are being developed. Heuristics are being developed to automate the proof of PVS theorems. We are also simplifying the process of using PVSio-web to construct prototypes from models. The aim is to make these techniques accessible to a wider group of developers.

Acknowledgement

We are grateful to Nuno Rodrigues, João Vilaça and Nuno Dias from IPCA (Polytechnic Institute of Cavado and Ave) who developed the first prototype of the pill dispenser. José C. Campos, Paolo Masci and Michael Harrison were funded by project NORTE-01-0145-FEDER-000016, financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

References

1. Beyer, H., Holtzblatt, K.: Contextual design: defining customer-centred systems. Morgan Kaufmann (1998)
2. Bolton, M.L., Bass, E., Siminiceanu, R.: Using formal verification to evaluate human-automation interaction, a review. IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans (99), 1–16 (2013)
3. Bolton, M., Jiménez, N., van Paassen, M., Trujillo, M.: Automatically generating specification properties from task models for the verification of human-automation interaction. IEEE Transactions on Human Machine Systems **44**(5), 561–575 (2014)

4. Bowen, J., Reeves, S.: Combining models for interactive system modelling. In: Weyers, B., Bowen, J., Dix, A., Palanque, P. (eds.) The Handbook of Formal Methods in Human-Computer Interaction, pp. 161–182. Springer International Publishing, Cham (2017)
5. Carroll, J. (ed.): Scenario based design: envisioning work and technology in system development. Wiley (1995)
6. Fields, R.E.: Analysis of erroneous actions in the design of critical systems. Ph.D. thesis, Department of Computer Science, University of York, Heslington, York, YO10 5DD (2001)
7. Gould, J.D., Lewis, C.: Designing for usability: key principles and what users think. Communications of the ACM **28**(3), 300–311 (1985)
8. Haesen, M., Van den Bergh, J., Meskens, J., Luyten, K., Degrandsart, S., Demeyer, S., Coninx, K.: Using storyboards to integrate models and informal design knowledge. In: Hussmann, H., Meixner, G., Zuehlke, D. (eds.) Model-Driven Development of Advanced User Interfaces, pp. 87–106. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
9. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**, 231–274 (1987)
10. Harrison, M., Masci, P., Campos, J.: Verification templates for the analysis of user interface software design. IEEE Transactions on Software Engineering (2018), epub ahead of print
11. Martinie, C., Palanque, P., Barboni, E., Winckler, M., Ragosta, M., Pasquini, A., Lanzi, P.: Formal tasks and systems models as a tool for specifying and assessing automation designs. In: Proceedings of the 1st International Conference on Application and Theory of Automation in Command and Control Systems. pp. 50–59. ATACCS '11, IRIT Press, Toulouse, France, France (2011)
12. Masci, P., Oladimeji, P., Zhang, Y., Jones, P., Curzon, P., Thimbleby, H.: PVSio-web 2.0: Joining PVS to HCI. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 470–478. Springer International Publishing, Cham (2015)
13. Monk, A., Wright, P., Haber, J., Davenport, L.: Improving your human-computer interface: a practical technique. Prentice-Hall (1993)
14. Mori, G., Paternò, F., Santoro, C.: CTTE: Support for developing and analyzing task models for interactive system design. IEEE Transactions on Software Engineering **28**(8), 797–813 (2002)
15. Muñoz, C.: Rapid prototyping in PVS. Tech. Rep. NIA Report No. 2003-03, NASA/CR-2003-212418, National Institute of Aerospace (2003)
16. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) Eleventh International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer-Verlag (1992)
17. Sommerville, I.: Software Engineering. Addison-Wesley (2010)
18. Watson, N., Reeves, S., Masci, P.: Integrating user design and formal models within PVSio-Web. In: Workshop on Formal Intergrated Development Environment (F-IDE-18). Electronic Proceedings in Theoretical Computer Science (EPTCS) (2018)