Advance Access publication on 21 April 2021
doi: 10.1093/iwcomp/iwab012

Balancing the Formal and the Informal in User-centred Design

Michael D. Harrison[1,*], Paolo Masci[2] and José Creissac Campos[3,4]

[1]*School of Computing, Newcastle University, Urban Sciences Building, Newcastle upon Tyne, NE4 5TG, UK*

[2]*National Institute of Aerospace, 100 Exploration Way, Hampton, VA 23666, USA*

[3]*Department of Informatics, University of Minho, Campus de Gualtar, 4710-057 Braga, Portugal*

[4]*HASLab - High-Assurance Software Laboratory / INESC TEC, Campus de Gualtar, 4710-057 Braga, Portugal*

[*]*Corresponding author: michael.harrison@ncl.ac.uk*

This paper explores the role of formal methods as part of the user-centred design of interactive systems. An iterative process is described, developing prototypes incrementally, proving user-centred requirements while at the same time evaluating the prototypes that are executable forms of the developed models using 'traditional' techniques for user evaluation. A formal analysis complements user evaluations. This approach enriches user-centred design that typically focuses 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 use of formal methods provides a systematic approach to checking plausibility and consistency during early design stages, while at the same time enabling the generation of executable prototypes. The technique is illustrated through an example based on a pill dispenser.

RESEARCH HIGHLIGHTS

- The paper describes a user centred design process that integrates the use of formal models, theorem proving and prototypes based on the formal description and empirical analysis.
- The process is illustrated with a realistic case study based on a pill dispenser device suitable for use in common areas in care homes or hospitals.

Keywords: user-centred design; formal methods

Handling Editor: Regina Bernhaupt

Received 5 December 2019; Revised 20 May 2020; Accepted 10 March 2021

1. INTRODUCTION

Traditional user-centred design approaches such as contextual design [Beyer & Holtzblatt, 1998] and scenario-based design [Carroll, 1995] focus on user tasks early and throughout the design process. They measure usability empirically and test and refine the design iteratively, based on results obtained through user evaluation sessions. The evaluations build on the use of *scenarios* and *sketch designs*. Scenarios capture typical or exceptional situations and tasks. Sketch designs represent possible design solutions that would improve the situation. They are often non-functional, for example it could be a simple PowerPoint presentation or a paper storyboard. Think-aloud techniques such as cooperative evaluation [Monk *et al.*, 1993] are typically used to collect feedback necessary to assess the design and judge whether a further iteration would be appropriate. These usability methods are well understood and provide important insights about the context inwhich a device is used, as well as the needs of different users.

In safety critical contexts, however, where guarantees about the interaction between users and systems are needed,

prototyping is not enough to provide the required level of assurance. Even if the prototype is detailed enough to capture all relevant aspects of the system, simple manual inspection does not provide the required level of analysis thoroughness.

This paper advocates a complementary process in which each step of the user-centred design process uses the, by now, traditional approach of presenting the current version of the interface to users *but*, at the same time, also involves a formal analysis of the model that is developed alongside the design. Specific aspects of the approach will be illustrated using a concrete example based on an automated pill box for dispensing medication to patients at specific times.

The initial sketch design of the pill box is introduced in Section 3, after the design process and the tools used to support the design process in this paper are introduced (in Section 2). The sketch design includes an explanation of the sketch prototype and how an executable formal model can be derived from the sketch. A requirement relating to the functionalities that should be available to different intended users (doctor, carer and pharmacist) is also discussed.

An enhanced design is then presented that fills various gaps observed of the initial design (Section 4). The formal model associated with the enhanced design is mechanically checked against plausibility properties (Section 5) using lightweight formal methods based on simulation and testing. The enhanced design is validated with end users by integrating the same formal model in a realistic interactive prototype that can be effectively presented to end users and domain experts (Section 6). Formal verification of requirements of the model is then described (Section 7). Use-related requirements include the consistency of the actions offered by the enhanced design (Section 7.1), the reversibility of scrolling behaviour supported by designed scrolling actions (Section 7.2) and mutual exclusion of different user pathways (Section 7.3). The final sections present a discussion of comparable approaches (Section 8) and concluding remarks (Section 9).

The formal aspects of two iterations of the user-centred design process are demonstrated for the illustrated design. This demonstrates a framework that may be used effectively for the design of use aspects of safety critical interactive systems.

Contributions of the paper are the following: (i) a user-centred design process that integrates the use of formal models and empirical analysis and (ii) an application of the approach to a realistic case study based on a pill dispenser device.

This paper is an extended version of Harrison *et al.* [2018]. The main extensions are as follows: an enriched description of the approach for integrating the formal analysis in standard user-centred design methods, an extended presentation of the requirements of the pill box device and a detailed illustration of the formal analysis carried out to verify use-related requirements of the pill box device.

A similar approach was presented in Harrison *et al.* [2019b], where verification was used *post hoc*, to guarantee relevant risks had been considered in the design. While the approach is similar, the concern in this paper is different and complementary, as the approach is used to support the design process itself.

2. THE APPROACH

Usability engineering usually assumes that part of design development involves a (possibly) iterative process in which designs are subject to some form of evaluation. In some cases, these design stages can involve non-functional prototypes, see for example Beyer & Holtzblatt [1998]. This paper argues that user-centred design can involve a formal stage. The design is modelled and the model is executed but at the same time properties are developed from user-centred design requirements developed with the participation of users. This may include usability heuristics, for example Nielsen & Molich [1990], and use-centred regulatory requirements, for example Masci *et al.* [2013]. The envisaged design process is as follows:

Step 1: use-centred requirements. An initial set of use-centred requirements is developed through, for example, contextual enquiry. These requirements are typically formulated using natural language. At each iteration of the design process the requirements are validated or supplemented by further requirements relevant to the current state of the model.

Step 2: executable formal model. A formal model is developed that captures the functionalities indicated in the requirements. The formal model is checked mechanically using lightweight formal methods based on simulation and testing. The aim of this step is to gain confidence that the specification satisfies basic properties, e.g. coverage of conditions.

Step 3: validation of requirements. A mockup prototype is created that can be presented to end users and domain experts. The visual appearance of the mockup captures that of the final system. The behaviour of the mockup is driven by the executable formal model developed in Step 2. The mockup is evaluated with end users and domain experts. The main aim of this step is to *validate requirements*, i.e. make sure that developers are creating the right system. Requirements, model and mockup are modified as gaps and errors are found. This step iterates as necessary.

Step 4: verification of requirements. Requirements are proved of the model using exhaustive formal methods approaches, e.g. model checking or theorem proving. The formal model is modified when verification attempts point out corner cases where requirements are not satisfied. Requirements may also be modified as gaps and errors are found through the formal analysis. End users are brought back into the loop when iterating the design process (see also description of Step 5)

Step 5: iterating the system design. Requirements and mockup are iterated starting again at Step 1. Step 2 brings end users and domain experts back into the development loop. Requirements and mockup are re-validated in Step 3, to make sure the changes introduced by developers are accepted by end users and domain experts. Changes introduced in Step 3 lead to a new formal verification phase (Step 4). The iteration continues until a satisfactory design has been produced.

The approach requires, besides support for developing mockups, a formal verification tool (theorem prover or model checker) that is able to support both the verification and the animation of the models. The technologies adopted in this work are introduced in the next sub-section.

2.1. Primer on PVS and PVSio-web

PVS [Owre *et al.*, 1992] is an interactive theorem proving system based on sequent calculus. PVS specifications are called *theories*. Properties to be proved of the specification are called *theorems*. Theories and theorems are specified in a higher-order logic language.

The basic elements of the PVS syntax are as follows (additional elements will be introduced and explained later in Section 4, when introducing the pill box example):

- Function types: they are type expressions of the form `[domain -> range]`, where `range` can be another function type. The commands defined in this paper are typically of the form `[state -> state]`; hence, their type defines a function that transforms the current state of the device into a new updated state following the execution of an action (for example, `pwd_screen(st: state): state`, see Listing 1.7). Anonymous functions are expressed using the `LAMBDA` keyword. They can be conveniently used to initialize and update data structures. For example, `LAMBDA (x: fields_type): FALSE`, where `field_type` is an enumeration, is a function of type `[fields_type -> boolean]` that can be used to create a hashmap table where all elements are initialized to the boolean constant `FALSE`.
- Subtypes: these types use a predicate to restrict the domain of another type. They are defined using the syntax `S: TYPE = { x: T | P(x) }`, where `T` is a type, `P` is the subtyping predicate and `S` is a subtype of `T`. The same subtype can be expressed using a more compact form `(P)`, where the predicate name is within round parentheses. Subtypes are used in the paper to describe commands that are only available under particular conditions. For example, in the definition of the command `password_screen` (see Listing 1.1), the function is only permitted if a predicate `per_password_screen`, which is called the *permission* function, is true. Hence, the signature of `password_screen` has the following form: `password_screen(st: (per_password_screen)): state`. The predicate `per_password_screen` is only true if the state of the device is in certain modes.
- Enumerated types are declared by listing the enumerated constants between a pair of curly brackets. An example is `path_mode_type: TYPE = { scripts_path, meds_path, no_path }`, where `path_mode_type` is the name of the enumerated type and `scripts_path`, `meds_path` and `no_path` are the enumerated constants.
- Record types are declared by listing the name of the record attributes between a pair of square brackets and the hash symbol. An example record type declaration is as follows:

```
patient_type: TYPE =
  [# p_name : p_index,
     p_fields : fields_set,
     scripts_index: s_index,
     scripts: list_script_type #]
```

  where `patient_type` is the name of the record type; `p_name`, `p_fields`, `script_index` and `scripts` are record attributes. `p_index` and `s_index` are types defining the set of patient and script names, respectively. `fields_set` defines the set of fields describing the patient and `list_script_type` the list of prescriptions for the patient.
- Record literals are specified as a list of assignments between a pair of round brackets and the hash symbol. An example literal is as follows:

```
nil_patient: patient_type =
  (# p_name := p_null,
     p_fields := LAMBDA (x: fields_type): FALSE,
     scripts_index := s_null,
     script := nil_list_script #)
```

  where `nil_patient` is the name of the literal. This literal specifies initial values for the patient fields. The expression `nil_patient`p_name` can be used to access a record field. An equivalent functional notation `p_name(nil_patient)` can also be used for the same purpose.

As mentioned earlier in this section, the PVS theorem prover uses sequent calculus. Theorems to be proved of a specification are presented in the form

```
{-1} A1
{-2} A2
{-3}..
|———————
{1} S1
{2} S2
{3}..
```

where A1,..., An are formulae called *antecedents* and S1,..., Sn are formulae called *consequents*. Intuitively, antecedents can be thought of as the hypotheses of the theorem, and the consequents comprise the theorem to be proved. Inference rules are used to perform the proof. A proof is considered complete when the application of the inference rules leads to a consequent that is true, or an antecedent that is false, or when an antecedent occurs also as a consequent. A first example in the paper of a proof of this kind is presented and discussed in Section 3.3.

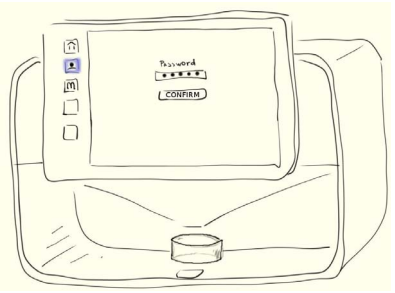
PVSio-web [Masci *et al.*, 2015b] is a web-based environment that enables the creation of interactive prototypes based on executable PVS specifications. The toolkit supports the creation of both *storyboard-based prototypes* using mockup pictures of different screens of the system under development [Watson *et al.*, 2018] and *high-fidelity prototypes* that can closely resemble the visual appearance and behaviour of a final product. The interactive prototypes, so constructed, can be evaluated with end users.

PVSio-web prototypes use a split architecture to create a separation of concerns between the specification of the behaviour of the prototype and its visual appearance.

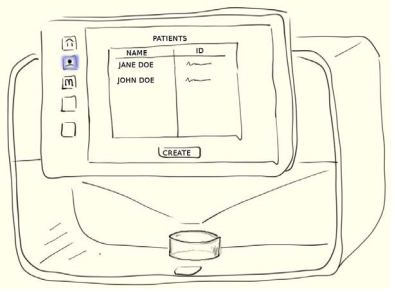
- The back-end of a PVSio-web prototype defines the behaviour of the prototype. The PVSio evaluation environment [Muñoz, 2003] is used to compute the evolution of the prototype, including how the prototype reacts to user actions and other system events. PVSio executes specifications written in the logic language of the PVS verification system by automatically translating the model into Lisp code. PVS specifications can either be created manually or automatically generated from Emuchart diagrams. Emuchart is a graphical state transition language that is a simplified version of Statecharts [Harel, 1987].
- The front-end is executed in a Web browser, rendering the visual appearance of the prototype. A picture of the real system is used as a basis for its visual appearance. JavaScript code is used to create interactive areas over the picture to enable user interactions with the prototype and are translated into evaluation commands for the back-end. A set of widgets provided by PVSio-web facilitates the definition of common interactive elements such as buttons and touchscreen elements. The front-end is refreshed every time the state of the underlying PVS specification changes. State attributes of the PVS specification that are intended to be visible in the user interface are rendered as displays in the prototype, thereby reproducing the attribute's look and feel.

3. THE DESIGN PROBLEM

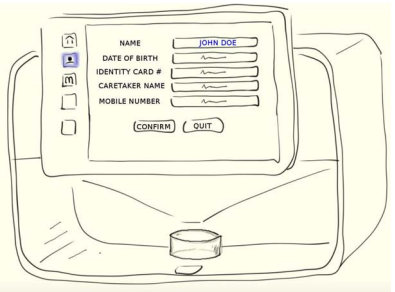
The design problem that is used in the paper had been explored originally at the Polytechnic Institute of Cávado and Ave in Portugal. A prototype had already been developed by that organization. A system was required that could be provided in care-home common rooms or hospital wards. Its purpose would be to enable the alerting and dispensing of medications at appropriate times and with appropriate prescriptions. It was proposed that patients be alerted when a dose was due according to their particular prescriptions. Only the designated patient, or their doctor or nurse carer, was to be permitted to access the required dose. The solution that had been originally sketched, see Fig. 1, assumes a stand-alone device that provides the required facility to patients. The sketch, used as a starting point for the analysis, was developed from the prototype that already existed. Facilities in the proposed design illustrated the following: (i) a database of medications and doses, available to be used as prescriptions, that could be viewed and in some cases updated by authorized personnel and (ii) the updating of patient details and associated prescriptions. This latter facility



(a) Password screen



(b) Patient list screen



(c) Patient details screen

FIGURE 1. Example sketch images produced for the initial prototype.

required both an appropriate authorized person and the patient. The patient's thumb print was to be used for user authentication.

The device, as envisaged, 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 or through gated access by the designated carer (though this aspect of the administration process is not the focus of the paper). The device maintains a database of patients who have been subscribed to the system as well as a list of their prescriptions. The pill dispenser supports 'columns' of pills from which the patient can obtain their required dose.

3.1. Pathways

The sketch design indicates three user pathways: (i) making available to the patient the appropriate medication at the appropriate time, (ii) setting up and editing the patient details along with a list of medication prescriptions and (iii) updating the medications database. The rest of this paper focuses only on prototypes and specifications based on pathways (ii) and (iii) because the main design challenges covered in these two pathways also relate to the first pathway (see next sub-section for a discussion of the design challenges). A design challenge, that is not discussed here, relates to issues associated with timing. Our previous work on infusion pumps [Harrison *et al.*, 2017] demonstrates how to address the timing dimension.

3.2. Design challenges and preliminary requirements

The pathways provide insights into the initial set of requirements. The following main design challenges can be identified:

- Patient functions. The system should include functions that allow a patient to obtain a medication through thumbprint identification.
- Carer functions. The system should allow designated carers to create patient logins, enter patient details and enter prescription details.
- Pharmacy functions. Functions should also be provided to update medications and prescription details.

Preliminary requirements are identified based on these design challenges. The initial focus is on the security and usability of the design. The pathways must satisfy requirements that relate to mutually exclusive access. The person that is authorized to update the medications database should not have access to patient details. The person that is authorized to update and view patient details should not be able to update the medical database. Clearly, the design should not allow a circumstance where the user is able to access or update data that should not be available to them. The design should also satisfy use-centred requirements. In particular, actions available to the user should have consistent behaviour. These requirements will be made more precise as the specification is developed in subsequent sections of the paper.

3.3. The initial formal model

A video of the early prototype of the pill box device was provided to the authors. This formed the basis for a storyboard that was suitable for initial evaluation with stakeholders (see Fig. 1). Sketches representing each display state were created based on the material provided by the video. Appropriate images were used to illustrate entry, or modification, of patient details as well as displays that indicate the requirement for password entry by the nurse or carer responsible for setting up patient details. For example, images relating to patient details illustrate the following aspects:

- when patient information can be entered;
- the authentication sequence based on the patient's thumb print;
- how to access the list of patients in the database;
- how to change the details stored for each patient;
- how a prescription can be added or removed.

Similarly, storyboard images, related to the medications database, indicate that the database is protected by password, and access is only enabled for doctors or pharmacists who are able to enter details of medicines. This pathway includes screen displays that allow entry or editing of medicines. The medicine pathway includes display screens that allow medicines to be listed or displayed and modified. The storyboard describes the paths that are possible with non-functional representations of these displays. It gives no information about the other actions within each state.

The storyboard images were used to create a first interactive prototype in PVSio-web. Figure 2 indicates a phase of the creation of the prototype. 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.

PVSio-web creates a state machine diagram from the interactive storyboard. The state machine is represented using an Emuchart. The diagram is then translated into an executable PVS specification. Each node in the Emuchart represents a display screen, and each labelled edge represents a transition between screens. A fragment of the Emuchart can be seen in Fig. 3.

Listing 1.1 shows the PVS specification that was generated automatically from the Emuchart diagram. Nodes are translated into an enumerated type `Mode`. Labelled edges are translated into PVS functions. The state of the diagram is represented using a record type `state` that includes an attribute `current_mode` of type `Mode`, indicating the current screen. The initial state of the specification is thus specified

(Lines 8 and 9) as a record literal with `current_mode` set to `initial_screen`.

The transition function `password_screen` (Lines 18–29) is linked to the Emuchart edges that enter node `password_screen`. The function uses a subtyping predicate, `per_password_screen`, to restrict the domain of the function. The predicate reflects the structure of the Emuchart, as edges entering node `password_screen` originate only from three nodes, `database_password_screen`, `pill_dispensed_screen` and `initial_screen` (see Fig. 3). Two auxiliary functions, `leave` and `enter_into`, are used to update attribute `current_mode`.

This storyboard provides sufficient information to serve two purposes.

The first purpose is to allow a user to navigate the state transition model associated with the interactive storyboard. It shows sketch images at each state of the model (see Fig. 2 for example). In practice, this can be used as a preliminary stage of user evaluation with relevant stakeholders.

The second purpose is to provide a basis for demonstrating that the initial requirements, developed through the initial elicitation process, are satisfied. An important initial requirement is that the patient database and the medical database can only be updated by relevant authorized personnel who have entered their passwords (see Section 3.2). Those authorized to access the medical database are not authorized to access the patient database.

In the case of this initial requirement, more formal analysis enables a more exhaustive exploration of all the possible paths. To specify this requirement as a property, it was necessary to adjust the initial specification generated automatically from the Emuchart. Additional attributes were added manually to the PVS specification necessary to keep track of which pathway is taken and which password is used to access the system.

The state of the device is augmented with two attributes: `path_mode_set`, indicating which credentials were used to access the system, and `path_mode`, which associates each screen to a pathway. If these two attributes differ, then the user is attempting an unauthorized entry. These changes in the structure of the state require some adjustments in functions `init` and `enter_into` generated automatically from the Emuchart, see Listing 1.2. The transition (`enter_into`) is updated by including two assignments to the added attributes to `state` (see Lines 23–30).

An example requirement that can now be checked is as follows: it should not be possible to reach the screen `new_med` (which allows updating of the medical database) unless the user has gone through the `db_pwd` screen. The screen `new_med` is associated with `meds_path`, and therefore the attribute

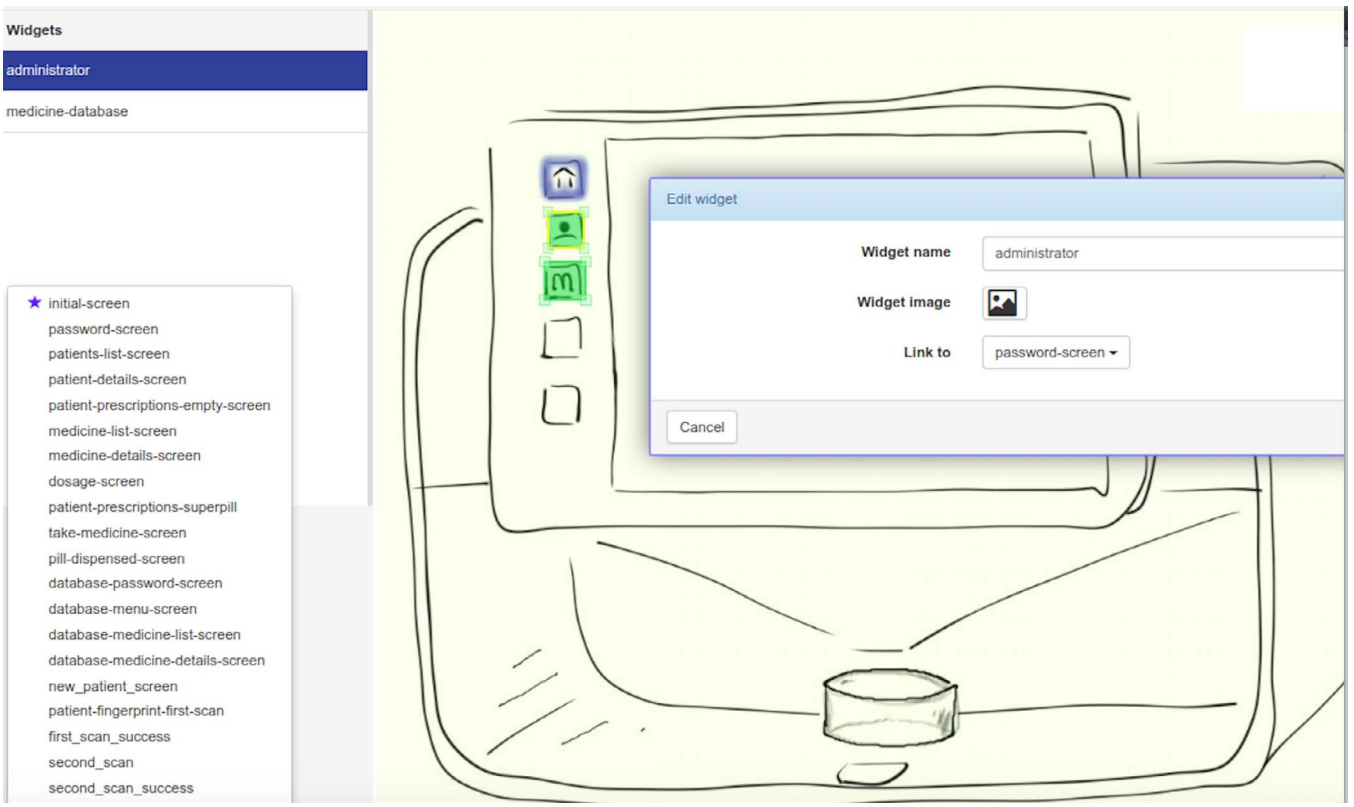


FIGURE 2. Creation of the sketch design in PVSio-web.

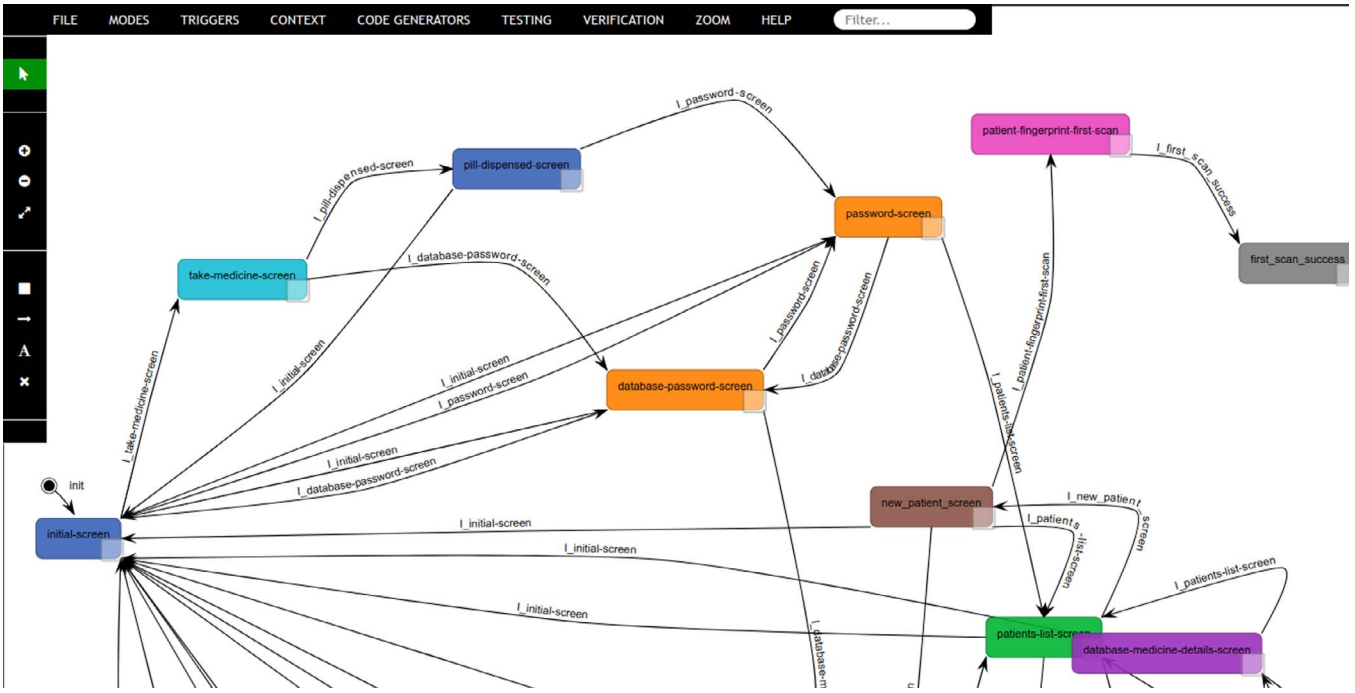


FIGURE 3. The initial sketch: Emucharts diagram.

```
1  pill_dispenser: THEORY BEGIN
2    %-- sketch screens
3    Mode: TYPE = { initial_screen, password_screen,
4                   patients_list_screen, ... }
5    %-- state attributes
6    state: TYPE = [# current_mode: Mode #]
7
8    %-- init function
9    init: state = (# current_mode := initial_screen #)
10
11   %-- subtyping predicate for password_screen
12   per_password_screen(st: state): bool =
13     (screen(st) = initial_screen)
14       OR (screen(st) = pill_dispensed_screen)
15       OR (screen(st) = database_password_screen)
16
17   %-- edges entering password_screen
18   password_screen(st: (per_password_screen)): state =
19     COND
20     screen(st) = initial_screen
21     -> LET st = leave(initial_screen)(st)
22         IN enter_into(password_screen)(st),
23     screen(st) = pill_dispensed_screen
24     -> LET st = leave(pill_dispensed_screen)(st)
25         IN enter_into(password_screen)(st),
26     screen(st) = database_password_screen
27     -> LET st = leave(database_password_screen)(st)
28         IN enter_into(password_screen)(st)
29     ENDCOND
30   %-- ...more transition functions omitted
31 END pill_dispenser
```

LISTING 1.1. Fragment of the PVS specification generated from the Emuchart.

```
1  path_mode_type: TYPE = {
2    scripts_path, meds_path, no_path
3  }
4
5  state: TYPE = [#
6    current_mode: Mode,
7    path_mode_set: path_mode_type,
8    path_mode: path_mode_type
9  #]
10
11 init: state = (#
12   current_mode := initial_screen,
13   path_mode_set := no_path,
14   path_mode := no_path
15 #)
16
17 screen_map(m: mode_type): path_mode_type
18   = %-- definition omitted for brevity
19
20 enter_into(m: mode_type)(st: state): state =
21   st WITH [
22     current_mode := m,
23     path_mode_set :=
24       COND
25         m = pwd -> scripts_path,
26         m = db_pwd -> meds_path,
27         m = initial -> no_path,
28         ELSE -> path_mode_set(st)
29       ENDCOND,
30     path_mode := screen_map(m)
31   ]
```

LISTING 1.2. Re-defining `enter_into`.

`path_mode_set` should be `meds_path` if the user has, as required, gone through the medical database password (`db_pwd`) screen. The formalization of this requirement can be found in the theorem given in Listing 1.3. The theorem uses a structural induction. It checks, for all accessible states, that `path_mode` and `path_mode_set` are as they should be, that is defined by `path_function` (see Line 3) and remain the same through state transitions (see Line 6). Note that

path_function is defined to associate the expected path mode with each state of the model. This function is described in detail in relation to the second iteration of the model (see Listing 1.25).

```
pathway_mode_thm: THEOREM
  FORALL (pre, post: state):
    (path_function(current_mode(pre)) =
      path_mode_set(pre) AND
        state_transitions(pre, post)) IMPLIES
          (path_mode(post) = path_mode_set(post))
```

LISTING 1.3. Pathway theorem relating to the sketch design.

A state is accessible if it can be reached from a previous state by a transition function. The transition function is described by the predicate state_transitions, part of which is illustrated in Listing 1.4.

```
state_transitions(pre, post: state): boolean =
  per_password_screen(pre) AND
  post = password_screen(pre)
  OR per_take_medicine_screen(pre) AND
  post = take_medicine_screen(pre)
  OR ...
```

LISTING 1.4. The state transitions predicate.

The theorem fails with many counter-examples. One of the counter-examples indicates that it is possible to move from db_menu and then to db_patient_list (which can only be viewed by a carer). The counter example generated by PVS, when attempting proof, can be seen in Listing 1.5. The proof fails. This can be seen by combining elements of the antecedent [-3] in Lines 4–8. path_mode_set(post) is meds_path, but the current path mode is scripts_path. The consequent in Line 13 therefore cannot be proved.

```
1  pathway_mode_thm.4.2.2.1.2.2.2.1 :
2  [-1]   current_mode(pre!1) = db_menu
3  [-2]   meds_path = path_mode_set(pre!1)
4  [-3]   post!1 = pre!1
5  WITH [previous_mode := db_menu,
6  current_mode := patient_list,
7  path_mode_set := pre!1`path_mode_set,
8  path_mode := scripts_path
9  |-------
10 [1]    current_mode(pre!1) = db_pwd
11 [2]    current_mode(pre!1) = pwd
12 [3]    current_mode(pre!1) = initial
13 [4]    (path_mode(post!1) = path_mode_set(post!1))
```

LISTING 1.5. Counter-example to Listing 1.3 theorem.

Further details in the design and its specification are required to prove a suitable path requirement as well as other use-centred requirements. These details will be discussed in Section 4.

4. THE SECOND ITERATION

The second iteration aims to fill gaps in the initial prototype (e.g. data entry functions and other details that were shown only in the images of the storyboard and not captured in the Emuchart diagram). The first iteration of the model simply indicates which transitions can take place between which screens. It does not provide detail about the content of the values entered, simply that the information is entered. The model created in the second iteration describes, in more detail, all these aspects. It provides a more functional representation that can be used in analysing requirements and evaluating the usability of a prototype.

Changes, as a result of evaluation of the previous iteration of the design, through the sketch prototype with end users, and through analysis of the pathway requirements, led to the new specification. Clearly, this may not be a refinement of the design in the strict formal sense [Morgan, 1994]. Changes in response to evaluation may in fact include changes in functionality.

The changes involved restructuring the model (see the repository[1] for a full specification of the restructured model). The new specification describes the circumstances under which an action can be taken (for example, through the entry of relevant data fields) and also the effect of the action. The result of executing an action depends on the display mode. The device state therefore describes the structure of the screen, including the actions that are visible and available, as well as which fields must be entered before an action can be completed. Actions cause transitions between modes and can also include updates to the state, for example updates to the patient database. Transitions may also be caused by selecting fields and entering values. These latter 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).

The remainder of this section describes illustrative fragments of the new specification in more detail.

4.1. Modes, keys, fields and databases

The type associated with display modes and path modes are based on those identified in the first iteration. Fields used to handle data entry (Lines 16–18 of Listing 1.6) and a field indicating the availability of actions (Line 19) have been added to the device state. The definition of these fields required the creation of two new data types: available_actions_type and fields_set (Lines 10–11). They are arrays of booleans that store information about the availability of actions and fields in a given device state.

The new version of the model includes functions that represent transitions between display states in the same way as

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

the first iteration. For example, `password_screen` (the initial specification is in Listing 1.1) is now represented as a function `pwd_screen` (see Listing 1.7) which changes mode to the display that requests a password to enter the scripts path. The function changes attributes `mode`, `path_mode` and `path_mode_set` appropriately. The function changes the availability of actions (Lines 6–9) as well as the visibility of fields (Lines 10–11) and indicates that no actions are selected and no fields selected or entered. A function `clear_screen` (Line 2) provides the basic blank state of the screen (with no actions or fields, and fields are not selected or entered) which is enriched by changes to the `state` attributes defined in the `WITH` clause (Lines 3–12).

```
1  mode_type: TYPE = { initial, pwd, db_pwd, db_menu,
2                      patient_list, db_med_list,
3                      new_patient_details, ... }
4
5  actions_type : TYPE = { key1, key2, key3,
6                          confirm, create, ... }
7  fields_type : TYPE = { password, dob, dosage,
8                         id_card, mob, carer, ... }
9
10 available_actions_type: TYPE = [actions_type->boolean]
11 fields_set: TYPE = [ fields_type -> boolean ]
12
13 state: TYPE = [# mode: mode_type,
14                 path_mode: path_mode_type,
15                 path_mode_set: path_mode_type,
16                 vis_field: fields_set,
17                 sel_field: fields_set,
18                 ent_field: fields_set,
19                 action: available_actions_type,
20                 ... #]
```

LISTING 1.6. Types used in the second iteration model.

From the password screen, the user is first required to select the password field (which is visible, see Lines 10–12 in Listing 1.7) and then enter the password before selecting an appropriate action. Two new functions are required to specify this functionality: `enter`, which defines how to enter the password, and `select`, which defines how to select fields and actions in the screen.

```
1  pwd_screen(st: state): state =
2     clear_screen(st) WITH [
3          mode := pwd,
4          path_mode := scripts_path,
5          path_mode_set := scripts_path,
6          action :=
7             LAMBDA(x: actions_type):
8               x = key1 OR x = key2 OR
9               x = key3 OR x = confirm,
10         vis_field :=
11            LAMBDA(x: fields_type):
12              x = password
13    ]
```

LISTING 1.7. The second iteration password screen.

The function `enter` is illustrated in Listing 1.8. In the simple case of password entry, the function checks that the password field is selected (Line 2) and assigns the field as entered (Lines 16–18) and clears the selection of the field (Line 19).

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

LISTING 1.8. Entering a field.

In the cases identified by the permission functions (Lines 5, 9 and 13), further checks are carried out when specific fields are being entered. For example, entering details associated with daily or weekly prescriptions will vary according to the period. There is insufficient space to describe this part of the specification in detail. The full specification in the repository provides more information.

The `select` function is polymorphic and can be applied both to fields (Listing 1.9) and actions (Listing 1.10). When applied to a field, if the field is visible (Line 2 in Listing 1.9), then the function flags the given field as selected. This is not made explicit in the model but there is an underlying assumption that there is a process of selection that maps to the `select` function (for example moving the cursor to the action icon or displayed field and using a mouse and clicking to select). Otherwise, the state remains unchanged.

```
1  select(f: fields_type, st: state): state =
2    IF st`vis_field(f)
3    THEN st WITH [
4      sel_field := LAMBDA(x: fields_type): x = f
5    ] ELSE st ENDIF
```

LISTING 1.9. The select functions.

When applied to an action, the `select` function checks that the action is available and then invokes an auxiliary function `act` that updates the device state (Line 2 in Listing 1.10). If the action is not available, the state remains unchanged.

```
1  select(a: actions_type, st: state): state =
2    IF st`action(a) THEN act(a, st)
3    ELSE st ENDIF
```

LISTING 1.10. The select functions.

A fragment of the definition of `act` is shown in Listing 1.11. The fragment illustrates the actions associated with `key1` and `create`.

```
act(a: actions_type, st: state): state =
  COND
    a = key1 AND per_act_key1(st) -> act_key1(st),
        ...
    a = create AND per_act_create(st) -> act_create(st
    ),
    ELSE -> st
  ENDCOND
```

LISTING 1.11. Auxiliary function `act`.

The `key1` action simply makes a transition to the initial screen (see Listing 1.12).

```
act_key1(st: (per_act_key1)): state = init_screen(st)
```

LISTING 1.12. The act key1 function.

The `create` action causes, for example, transition to display modes that allow the entry of new patient details or new medicine details. The function is extensive and Listing 1.13 illustrates only part of it.

```
act_create(st : (per_act_create)): state =
  COND
    mode(st) = patient_list
      -> new_patient_details_screen(st),

    mode(st) = db_med_list
      -> new_med_screen(st),
    ...
  ENDCOND
```

LISTING 1.13. The act create function.

A function `new_patient_details_screen` transforms the state to produce a screen that enables the entry of the details of the new patient. The specification of the function is in Listing 1.14. It identifies the current mode of operation of the device (Line 4), as well as the actions that are visible (Lines 6–9), the fields that are visible (Lines 19–22) and the fields that are selected (none in this case, see Line 23). To keep this initial model simple and avoid the definition of functions for the input of strings, the patient name is taken to be identified automatically by the device (Line 2) and already filled in when the user enters this mode (Line 24). The attribute `path_mode` describes the path that should have been entered, based on user authentication that had occurred at some stage in the past. Attribute `path_mode_set` is not changed and specifies the path that has actually been established. As will be shown in Section 7.3, these attributes are used to support the verification of the path exclusion requirement.

Each time a field is entered, temporary data structures, necessary for data entry, are initialized: patient record (Lines 10–14),

```
new_patient_details_screen(st: state): state =
  LET np = next_pid(st`p_max)
   IN clear_screen(st) WITH
      [ mode := new_patient_details,
        path_mode := scripts_path,
        action :=
          LAMBDA(x: actions_type): (x = key1)
                    OR (x = key2) OR (x = key3)
                      OR (x = confirm) OR (x = quit),
        temp_patient :=
         (# p_name := np,
            p_fields := LAMBDA(x: fields_type):FALSE,
            scripts := LAMBDA (s: s_index):nil_script,
            scripts_index := s_null #),
        temp_script := nil_script,
        temp_med := nil_med,
        m_current := m_null,
        p_current := np,
        vis_field :=
          LAMBDA(x: fields_type): (x = name)
                    OR (x = dob) OR (x = id_card)
                      OR (x = carer) OR (x = mob),
        sel_field := LAMBDA(x: fields_type): FALSE,
        ent_field := LAMBDA(x: fields_type): x = name
     ]
```

LISTING 1.14. The specification of the new patient details screen.

prescriptions associated with the patient (Line 15) and the list of medicines (Line 16). Only when the user confirms the entered data, the temporary structures are used to update the relevant databases.

Databases are represented as a list of records. For example, the data type used for the patient database is in Listing 1.15.

```
list_script_type: TYPE = [s_index -> script_type]
patient_type: TYPE = [# p_name : p_index,
                        p_fields : fields_set,
                        scripts: list_script_type,
                        scripts_index: s_index #]
patient_db_type: TYPE = list[patient_type]
```

LISTING 1.15. Patient database types.

The patient name (attribute `p_name`) is used as key in the database. Information in the patient record include a set of patient fields indicating date of birth, carer and so on (Line 3), as well as a list of prescriptions associated with the patient (Lines 4–5). These details can be found in the repository.

4.2. Evaluating the new specification

When the new specification is ready, three steps are taken again to evaluate the design (Steps 2, 3 and 4 illustrated in Section 2). Step 2 involves checking plausibility properties of the specification using lightweight formal methods. Step 3 produces a prototype that can be presented to end users and domain experts for usability evaluation. Step 4 consolidates the design by proving theorems necessary to show that the specification satisfies the requirements established as part of the design process. These steps are illustrated in the remainder of this paper.

It is worth emphasizing that the specification produced in the first iteration simply puts flesh on the storyboard prototype described in Section 3.3. The version under analysis in this second iteration provides more detail about the actions, modes and fields used for the entry of relevant parameters. As already highlighted, this process of fleshing out interaction detail and adding functionality is not conventional formal refinement because at each step the design is in flux, open to changes as a result of evaluation and discussion with potential users.

5. ANIMATING FOR PLAUSIBILITY

Plausibility of the specification can be assessed by using PVSio. This allows animation of grounded versions of the specified functions. In consequence, a form of direct interaction with the model is possible to exercise the available actions and observe their effect on the state of the system. It becomes possible to explore some situations in which actions do not have the expected behaviour. Using PVSio does not allow the exhaustive analysis possible through model checking. Model checking provides a valuable alternative approach to analysis of this kind, see for example Bolton *et al.* [2013] and Harrison *et al.* [2015]. The goal at this stage is to establish a first impression about the model and to reveal any obvious problems, before more exhaustive analysis is carried out.

The first step in evaluating the specification in PVSio is to introduce auxiliary functions to enable the display of the specification's data structures as intelligible string representations. This is necessary for user-defined data types, as the current distribution of PVS supports automatic conversion into strings only for basic data types (integer, reals, enumeration, etc.). As an example, Listing 1.18 shows the state of the pill dispenser after a sequence of actions have been carried out. The auxiliary function necessary for displaying the data structures relative to the patient can be seen in Listing 1.16 (the full specification can be found in our repository[2]). Hence, the effect of function `patient_type2string` is to transform a literal of type `patient_type` into a string. The fields of the structure type are identified by appropriately named strings (e.g. string `p_name` identifies the value of the corresponding type attribute, where `pt`p_name` is converted into a string using a similar conversion function). The function uses a built-in operator + provided by the PVS language to concatenate strings. The keyword `CONVERSION` at the bottom of the function definition instructs PVS to use `patient_type2string` whenever a translation to string is necessary for `patient_type`.

The auxiliary function for converting the state of the model into a string is shown in Listing 1.17.

An example evaluation of the specification, using PVSio, is now illustrated. As part of the check that the specification is plausible, a sequence of actions is constructed manually

```
1  patient_type2string(pt: patient_type): string =
2     "(# p_name := " + pt`p_name + newline +
3       " p_fields := " + pt`p_fields + newline +
4       " scripts_index := " + pt`scripts_index +
5                                        newline +
6       " scripts := " + pt`scripts + "#)"
7    CONVERSION patient_type2string
8
```

LISTING 1.16. Grounding `patient_type`.

```
1   print_state(st: state): string =
2  LET ans = "(# " + newline,
3  ans = ans + "mode := " + st`mode + newline,
4  ans = ans + "path_mode := " + st`path_mode +
5      newline,
6  ans = ans + "path_mode_set := " +
7      st`path_mode_set + newline,
8  ans = ans + "vis_field := " + st`vis_field +
9            newline,
10 ans = ans + "sel_field := " +
11     st`sel_field + newline,
12 ans = ans + "ent_field := " + st`ent_field +
13     newline,
14 ans = ans + "action := " + st`action +
15     newline,
16 ans = ans + "patient_id_line := " +
17     st`patient_id_line + newline,
18 ans = ans + "med_id_line := " +
19     st`med_id_line + newline,
20 ans = ans + "script_line := " +
21     st`script_line + newline,
22 % continues the rest of the state...
23      IN ans + " #)"
```

LISTING 1.17. Displaying the state using PVSio.

that generates patient and medication databases (this sequence will be abbreviated as `init11`) and can be found in the file main.pvs held in the repository. Simulating this sequence, using PVSio (using the `print_state` function) shows the elements of the state produced.

The fragment of the state produced by using the function (Listing 1.18) focuses on the visible (that is displayed) parts of the state produced by executing the sequence represented by `init11`. The current mode (`db_med_list`, Line 2) shows the medication database list. It displays the first five medication items as indicated by the attribute `med_id_line` ('1' identifies the name of first visible item and '5' the fifth). `patient_id_line` indicates the display of up to five patients (a value `0` indicates no display on the corresponding line). `script_line` shows prescriptions for the current patient. Here `FALSE` indicates that a prescription line is not displayed. The path mode that has been set by entering the relevant password is identified by the attribute `path_mode_set` is `meds_path` (Line 4), which is what it ought to be as indicated by the attribute `path_mode` (Line 3). Three actions are shown as available (`key1`, `key2` and `create`); no data entry fields are visible. Up and down actions are also available in this mode but at this stage of the design it is not yet clear how these actions will be represented in the interface.

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

```
"(#
mode := db_med_list
path_mode := meds_path
path_mode_set := meds_path
vis_field := {  }
sel_field := {  }
ent_field := {  }
action := { key1 key2 create  }
patient_id_line := { l(4) := 0 l(3) := 0 l(2) := 0
                           l(1) := 0 l(0) := 0 }
med_id_line := { l(4) := 5 l(3) := 4 l(2) := 3
                      l(1) := 2 l(0) := 1 }
script_line := { l(4) := FALSE l(3) := FALSE l(2) :=
    FALSE
                     l(1) := FALSE l(0) := FALSE }
patients_db := (: ...  :) % contains the details of
    each patient
meds_db := (: ...  :) % contains the details of each
    medication
% detail omitted...
 #)"
```

LISTING 1.18. Displaying the effect of the sequence `init11`.

PVSio provides a valuable means of testing the specification, evaluating whether it behaves as required. One feature that can be explored of this specification (which will be further analysed in Section 7.2) is *reversibility*. The property is considered in the context of the sequence of actions mentioned earlier that builds a database of both patients and medications (`init11`). This sequence constructs a database that contains more than five medications. The sequence can be used to test whether scrolling down the visible medication list (as specified by `med_id_line`) in display mode (`db_med_list`) followed by scrolling it up produces the same display as before the scrolling actions were taken. This can be demonstrated by considering `editdown0`, which adds a scroll down action.

```
editdown0: state = LET st = init11, st =
scroll_down_med_list(st) IN st
```

```
"(#
mode := db_med_list
...
med_id_line := { l(4) := 6 l(3) := 5 l(2) := 4
                        l(1) := 3 l(0) := 2 }
...
#)"
```

LISTING 1.19. Displaying the effect of the sequence defined by `editdown0`.

The resulting action of scrolling down is followed by scrolling up (see Listing 1.20):

```
editdownup0: state = LET st = editdown0,
st = scroll_up_med_list(st) IN st
```

PVSio therefore makes it possible to check that the behaviour of the model coincides with the expected behaviour for this one case. It does not provide, however, a representation that is easily communicable to domain experts, who are unlikely to have a background in formal modelling. For this, a prototype representing the system would be more useful, while

```
"(#
mode := db_med_list
...
med_id_line := { l(4) := 5 l(3) := 4 l(2) := 3
                        l(1) := 2 l(0) := 1 }
...
#)"
```

LISTING 1.20. Displaying the effect of the sequence defined by `editdownup0`.

at the same time providing the means to do usability evaluation of the prototype. This is illustrated in the next section.

6. CREATING A REALISTIC PROTOTYPE FROM THE SPECIFICATION

An interactive prototype is constructed that is driven by the formal specification described in the previous section. 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 to 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 1.21 shows a snippet of JavaScript code created manually to produce the *home* button of the prototype. The code uses a library of widgets provided by PVSio-web, which are designed to simplify the task of linking model behaviours with graphical user interface elements. `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 `click_act(key1, st)`.

The *second argument* of the constructor 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, which 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

FIGURE 4. Pillbox prototype based on design image.

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

LISTING 1.21. Creation of a touchscreen button using PVSio-web.

associated with the button generates a new system state, as well as information about 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).

This enhanced 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. This prototype provides a functional representation of the modelled design that may be submitted to user evaluation.

7. PROVING PROPERTIES OF THE MODEL

The enhanced version of the model now includes specification of actions and their effects insofar as they are relevant to the use of the device. A complementary analysis of the second iteration can be achieved by assessing the model against use-centred requirements [Harrison *et al.*, 2019a]. By this means, a more exhaustive analysis of the emerging design can be achieved than would be possible with the functional prototype typically used in user-centred design. It also supports software engineering of the system using a spiral model and the mapping of a requirements specification including use-centred requirements [Sommerville, 2010].

When discussing the first sketch prototype (Section 3.3), the principle that the pathways should be mutually exclusive was explored. In the case of the sketch model, there were many cases where the property failed. This requirement was partly a security requirement but it could also be considered to be a use-related requirement, concerned as it is with the integrity of use paths. It will be returned to in Section 7.3. Before considering this requirement, two use-centred requirements, *consistency* and *reversibility*, will be considered. A preliminary consideration of reversibility was described using the PVSio based simulation in Section 5.

7.1. Consistency

An important aid to the use of an interactive system is that a named action should have consistent behaviour irrespective of the mode in which the action is invoked. Examples in the context of this design are the 'ok' and 'quit' actions. A suggested requirement is that the 'ok' action always leaves the mode which immediately precedes it and updates state, while 'quit' leaves the mode and never updates. These requirements are examples of *action consistency* defined in Harrison *et al.* [2019a] as follows:

Action Consistency

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

The action consistency property is formulated as a property of either a single action or of a group of actions (they will be referred to as *Act*), which may exhibit similar behaviours. A relation $\varphi : C \times C$ expresses the intended notion of consistency connecting a filtered state, before an action occurs (captured by *pre_filter* $:\sim S \times MS \rightarrow C$), with a filtered state after the action (captured by *post_filter* $:\sim S \times MS \rightarrow C$). *MS* refers to possible modes that limit the validity of the filter. In the case which is being considered, *C* is a subtype of the state type. Both *pre_filter* and *post_filter* can be expressed as follows: `filter: state -> [# meds_db: med_db_type, patients_db: patient_db_type #]` and the *guard* is `true`, in other words there is no guard. The relation φ is equality and a single action is considered `quit`.

The consistency property can be formulated in a PVS theorem as follows:

```
attemp_quit_consistency_thm:
 THEOREM FORALL (st: state):
  LET st1 = select(quit, st)
   IN (st`meds_db = st1`meds_db AND
       st`patients_db = st1`patients_db)
```

This formulation uses a definition of st1 as the state after the quit action has been applied. It also removes the explicit definition of filter and guard. Attempts to prove the theorem, using the PVS theorem proving assistant, over the enhanced specification produces the following result when applying the general proof command grind:

```
Rule? (grind)
Trying repeated skolemization,
instantiation, and if-lifting,
this simplifies to:
attempt_quit_consistency_thm.2.1 :
{-1} st!1`action(quit)
{-2} creation_success?(mode(st!1))
  |- - - - - - -
{1} st!1`patients_db =
    p_insert(st!1`p_current, temp_patient
             (st!1), patients_db(st!1))
Rule?
```

The command grind fails to find a proof and offers a consequent ({1}) to be proved in which a new patient record has been inserted in the patient database. It is clear that, in general, this new state is not equivalent to the state before quit was applied. The condition that grind offers that it cannot prove involves the following antecedents:

- {-1} the action that is selected in the new state is quit.
- {-2} the mode of the new state is creation_success.

This formulation indicates that the mode of the state signifies successful creation of a new patient record. This occurs in the design at the end of a sequence in which a patient is recording a thumb print for future use in accessing their medicine. This design decision breaks consistency in the user interface. The action consistency theorem can be proved when the guard of the consistency property is changed to (mode(st) /= creation_success) (see Listing 1.22). Therefore, this indicates that the quit action satisfies the consistency requirement in all other device states. A decision for the analyst therefore is whether this isolated inconsistency is acceptable. This issue is one that would then be a matter for further consideration—should the design change or would a user have no issue with this inconsistency? There may be good reasons for keeping the design as is.

```
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)
```

LISTING 1.22. The modified consistency theorem.

There are many properties of this version of the design that relate to its consistency. It is relatively common that actions are inconsistent in some detail. A general requirement of the emerging design may be that all actions designed to change mode that may change one of the databases should be *consistent* so that the user is not confused as to the effect of the action. Hence, any use of the ok action will change the relevant database, while any use of the quit action will leave the relevant database unchanged.

7.2. Reversibility

An initial consideration of reversibility can be seen in the discussion of plausibility (in Section 5) using PVSio. Reversibility is typical in applications that require number entry, for example. A general reversibility property is identified in the reversibility template formulated in Harrison *et al.* [2019a]. The template specifies that for a group of actions $Act \subset S \rightarrow S$, the fact that action $a \in Act$ is reversed by action $b \in Act$, subject to constraints specified by $guard : S \rightarrow \mathbf{B}$ and a $filter : S \rightarrow C$ (where these constraints relate to the mode in which the actions are used) is expressed as follows:

Reversibility

$$\forall s \in S : guard(s) \implies filter(b(a(s))) = filter(s)) \quad (2)$$

In the case of the emerging design, reversibility is a requirement of any action that scrolls up and down the list of medications and patients. There are typically more medications and patients than would fit to a display. For this reason, the proposed design requires four actions: scroll_up_patient_list and scroll_up_med_list and their inverses scroll_down_patient_list and scroll_down_med_list. In modes where lists of medications, patients and prescriptions associated with patients are visible (that is when the actions are permitted), it should be a natural process to move up and down the list. The reversibility template can be used to analyse these scroll characteristics. In the case of the patient list, a *guard* is required namely mode(st) = patient_list and in the case of the medications list mode(st) = db_med_list. In this case, filter describes the whole state, which is the whole state of the pill dispenser should be unchanged. These two consistency properties are combined as a single function confirm_ud_scroll_fn (see Listing 1.24).

Proving the theorem associated with this property requires a structural induction similar to that used in the pathway_mode_thm (Listing 1.3) that was proved for the first iteration of the model. A new transition relation is required in the case of the enhanced model that involves a richer set of transitions, see extract of the full definition in Listing 1.23.

```
state_transitions(pre, post: state): boolean =
 (pre`action(key1) AND post=select(key1, pre)) OR
 (pre`action(key2) AND post=select(key2, pre)) OR
 (pre`action(key3) AND post=select(key3, pre)) OR
 (pre`action(confirm) AND post=select(confirm, pre)) OR
 ...
```

LISTING 1.23. An extract from the state transition function.

This formulation of the theorem requires that the initial state of the device and all states that can be reached from this state have the required property. The initial state of the model satisfies the predicate: `(init?(pre))`.

```
%-- 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)
%-- 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))
```

LISTING 1.24. Reversibility of scroll actions.

The theorem, as formulated, is proved true of the design, as can be seen by rerunning the theorem in pillboxchecks.pvs in the repository as previously cited.

7.3. Pathway exclusivity

The last requirement considered in this paper revisits *path exclusivity*. This requirement was visited in the first iteration of the model (Listing 1.3). In the case of the sketch model, there were many cases where the property fails to be true. The informal version of the requirement, that is repeated below, requires more detail than is provided in the sketch model.

(i) The doctor/carer should be able to see medication details but not update the meds database.
(ii) The designated pharmacist should not be able to see patient details.

The approach to proof is similar to that attempted in the case of the first version of the model. These two paths are checked in two theorems. The first requires that only modes that are designated `meds_path` mode are entered when the `path_mode_set` attribute is also `meds_path`. It uses `path_function` (redefined to deal with the change of mode names, see Listing 1.25) to assert that the path mode for the state before the action is appropriate.

This theorem is found in Listing 1.26.

Proving this theorem also assumes an induction process. When a device is in a state that should be part of the meds

```
path_function(m: mode_type): path_mode_type =
  COND
  m = initial -> no_path,
  m = pwd -> scripts_path,
  m = db_pwd -> meds_path,
  m = db_menu -> meds_path,
  m = patient_list -> scripts_path,
  m = db_med_list -> meds_path,
  m = new_patient_details -> scripts_path,
  m = patient_details -> scripts_path,
  m = db_med_details -> meds_path,
  m = new_med -> meds_path,
  m = patient_scripts_list -> scripts_path,
  m = patient_script -> scripts_path,
  m = new_patient_script -> scripts_path,
  m = dose -> scripts_path,
  m = new_dose -> scripts_path,
  m = scan -> scripts_path,
  m = scan_enabled -> scripts_path,
  m = creation_success -> no_path
  ENDCOND
```

LISTING 1.25. Path function.

```
med_path_check: THEOREM
      FORALL (pre, post: state):
         ((path_mode_set(pre) = meds_path) AND
           (path_function(pre`mode) = meds_path)
          AND (pre /= post) AND
         state_transitions(pre, post) AND
         (mode(post) /= pwd)) IMPLIES
          ((path_mode(post) = meds_path) OR
          (path_mode(post) = no_path))
```

LISTING 1.26. Pharmacist's pathway theorem.

path (`path_function(pre`mode) = meds_path`) then the medications password should have been previously entered (`path_mode_set(pre) = meds_path`). Any transition will either lead to continue in the meds path or to leave the meds pathway completely.

The second theorem allows the designated doctor/carer to view patient and medicine details but does not allow them to update the meds database. The theorem in this case is defined in Listing 1.27. The theorem requires that the meds database is untouched in the scripts path.

```
 script_path_check: THEOREM
    FORALL (pre, post: state):
      (path_mode_set(pre) = scripts_path AND
      state_transitions(pre, post)) IMPLIES
        (meds_db(pre) = meds_db(post))
```

LISTING 1.27. Designated doctor/carer pathway theorem.

This version of the theorem fails and it is necessary to introduce additional constraint in the `select` function (see Listing 1.9) so that selection of a field is prevented when it would conflict with the pathway requirement, see Listing 1.28. In the case of the doctor/carer pathway, it is necessary to modify the select field function to prevent field selection when in the `db_med_details` mode.

```
select(f: fields_type, st: state): state =
  IF vis_field(st)(f) AND
     NOT ((mode(st) = db_med_details) AND
        (path_mode_set(st) = scripts_path)) THEN
   st WITH
     [sel_field :=
      LAMBDA(x: fields_type):
        x = f
      ]
   ELSE st ENDIF
```

LISTING 1.28. Select field function.

This means that the user can see the medicine details but cannot update the device. Both theorems, representing the two pathway requirements, are true of the specification.

8. RELATED WORK

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. In Furniss *et al.* [2014] and Masci *et al.* [2015a], formal methods have been integrated with contextual enquiry with the aim of supporting the work of a field investigator (as opposed to supporting the development of a device). Written notes provided by the field investigator described workflows carried out by clinicians in a hospital. This included how information resources are communicated and transformed throughout the socio-technical system. The notes were manually translated to a PVS model by a formal methods expert, and formal analysis was used to identify gaps and weaknesses that could warrant further investigation.

In Masci *et al.* [2012], formal methods were used to support incident investigation methods. Key aspects of an incident report were modelled in PVS with the aim of analysing claims and hypotheses described in the report. In Masci *et al.* [2017], formal methods were used to extend a standard hazard analysis process and promote rigorous specification of safety requirements.

Bowen & Reeves [2017] 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. It is not clear, however, that executable versions of their models have been developed. Haesen *et al.* [2011] 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 *et al.* [2014], Mori *et al.* [2002] and Fields [2001] combine task and functional models. Martinie *et al.* [2011] combine visual, functional and task elements.

9. CONCLUSION

The process of user-centred design described in the paper is iterative. The initial non-functional design is based on the initial interviews with, or observations of, participants in the work environment. In this case, it would be envisaged that patients, carers, pharmacists and doctors would be involved in this initial process. The initial non-functional design would be evaluated both by the participants and then by considering the preliminary requirements (for example, in this case, that the pathways are mutually exclusive).

Subsequent iterations of the specification will generate a functional prototype that can be subjected to further user evaluation. The requirements that have been developed through the previous stages of the process will be proved of these iterations and as features are added or issues are discovered in the use of the design further requirements will be added.

The final result will be a design that has been tested with users and provably satisfies usability requirements. It will be a design that is based on a specification that will additionally support safety and security analysis.

Two iterations of the design are described in this paper. The model of the first iteration, the sketch design, was produced automatically from an initial sketch. A single property, the pathway property, was proved of that model. The second iteration involved a specification consisting of 83 functions, analysed using 26 PVS theorems, which were proved within on average 3 seconds. The PVS system was installed on an Apple Macbook Pro with a 2.9 GHz Intel Dual Core i5. The conversion functions required to animate the specification involved 189 lines. Similar much larger specifications have been used to model a variety of full-scale interactive systems, for example Harrison *et al.* [2019a,b]. In the case of Harrison *et al.* [2019b], a model of the controller for a neonatal dialysis machine was developed. Safety properties were checked in meetings involving the developers and a formal analyst. In many cases where a property failed, the model or the property could be adjusted, and appropriate rationale developed, within the meeting.

A future dimension of this 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. The process of using PVSio-web is being simplified to construct prototypes from models. The aim is to make these techniques accessible to a wider group of developers.

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. It is hoped that analysis, as an adjunct to the techniques and approaches of user-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 such as the one used here for illustration purposes. However, the safety of medical devices, in particular, is crucial and a thorough analysis of usability issues is a key contribution ensuring safety.

Acknowledgments

Nuno Rodrigues, João Vilaça and Nuno Dias from IPCA (Polytechnic Institute of Cavado and Ave) developed the first prototype of the pill dispenser.

Funding

This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020.

REFERENCES

Beyer, H. and Holtzblatt, K. (1998) *Contextual Design: Defining Customer-Centred Systems*. Morgan Kaufmann.

Bolton, M., Jiménez, N., van Paassen, M. and Trujillo, M. (2014) Automatically generating specification properties from task models for the verification of human-automation interaction. *IEEE Trans. Hum. Mach. Syst.*, 44, 561–575.

Bolton, M. L., Bass, E. and Siminiceanu, R. (2013) Using formal verification to evaluate human-automation interaction, a review. *IEEE Trans. Syst. Man Cybern. Part A Syst. Humans*, 99, 1–16.

Bowen, J. and Reeves, S. (2017) Combining Models for Interactive System Modelling. In Weyers, B., Bowen, J., Dix, A. and Palanque, P. (eds), *The Handbook of Formal Methods in Human-Computer Interaction*, pp. 161–182. Springer International Publishing, Cham.

Carroll, J. (ed.) (1995) *Scenario Based Design: Envisioning Work and Technology in System Development*. Wiley.

Fields, R. E. (2001) Analysis of erroneous actions in the design of critical systems. PhD Thesis, Department of Computer Science, University of York, Heslington, York, YO10 5DD.

Furniss, D., Masci, P., Curzon, P., Mayer, A. and Blandford, A. (2014) 7 themes for guiding situated ergonomic assessments of medical devices: a case study of an inpatient glucometer. *Appl. Ergon.*, 5, 1668–1677.

Haesen, M., Van den Bergh, J., Meskens, J., Luyten, K., Degrandsart, S., Demeyer, S. and Coninx, K. (2011) Using Storyboards to Integrate Models and Informal Design Knowledge. In Hussmann, H., Meixner, G. and Zuehlke, D. (eds), *Model-Driven Development of Advanced User Interfaces*, pp. 87–106. Springer Berlin Heidelberg, Berlin, Heidelberg.

Harel, D. (1987) Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.*, 8, 231–274.

Harrison, M., Campos, J. and Masci, P. (2015) Reusing models and properties in the analysis of similar interactive devices. *Innov. Syst. Softw. Eng.*, 11, 95–111.

Harrison, M., Masci, P. and Campos, J. (2018) Formal Modelling as a Component of User Interface Design. In Mazzara, M., Ober, I. and Salaün, G. (eds), *Software Technologies: Applications and Foundations STAF 2018 Collocated Workshops (Revised Selected Papers)*. Lecture Notes in Computer Science, vol. 11176, pp. 274–294. Springer.

Harrison, M., Masci, P. and Campos, J. (2019a) Verification templates for the analysis of user interface software design. *IEEE Trans. Softw. Eng.*, 45, 802–822.

Harrison, M. D., Freitas, L., Drinnan, M., Campos, J. C., Masci, P., di Maria, C. and Whitaker, M. (2019b) Formal techniques in the safety analysis of software components of a new dialysis machine. *Sci. Comput. Program.*, 175, 17–34.

Harrison, M. D., Masci, P., Campos, J. C. and Curzon, P. (2017) Verification of user interface software: the example of use-related safety requirements and programmable medical devices. *ACM Trans. Hum. Mach. Syst.*, 47, 834–846.

Martinie, C., Palanque, P., Barboni, E., Winckler, M., Ragosta, M., Pasquini, A. and Lanzi, P. (2011) Formal tasks and systems models as a tool for specifying and assessing automation designs. In *Proc. 1st int. conf. application and theory of automation in command and control Systems, ATACCS '11*, pp. 50–59. IRIT Press, Toulouse, France.

Masci, P., Ayoub, A., Curzon, P., Harrison, M., Lee, I., Sokolsky, O. and Thimbleby, H. (2013) Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. In *Proc. ACM symposium engineering interactive systems (EICS 2013)*, pp. 81–90. ACM Press.

Masci, P., Curzon, P., Furniss, D. and Blandford, A. (2015a) Using pvs to support the analysis of distributed cognition systems. *Innov. Syst. Softw. Eng.*, 11, 113–130.

Masci, P., Huang, H., Curzon, P. and Harrison, M. D. (2012) Using PVS to Investigate Incidents Through the Lens of Distributed Cognition. In Goodloe, A. E. and Person, S. (eds), *NASA Formal Methods*. Lecture Notes in Computer Science, vol. 7226, pp. 273–278. Springer, Berlin Heidelberg.

Masci, P., Oladimeji, P., Zhang, Y., Jones, P., Curzon, P. and Thimbleby, H. (2015b) PVSio-web 2.0: Joining PVS to HCI. In Kroening, D. and Păsăreanu, C. S. (eds), *Computer aided verification: 27th int. conf. CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I*, pp. 470–478. Springer International Publishing, Cham.

Masci, P., Zhang, Y., Jones, P. and Campos, J. C. (2017) A hazard analysis method for systematic identification of safety requirements for user interface software in medical devices. In *15th int. conf. software engineering and formal methods (SEFM 2017)*, pp. 284–299. Springer.

Monk, A., Wright, P., Haber, J. and Davenport, L. (1993) *Improving Your Human-Computer Interface: A Practical Technique*. Prentice-Hall.

Morgan, C. C. (1994) *Programming from Specifications* (2nd edn). Prentice-Hall International.

Mori, G., Paternò, F. and Santoro, C. (2002) CTTE: support for developing and analyzing task models for interactive system design. *IEEE Trans. Softw. Eng.*, 28, 797–813.

Muñoz, C. (2003) Rapid Prototyping in PVS. Technical Report NIA Report No. 2003–03, NASA/CR-2003-212418, National Institute of Aerospace.

Nielsen, J. and Molich, R. (1990) Heuristic evaluation of user interfaces. In Chew, J. and Whiteside, J. (eds), *ACM CHI proc. CHI '90: empowering people*, pp. 249–256.

Owre, S., Rushby, J. and Shankar, N. (1992) PVS: A prototype verification system. In Kapur, D. (ed.), *Eleventh int. conf. automated deduction (CADE)*. Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer.

Sommerville, I. (2010) *Software Engineering*. Addison-Wesley.

Watson, N., Reeves, S. and Masci, P. (2018) Integrating user design and formal models within PVSio-Web. In *Workshop on formal intergrated development environment (F-IDE-18). Electronic Proc. theoretical computer science (EPTCS)*.