The specification and analysis of use properties of a nuclear control system

M. D. Harrison, P. M. Masci, J. Creissac Campos and P. Curzon

Abstract The chapter explores a layered approach to the analysis of the Nuclear Power Plant Control System described in Chapter 4. A model is specified to allow the analysis of use-centred properties based on generic templates. User interface properties include: the visibility of state attributes, the clarity of the mode structure and the ease with which an action can be recovered from. Property templates are used as heuristics to ease the construction of requirements for the control system interface.

1 Introduction

Formal modelling can offer substantial benefits when developing an interactive system. It enables systematic clarification of assumptions made about a design and supports verification that specified requirements have been met. This paper considers the Nuclear Power Plant Control System described in Chapter 4. The three use cases introduced in the book offer slightly different perspectives that might suggest different approaches to analysis. Broadly, analysis approaches may be classified as task orientated or based on characteristics of the interface. In the first category there are informal approaches, for example Cognitive Walkthrough [Polson et al., 1992], and formal approaches such as those of [Bolton et al., 2012]). These approaches

M. D. Harrison
School of Computing Science, Newcastle University, Newcastle upon Tyne, UK e-mail: michael.harrison@ncl.ac.uk

J. Creissac Campos and P. M. Masci
Dep. Informática / Universidade do Minho & HASLab / INESC TEC, Braga, Portugal, e-mail: jose.campos@di.uminho.pt,paolo.masci@inesctec.pt

P. C. Curzon
EECS, Queen Mary University London, Mile End Road, London e-mail: p.curzon@qmul.ac.uk

are concerned with the representation of the intended task and then to analyse the system that is intended to perform the task. In the second category, analysis may be based on the characteristics of the interface (for example, Heuristic Evaluation [Nielsen and Molich, 1990] and formal approaches such as those of [Campos and Harrison, 2009]). These approaches focus on, for example, the visibility or perceivability of key attributes of the device and analyse properties of the supported actions (for example, their predictability or undoability). The approach taken in this paper supports both styles of analysis. In the case of the task approach, the focus is on the constraints that determine the activities that the user performs, rather then focusing on prescribed normative behaviours. Constraints include the visibility of information (for example function key displays) that help the user to decide what action to take next.

A modelling approach based on layers of specification is designed to unify these two approaches to analysis, with the aim of maintaining the integrity of the specification. Analysis of the interactive system is facilitated by the use of property templates.

The chapter is organised as follows. Section 2 describes the features of the example that are relevant to illustrating the analysis. Section 3 discusses the structure of the model that describes the interactive behaviour of the system. Section 4 describes the tools, including the set of property templates, that are used to drive the analysis. Section 5 details the model of the example and describes the process of instantiating the property templates to be theorems over the model. Finally we describe related work (Section 6) and conclusions (Section 7).

2 The use case

In the present example two analytic perspectives are taken.

- How well does the interface support operating procedures[1] developed to help the controller start up or close down the system?
- Is the operator able to monitor and make appropriate adjustments to the process? Is there sufficient information for operators to understand what is happening and are suitable actions visible and available?

These two perspectives require different styles of analysis. The first is concerned with how effectively the display, and the actions it supports, can be invoked as required by an operator who is following the start-up and close-down operating procedures. The second is concerned with the display, the graphics, the status display, the sliders, the enabled actions and how these change the display and reflect the state of the underlying process. Whatever the level of analysis of the user interface, it is important to understand the interface to the underlying system. The interface of the system should aid understanding: by making parts of the underlying process visible

[1] `http://www.hci-modeling.org/nppsimulator/BWRSimulationDescription.pdf`

to the user; producing visible feedback to enable the operators to assess what has been done. Interactive systems of any complexity have a common characteristic that some elements of the state of the system are perceivable (for example, visible or audible), and that user actions transform the state [Duke and Harrison, 1993]. Furthermore, not all actions are permitted all of the time, and the behaviour of actions can depend on distinguished state attributes called *modes*, see [Gow et al., 2006] for further discussion. The modes in this case determine, for example, whether the control rods are being controlled automatically or manually. Modes also determine specific interactions relating to the behaviour of the mouse: its position and whether the mouse button is pressed or not. For example when the mouse button is pressed and the cursor position coincides with a slider on the screen, and the slider is not in automatic mode, then dragging the cursor moves the position of the slider thus changing the relevant behaviour of the component of the process that it represents.

Users have difficulty understanding the progress of a system when elements of the state of the system, that are relevant to that understanding, are not visible in a form that makes sense to them. At the same time, confusion can arise when actions relevant to the current activity are, apparently or actually, disabled by the system, or when the actions have an unexpected or inconsistent effect with respect to the users' knowledge and experiences of the system. Actions and states are therefore elemental in understanding interactive behaviour. Modes are also important. It is unusual that an interactive system is so simple that actions always have the same effect.

To achieve the goals and activities required of the users, most interactive systems are designed more or less effectively to ensure that the information required (we call them *information resources* [Campos et al., 2014]) are made explicitly available, and in a form that can be easily understood by the users. A role of a model of the interactive system is therefore to make these information resources explicit so that assumptions about the constraints they impose may be analysed.

3 Structure of the models

It is important to distinguish between interactive systems and the components of interactive systems. Interactive systems are socio-technical systems involving people, devices, and artefacts (desks, pieces of paper, pens, tablets and so on). The primary focus of the modelling approach illustrated here is of the interactive devices that are the components of the interactive system. The presented property templates capture aspects of the system that can facilitate device-user interaction.

3.1 The interface specification

The specification of an interactive system includes a definition of the set of actions, including user actions, that are possible within them. These actions affect and are

affected by the state of the system. The behaviour of actions is often determined by the mode of the device. The proposed model of the interactive system also makes explicit the information resources that are assumed to aid the use of the system. Assumptions about the activities for which the system is designed are also made explicit. An action is a transformation supported directly by the interface. An activity is a means to achieve some work goal, for example achieving a steady state of the system with maximum voltage.

The interface specification describes what the display shows and captures the effects of user level actions. The display will show some features of the state of the reactor, these features may be encoded as part of the interface. It will also show the user actions that are translated into actions within the reactor. The specification includes display widgets showing simple status information. These include widgets labelled RKS, RKT, KNT, TBN, $WP1$, $WP2$, CP, AU. These displays are associated with a range of colours indicating status. The display also shows actions associated with the valves: $SV1$, $SV2$, $WV1$, $WV2$ and sliders that change the position of the control rods and the status of the valves.

Analysis of an interactive device is then concerned with proving that relevant feedback is given on completing an action, that relevant information is available before an action is carried out, that it is possible to recover from an action in specified circumstances, that it is always possible simply to step to some home mode whatever the state of the device and that actions can be completed consistently.

3.2 Structuring specifications

The model of the interactive system is structured as four layers. The first layer simply specifies the constants and types used throughout the specification. It includes types relating to the devices involved and the entities that are in the broader system. For example, in the case of the reactor these types would include notions such as *pressure*, *volume* and *temperature*. There would also be types associated with pumps and valves. Constants would include maximum and minimum values required to trigger error events in certain situations.

The second layer describes assumptions about the underlying process, managed or controlled by the devices that are required to enable the analysis of the characteristics of the interactive device. This layer is often reused across families of device models when exploring the effects of differing user interfaces. For example, in [Harrison et al., 2015a], different brands of IV infusion devices share the same pump layer. The process layer, in the case of the nuclear process, is the simplest model of the nuclear reactor that will allow a proper consideration of interactive behaviour of the control system. A specification of the underlying reactor, describing the details of the relation between reactor core and turbine, would include attributes defining water level and pressure for each. The specification at this level would also define the characteristics of the pumps and valves. The pumps would be associated with rates per minute and the valves would be on or off. A number of actions will be spec-

ified at this level. An action *tick* is used to represent the interval of one minute and update the attributes to describe the evolving process. There will be further actions switching pumps on and off, opening and closing valves and changing the value of flow in the pump, for example.

The third layer describes the interface to the interactive device or system. This model uses the process description described in the second layer. It makes those aspects of the state that are visible explicit through the interface. It describes the user actions, including for example how the sliders or buttons or other display widgets work. The third layer of the specification of the nuclear power plant control user interface specifies how the user sets, controls and views the operation of the reactor. It is specific to this particular interface, whereas the reactor specification (given in the second layer) may be more generic and therefore used with several user interfaces. It provides opportunity to explore the variety of user interfaces that may be appropriate for supporting human-machine interactions necessary to control the reactor.

The fourth, and final, layer makes explicit the information resources that are required for different actions in different circumstances. It captures constraints on action based on the goals and activities that the user achieves [Campos et al., 2014]. This layer contains an interactive system view. The activities and actions are "resourced" by user interfaces for devices that are used in the interactive system or, indeed, any other source of relevant information that is present within the interactive system. It adds attributes that are not captured by the devices and includes (meta-)actions that describe activities that may involve actions of the interactive devices. An example of this fourth layer used in a different context can be found in [Masci et al., 2012].

The models to be considered have some or all of the following characteristics depending on layer.

- A set of *actions* $a : A = S \longmapsto S$ where S is a set of states. Actions are partial functions. They are made total by including a value "undefined" ($\perp$). A permission function *per* takes an action and determines whether it is defined for a value in its domain $per : A \to (S \to T)$ such that $per(a)(s) = true$ if $a(s) \neq \perp$.
- A *state* is a set of attributes. Functions of the form $filter : S \to C$ where C is an attribute will often be used to extract an attribute of the state. The attribute is itself a domain, for example temperature or pressure. Similarly, some elements of the state are part of the interface and are perceivable. p_filter will often be used to describe the filter that extracts the corresponding visual attribute to the value extracted by $filter$. Alternatively a predicate $vis\_filter : C \to T$ may be used to assert that the value of the filtered attribute is visible.
- The function *mode* is a particular form of $filter$, namely $mode : S \to MS$. It extracts the modes of the model, where *MS* is an attribute that ranges over a set of modes. In the example, one set of modes relates to the types of variable being entered through number entry.

4 Tool support

Two approaches to specification and proof are feasible for the kinds of model described here: model checking and theorem proving. The theorem proving approach is appropriate here because a potentially important feature of the analysis, not discussed further in this short chapter, concerns the mechanisms for number entry. Since the domain of numbers is relatively large, proof using model checking can result in analyses of very large models that can be intractable. Interested readers are redirected to [Harrison et al., 2015b] for example applications of our layered approach to number entry.

4.1 Representing and proving the model

The automated theorem prover used is the *Prototype Verification System (PVS)* [Shankar et al., 1999]. The system combines a specification language based on higher-order logic with an interactive prover. PVS has been used extensively in several application domains. The higher-order logic supports the usual basic types such as `boolean`, `integer` and `real`. New types can be introduced either in a declarative form (these types are called *uninterpreted*), or through *type constructors*. Examples of type constructors used in the present specification are function and record types. Function types are denoted `[D -> R]`, where `D` is the domain type and `R` is the range type. Predicates are Boolean-valued functions. Record types are defined by listing the field names and their types between square brackets and hash symbols. Predicate subtyping is a language mechanism used for restricting the domain of a type by using a predicate. An example of a subtype is `{x:A | P(x)}`, which introduces a new type as the subset of those elements of type `A` that satisfy the predicate `P`. The notation `(P)` is an abbreviation of the subtype expression above. Predicate subtyping is useful for specifying partial functions. This notion is used to restrict actions to those that are permitted explicitly by the permission predicates mentioned when describing the models in general terms. Specifications in PVS are expressed as a collection of *theories*, which consist of declarations of names for types and constants, and expressions associated with those names. Theories can be parametrised with types and constants, and can use declarations of other theories by importing them. The prelude is a standard library automatically imported by PVS. It contains a large number of useful definitions and proved facts for types, including common base types such as Booleans (`boolean`) and numbers (e.g., `nat`, `integer` and `real`), functions, sets, and lists.

The specification of the models takes a standard form as described in Section 3.2. A model consists of a set of actions and a set of permissions that capture when the actions can occur.

```
action: TYPE = [state -> state]
```

For each action there is a predicate:

```
per_action: TYPE = [state -> boolean]
```

that indicates whether the action is permitted.

4.2 Property Templates

Property templates are generic mathematical formulae designed to help developers to construct theorems appropriate to the analysis of user interface features. The aim is to make these programmable devices more predictable and easy to use. The particular set of templates considered here is derived from [Campos and Harrison, 2008]. A formulation of these properties based on *actions*, *states*, and *modes* is presented, along with a brief summary of the use-related concerns captured by the template. There are two types of property, properties that relate states where a specific action has taken place, and properties that relate a state to *any* state that can be reached by *any action* from that state. The relation $transit : S \times S$ relates states that can be reached by any action. For a particular model $transit$ will be instantiated to connect states by the actions provided by the system.

Completeness. This template checks that the interactive system allows the user to reach significant states in one (or a few steps). For example, being able to reach "home" from any device screen in one step is a completeness property. The completeness template asserts that a user action will transform any state that satisfies a predicate $guard : S \to T$ into another state that satisfies a predicate $goal : S \to T$. The guard is introduced to make it possible to exclude states that may not be relevant.

Completeness

$$\forall s \in S : guard(s) \land \sim goal(s) \\ \Rightarrow \exists a \in A \land per(a)(s) \land goal(a(s)) \quad (1)$$

Feedback. When certain important actions are taken, a user needs to be aware of whether the resulting device status is appropriate or problematic [AAMI, 2010]. Feedback breaks down into *state feedback*, requiring that a change in the state (usually specific attributes of the state rather than the whole state) is visible to the user, and *action feedback*, requiring that an action always has an effect that is visible to the user.

State feedback

$$\forall s1, s2 \in S,\ guard(s1) \land guard(s2) \land transit(s1, s2) \land \\ filter(s1) \neq filter(s2) \\ \Rightarrow p\_filter(s1) \neq p\_filter(s2) \quad (2)$$

Action feedback

$$\forall a \in S \to S,\ \forall s \in S : per(a)(s) \wedge \\ guard(s) \wedge (filter(s) \neq filter(a(s))) \\ \Rightarrow p\_filter(s) \neq p\_filter(a(s)) \quad (3)$$

In the case of state feedback the guard may be used, for example, to restrict the analysis to ensure that the device or system is considered to be in the same mode as a result of the state transition. Variants of the *feedback* properties will also be used that assume separate *visible* attributes are not specified in the model. Instead a relevant predicate $vis\_filter : S \to T$ is linked to $filter : S \to A$. $vis\_filter(s)$ is true for $s \in S$ if $filter(s)$ is visible. Both these variants will be used in Section 5. The choice is based on how the model is constructed.

Consistency. Users quickly develop a mental model that embodies their expectations of how to interact with a user interface. Because of this, the overall structure of a user interface should be consistent in its layout, screen structure, navigation, terminology, and control elements [AAMI, 2010]. The consistency template is formulated as a property of a group of actions $A_c \subseteq \wp(S \to S)$, or it may be the same action under different modes, requiring that all actions in the group have similar effects on specific state attributes selected using a filter. The relation *consistent* connects a filtered state, before an action occurs, with a filtered state after the action. The description of the filters and the *consistent* relation capture the consistency across states and across actions.

Consistency

$$\forall a \in A_c \subseteq \wp(S \to S), s \in S, m \in MS : \\ guard : S \times MS \to T \\ consistent : C \times C \to T \\ filter\_pre : S \times MS \to C \\ filter\_post : S \times MS \to C \\ guard(s,m) \wedge \\ consistent(filter\_pre(s,m), \\ filter\_post(a(s),m) \quad (4)$$

Consistency is a property of, for example, the cursor move actions when the mouse button is pressed. It may be used to prove that while the button is down the move actions will continue to be interpreted in the relevant mode.

Reversibility. Users may perform incorrect actions, and the device needs to provide them with functions that allow them to recover by reversing the effect of the incorrect action. The reversibility template is formulated using a $guard : S \to T$, and a $filter : S \to FS$ which extracts a set of focus attributes of the state:

Reversibility

$$\forall s \in S : guard(s) \Rightarrow \exists b : S \to S : \\ filter(a(b(s))) = filter(s) \quad (5)$$

Some properties simply maintain invariants for any state. Examples of such properties are *visibility* and *universality*. There are alternative formulations of these two properties. The first asserts that a predicate applied to one filtered value is true if and only if an appropriate predicate is true of the other filtered value. The second asserts that a filtering of the first value is equal to the determined filter of the second value. These two formulations are appropriate in different circumstances as will be briefly explored in Section 5. The style of interface described in this case study lends itself particularly to the second option.

Visibility. This property describes an invariant relation between a state variable that is not necessarily visible to the user and a user interface value that is visible to the user. Examples of these properties are: the current operational mode is always unambiguously displayed; a slider that shows the position of the control rods always shows the actual position of the control rods in the underlying process; the colour of the status attribute describes general characteristics of the value of the attribute. $filter(s)$ and $p\_filter(s)$ are the filters for the attribute and its perceivable counterpart.

Visibility

$$\forall s1, s2 \in S : transit(s1, s2) \wedge visible(s1) \Rightarrow visible(s2) \quad (6)$$

where $visible = pred\_filter(s) \Leftrightarrow pred\_p\_filter(s)$
or $visible = filter(s) = p\_filter(s)$

Universality. Universality generalises the visibility property requiring that given two filters of the state: $filter1$ and $filter2$, there are predicates on the filters that are equivalently true.

Universality

$$\forall s1, s2 \in S : transit(s1, s2) \wedge universal(s1) \Rightarrow universal(s2) \quad (7)$$

where $universal = pred\_filter1(s) \Leftrightarrow pred\_filter2(s)$
or $universal(s) = filter1(s) = filter2(s)$

5 Modelling the Nuclear Power Plant Control User Interface

The fragments of specification described in this section were taken from the description to be found in chapter 4 and a simulator (see the simulator description and code[2]) that includes a version of an interface to the nuclear controller. A more thorough description of the user interface was required than was available in the use case material to do a thorough analysis using the property templates. The analysis of IV infusion pumps described in [Harrison et al., 2015b] illustrates the more thorough approach. If such a detailed description had been available then it could be

[2] http://www.hci-modeling.org/nppsimulator/BWRSimulationDescription.pdf

further explored using PVSio-web [Masci et al., 2015] to ensure that assumptions made seem realistic and to demonstrate where properties of the system fail to be true. This would provide confidence that the behaviour of the interface as specified conforms with the expected user experience. The issue of validation of the model of the system is explored in more detail in [Harrison et al., 2014]. The introduction to the use-case contains the following paragraph.

> "The operation of a nuclear power plant includes the full manual or partially manual starting and shut down of the reactor, adjusting the produced amount of electrical energy, changing the degree of automation by activating or deactivating the automated steering of certain elements of the plant, and the handling of exceptional circumstances. In case of the latter, the reactor operator primarily observes the process because the safety system of today's reactors suspends the operator step by step from the control of the reactor to return the system back to a safe state."

The interface involves schematics of the process, the availability of actions as buttons and graphical indications of key parameters, for example temperature and levels. The specification of the model can be layered using the levels described in Section 3 as follows.

5.1 Types and constants

This contains generic definitions that will be used throughout other layers of the theory. It defines types such as:

```
pump_type: TYPE = [# speed: speed_type,
                     on: boolean
                  #]
```

The pump attribute is defined to have a speed of flow and to be either on or off. There are several pumps with the same characteristics as defined by the following function type:

```
pumps_type: TYPE = [vp_number -> pump_type]
```

This type definition allows the definition of multiple pumps indexed by an integer (`vp_number`). This type relates to the process layer. The following types are used in the interface layer.

```
cursor_type = TYPE [* x: x_type,
                      y: y_type *]
```

The type `cursor_type` specifies the type of the cursor on the controller display. It is tied to the physical position of the mouse. The details of how this is done will not be described here. It will be assumed that there is a function `mouse` that extracts the current cursor position of the mouse. The slider which is also found in the interface layer is specified as follows.

```
slider_type : TYPE =
                   [#  ypos: y_type,
                       lx: x_type,
                       rx: x_type,
                       xpos: x_type #]
```

The slider type specifies the current x-position of the cursor when the slider has been selected (`xpos`). It specifies the left and right limits of the slider (`lx` and `rx`) and the y position of the slider (`ypos`). As a simplification for the illustration the slider is assumed to have no depth. In the real system sliders also have a depth and therefore the y-coordinates will also have boundaries.

5.2 The process layer

The process layer describes sufficient details of the underlying process of the nuclear reactor to provide an adequate underpinning for the interface. The interface captures, for example, those situations where the process automates and therefore removes the ability of the operator to change settings manually. The model describes the ongoing process in terms of a single action *tick* that updates attributes of the pump state as time progresses.

```
tick(st: npp): npp =
st WITH
 [time := time(st) +1,
  sv :=
    LAMBDA (n: vp_number):
      COND
        n=1 -> (#
        flow :=
         COND
             sv(st)(1)`on ->
        (reactor(st)`pressure -
             condensor(st)`pressure)/10,
            ELSE -> 0
         ENDCOND,
        on := sv(st)(1)`on
                    #),
        n=2 -> (#
        flow :=
           COND
              sv(st)(2)`on ->
             (reactor(st)`pressure -
               condensor(st)`pressure)/2.5,
              ELSE -> 0
           ENDCOND,
        on := sv(st)(2)`on
                     #)
      ENDCOND,
   poi_reactor :=
      LET num_reactor =
            (old_pos_rods_reactor(st) -
                pos_rods_reactor(st))
       IN (
        COND
            num_reactor >= 0 ->
```

```
                    num_reactor / (time(st) -
                                   time_delta_pos(st)),
              ELSE ->
                     - num_reactor /
                          (time(st) - time_delta_pos(st))
         ENDCOND ),
   bw := (2*(100 - pos_rods_reactor(st))*
                      (900-reactor(st)`pressure))/620,
      ...
      ]
```

This fragment of specification illustrates the form of the process layer. The action that is described is `tick`. Its function is to update the process state (defined by type `npp`) attributes. Some of these attributes can be changed by actions that can be invoked by the operator (for example the two valves `wv(1)` and `wv(2)`) while others are internal to the process (for example `poi_reactor` and `bw`).

Further actions determine transformations of the process that can be invoked directly through the user interface. For example `control_rods` is an action that updates the process state to its new position. This position is determined by where the cursor is, as represented in the control rods slider, when the rod's position is not under automatic control.

5.3 The interface layer

The interface layer describes those attributes of the state of the process that are visible to the user and the actions that can be performed by the operator. The interface presents the state of the process to provide the operator with situation awareness. There are also displayed attributes that indicate sliders and buttons that can be used by the operator to control aspects of the process.

As illustration of the layer consider the mouse actions: *move*, *click* and *release*. The actions *move* and *release* have effects that depend on the modes that the interface is in. The action *click* changes the mode. The change of mode depends on the position of the cursor. Two sets of modes are specified. One set relates to the sliders on the display (`slider_mode`) while the other relates to the actions that are offered by the display (`action`). When the mouse has not been clicked or is over a space in the display that does not correspond to a slider or an action then `slider_mode = nulslimo` and `action = null_action`. When the mouse is clicked then a boolean attribute `clicked` is true. The permission that allows `click` to be an available action is as follows:

```
per_click(st: npp_int): boolean = NOT clicked(st)
```

click is an action that:

1. assigns a value to slider mode if the cursor is at the appropriate y-coordinate (`ypos`) and in the relevant range of x-coordinates for the slider (in reality there

may also be a range of y-coordinates) otherwise it sets the slider mode to the relevant null value.
2. assigns a value to action, the action mode, if the x and y coordinates are within the relevant range of an action button and the action is enabled. If the cursor is outside the defined button ranges then the action is set to null.

Fragments of the action specification are given below. They show the effect when the mouse is within the slider for the pump $wp1$ and the effect when the mouse is in the area of the action button that sets control rods to automatic mode.

```
click(st : npp_int): npp_int =
  LET x = cursor(st)`x AND y = cursor(st)`y IN
  st WITH
  [ slidermode :=
    COND
      y = wp1_slider(st)`ypos AND
      (x >= wp1_slider(st)`lx) AND
      (x <= wp1_slider(st)`rx) AND
                           NOT auto_wp1s(st) -> wp1s,
      ...
      ELSE -> nulslimo
    ENDCOND,
    action :=
    COND
      ((x <= acrarea(st)`lx) AND
      (x >= acrarea(st)`rx) AND
      (y <= acrarea(st)`dy) AND
      (y >= acrarea(st)`uy)) ->
            COND
                auto_cr -> acroff,
                ELSE -> acron
             ENDCOND,
      ...
      ELSE -> null_action
    ENDCOND,
    ...
    clicked := true
   ]
```

Different actions and therefore modes are specified depending on whether the pumps or the control rods are in automatic mode. Moving the cursor has different significance depending on whether the mouse is clicked or not. When the mouse is clicked outside the sensitive areas or when the mouse button is not depressed then the cursor coordinates only are changed. If the mouse is clicked within the space then the appropriate mode is taken. If the mode is related to a slider then the cursor on the slider is moved.

```
move(st: npp_int): npp_int =
    LET new_cursor = mouse(st) IN
    st WITH [
      cursor := new_cursor,
      new_wp1speed :=
```

```
      COND
       (slidermode = wp1s) AND
       (new_cursor`x > wp1_slider(st)`lx) AND
       (new_cursor`x <= wp1_slider(st)`rx)
          -> (wp1_slider(st)`lx - new_cursor`x) * max_flow /
                (wp1_slider(st)`lx - wp1_slider(st)`rx),
       ELSE -> new_wp1speed(st)
       ENDCOND,
     new_wp2speed := ...,
     new_cpspeed := ...,
     new_crposition := ...,
    ]
```

release has the effect of invoking actions in the process layer if the slider mode is non null and also the action is non null. Hence, for example, if `slidermode = wp1s`, that is the flow rate of wp_1 is being changed, then a function is invoked that changes the flow rate of the pump. This function is defined as part of the interface layer but it invokes the relevant function in the process layer.

```
release(st: npp_int): npp_int =
    COND
      slider_mode(st) = wp1s ->
         modify_wp1flow(st),
      slider_mode(st) = wp2s ->
         modify_wp2flow(st),
      slider_mode(st) = cps ->
         modify_cpflow(st),
      slider_mode(st) = crs ->
        modify_crpos(st),
      ELSE -> perform_action(st)
    ENDCOND
```

Aspects of the status of the process are captured in indicators (for example RKS, RKT). The colours of the indicators are linked to states of the underlying reactor (modelled in the second layer), for example if the value of a process attribute is outside specified bounds then the indicator shows the colour red. The model also specifies that the user can perform open/close actions on valves by highlighting the available option in the display.

5.4 Proving properties of the interface layer

Two examples will be used to illustrate how the template properties are instantiated in the interface layer. The first example is concerned with the *visibility* of different aspects of the underlying process, while the second is concerned with *consistency* when releasing the mouse button. The concern in the first example is with the CP pump as it transports water through the cooling pipes in the condenser. A display, specified by the attribute that is part of the interaction mode, *status_cp* simply indicates whether the pump flow is normal (green) or out of bounds (red). The property

to prove is that the display shows these colours correctly. This can be proved by instantiating the visibility property template (Equation 6 in Section 4.2).

```
cp_status_visible: THEOREM
   FORALL (pre, post: npp_int):
     init_state(pre) => cp_visible(pre) AND
     transit(pre, post) AND cp_visible(pre) =>
             cp_visible(post)
```

This theorem contains an induction based on the accessible states. The initial state is specified by the predicate `init_state`. The visibility property to be proved is:

```
cp_visible(st: npp_int):
  (pred_cp_filter(st) <=> pred_p1_cp_filter(st)) AND
  (NOT pred_cp_filter(st) <=> pred_p2_cp_filter)
```

The filter predicates are specified as follows.

```
pred_cp_filter(st: npp_int): boolean =
       process(st)`cp`speed > max_flow
pred_p1_cp_filter(st: npp_int): boolean =
       status_cp(st) = red
pred_ps_cp_filter(st: npp_int): boolean =
       status_cp(st) = green
```

Similar properties can be proved of a range of display features relating to, for example: the status of the process attributes; whether the pumps and control rods are in automatic mode; the values of pump flows and the position of control rods; whether it is possible to switch pumps on or off.

To illustrate the *consistency* property (Equation 4 in Section 4.2) the *release* action is considered in relation to the sliders. The theorem instantiates the template properties indicated in the formulation of the property. The guard and consistency predicates are specified over all the slider modes. We therefore slightly modify the formulation. The aim is to prove the property:

```
consistency_sliders(st: npp_int): boolean =
  con_guard(st) IMPLIES con_release(st)
```

There are four modes to be considered in *MS*, namely `wp1s`, `wp2s`, `cps` and `crs`. The guard checks that the mouse cursor is in the relevant region of the display depending on mode, that release is permitted and that the particular function is not currently automated.

```
con_guard(st: npp_int): boolean =
x_in_area(cursor(st)`x, slidermode(st), st) AND
per_release(st) AND NOT auto(slidermode(st), st)
```

The predicate `x_in_area` checks that the cursor is in a relevant position in relation to the slider.

```
x_in_area(x: x_type, sl: slimo_type, st: npp_int): boolean =
((sl=wp1s) AND
   (x>=wp1_slider(st)`lx) AND (x<=wp1_slider(st)`rx)) OR
((sl=wp2s) AND
   (x>=wp2_slider(st)`lx) AND (x<=wp2_slider(st)`rx)) OR
((sl=wp1s) AND
   (x>=cp_slider(st)`lx) AND (x<=cp_slider(st)`rx)) OR
((sl=wp2s) AND
   (x>=rods_slider(st)`lx) AND (x<=rods_slider(st)`rx))
```

The consistency relation is distributed across these modes as follows:

```
con_release(sl: slimo_type, st: npp_int): boolean =
release(st) =
st WITH
[ pump :=
  COND
    sl = wp1s -> pump(st)`wp1_flow(
                  (cursor(st)`x - wp1_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = wp2s -> pump(st)`wp2_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = cps -> pump(st)`cp_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = crs -> pump(st)`control_rods(
                  (cursor(st)`x - crs_slider(st)`lx)*
                   (control_range/sliderrange)),
    ELSE -> pump(st)
   ENDCOND,
   slider_mode := nulslimo,
   action := nullaction ]
```

The relation *consistent* is equality distributed over the modes, the `filter_pre` indicates what the new state of the process should be, that is for each mode a function in the underling process should be invoked that updates the relevant state attribute.

```
filter_pre(st) =
st WITH
[ pump :=
  COND
    sl = wp1s -> pump(st)`wp1_flow(
                  (cursor(st)`x - wp1_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = wp2s -> pump(st)`wp2_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = cps -> pump(st)`cp_flow(
                  (cursor(st)`x - wp2_slider(st)`lx)*
                   (flow_range/sliderrange)),
    sl = crs -> pump(st)`control_rods(
                  (cursor(st)`x - crs_slider(st)`lx)*
                   (control_range/sliderrange)),
```

```
     ELSE -> pump(st)
    ENDCOND,
    slider_mode := nulslimo,
    action := nullaction ]
```

and `filter_post(st) = release(st)`. *Consistency* relates the change in state `filter_post(st)` to the state before the action in which the mode determined action takes place in the process layer. It determines that *release* always invokes the mode relevant process action (changing pump flow or control rod position). The instantiated consistency theorem is an induction on the actions of the interaction model.

```
consistency_sliders_thm: THEOREM
  FORALL (pre, post: npp_int):
    (init_state(pre) IMPLIES consistency_sliders(pre))
 AND
    (consistency_sliders(pre) AND transit(pre, post)) =>
      consistency_sliders(post)
```

Attempting to prove this theorem identifies an issue with the simulator display. The four sliders occupy the same x-space. The sliders are implemented so that the slider will continue to be dragged across even when the y-coordinate is not in the slider area relating to the mode. It would be imagined that this characteristic would not be a feature of the real control-room display.

5.5 The activity layer

The purpose of the activity layer is to specify assumptions about how the attributes specified in the interface layer, as well as other specified attributes that may be external, are used to carry out the intended activities of the system. It is clearly necessary to know what the activities are that will be performed by the controllers. Typically this information would be gathered by observing existing processes, or by interviewing controllers, or by developing scenarios with domain experts that relate to anticipated constraints in terms of new design concepts. The approach is described in more detail in [Campos et al., 2014].

Given the limited information provided by the use case, it is difficult to develop and assess plausible assumptions. However we do have operating procedures associated with starting up and closing down the reactor. This will be the information that provides the basis for sketches of the activity layer given here.

The aim of start up is to bring output power to 700 MW (100% of possible output power) and to hold the water level in the reactor tank stable at 2100 mm. The operating procedure is as follows:

1. Open SV2
2. Set CP to 1600 u/min
3. Open WV1

4. Set WP1 to 200 u/min
5. Stabilise water level in the reactor tank at 2100 mm by pulling out the control rods
6. Open SV1
7. ...

The process of developing the activity model involves, for each step in the operating procedure, considering the information resources that are conjectured to enable the user to take the appropriate action in the interface. The action is resourced if information relevant to the use of the action is clear in the interface. The activity model also considers how the user is notified what the next action is to be performed. In the case of this fragment it will be assumed that the written operating procedure will be used to decide the sequence. However in other circumstances it should be considered whether the user will be allowed by the control system to change the order of the operating procedure and what the effect of such a change would be. The action `OpenSV2` is expressed in the model as

```
open_valve(st WITH [action := opensv2])
```

The open valve action is generic to the valves supported by the interface and is made specific by the attribute `action`. It may be assumed that this action would be triggered if

- the openSV2 button area is enabled, that is it is highlighted: `highlightosv2 = true`. This should only be true if `sv2_open = false`, a property checked of the interface model.
- the cursor is within the osv2 area:

```
(cursor(st)`x <= osv2area(st)`lx) AND
(cursor(st)`x >= osv2area(st)`rx) AND
(cursor(st)`y <= osv2area(st)`dy) AND
(cursor(st)`y >= osv2area(st)`uy)
```

The resource layer specifies all the constraints based on assumptions about what triggers the actions supported by the interface as well as activities that are to be performed. When these assumptions have been specified they can be used as additional constraints when proving theorems based on the templates. The resource layer makes it possible to prove whether the properties are true in circumstances that afford some measure of plausibility in relation to what users do.

Additional actions may be specified that characterise the activities performed by users. For example, consider the user activity *recover*, in contrast to the autonomous action that causes recovery. This activity would involve several actions before the goal of the activity is completed. Information resources would help the operator to begin the activity. This means that the activity also has information resource constraints. For example, it would specify that "increasing pressure", using the relevant action in the interface layer, would occur only if other actions had already been completed and the displayed tank, valve and pump parameters specified in the second layer were displayed (in the interface layer) indicating particular values.

Further activities include for example “monitor recovery”. This would be expressed as an action that describes the constraints on the operator when monitoring an autonomous recovery. The specification of the action would include the information resources that would be required in the monitoring process at different stages of the recovery and would specify the conditions in which any user actions would take place.

The value of expressing constraints in this way is that theorems that are instantiations of property patterns of Section 4.2 can be proved subject to the resource constraints. Properties can be considered that only relate to plausible interactions. This would be relevant if it was considered inappropriate to analyse properties across sequences of actions that would not plausibly occur. The implications of such an analysis are that an understanding of whether an action is plausible becomes more relevant and this requires an understanding of the human factors of a situation. This topic is considered in more detail in [Harrison et al., 2016]. It can also be proved that, for the steps of the operating procedures, the constraints are satisfied.

6 Related Work

Models, of the type outlined, have been developed for other interactive systems using both a model checking approach and a theorem proving approach [Masci et al., 2012, Harrison et al., 2015a, 2014, Campos et al., 2016]. The advantage of model checking is that it is possible to explore, more readily, reachability properties as well as potential non-determinisms. The disadvantage is that the size of model is seriously limited. It is possible to explore the essential details of the control of the nuclear power plant using a model checking approach but as soon as a realistic process model is used this becomes impossible. Making the model abstract enough, to make analysis feasible, would restrict what could be asked of the model. It would be more difficult to prove relevant properties.

Theorem proving allows analysis of larger models but properties may be more difficult to formulate and prove. In particular, while model checking allows simple formulations of reachability properties, these are difficult to specify using a theorem proving approach. There is a tradeoff to be made between the effort needed to develop a model amenable to verification and the effort needed to carry out the proofs. Typically a theorem proving based approach will gain advantage in the former, because of more expressive languages, and model checking in the latter, because of more automated analysis. In all cases, how to identify and express the properties of interest is also an issue.

Design patterns and property templates have been extensively studied in engineering practices. Most of the effort, however, has been devoted to creating patterns and templates for the control part of a system, rather than for the human-machine interface. [Vlissides et al., 1995] established a comprehensive set of standard design patterns for software components of a system. An example pattern is the *abstract factory*, which facilitates the creation of families of software objects (e.g., windows

of a user interface). Another example is the *adapter* pattern, which converts the interface of software components to enable integration of otherwise incompatible software components. These patterns are a de-facto standard in the software engineering community and they are widely adopted in engineering practices to solve common problems related to the realisation of software components. [Konrad and Cheng, 2002] discuss design patterns for elements of embedded systems. An example pattern is the *actuator-sensor* pattern, providing a standard interface for sensors and actuators connected to the control unit of an embedded system. Similarly, [Sorouri et al., 2012] present a design pattern for representing the control logic of an embedded system. [Lavagno et al., 1999] introduced *Models of computation (MoC)* as design patterns for representing interactions between distributed system components. Recently, [Steiner and Rushby, 2011] have demonstrated how these MoC can be used in model-based development of systems, to represent in a uniform way different time synchronisation services executed within the system. These and similar activities are concerned with design patterns for the control part of a system, as opposed to the human-machine interface – e.g., problems like how to correctly design the behaviour of data entry software in human-machine interfaces are out of scope.

Various researchers have introduced design patterns for the analysis of complex systems. For example, in [Li et al., 2014], verification patterns are introduced that can be used for the analysis of safety interlock mechanisms in interoperable medical devices. Although they use the patterns to analyse use-related properties such as "*When the laser scalpel emits laser, the patient's trachea oxygen level must not exceed a threshold* Θ_{O_2}", the aim of their patterns is to facilitate the introduction of a model checker in the actual implementation of the safety interlock, rather than defining property templates for the analysis of use-related aspects of the safety interlock. Other similar work, e.g., [Tan et al., 2015, King et al., 2009, Larson et al., 2012], also introduce design patterns for the verification of safety interlocks, but the focus of the patterns is again on translating verified design models into a concrete implementation – in [Tan et al., 2015], for example, the design patterns are developed for the automatic translation of hybrid automata models of a safety interlock into a concrete implementation.

Proving requirements similar to the properties produced from the templates of this paper has been the focus of previous work. For example, a mature set of tools have been developed using SCR [Heitmeyer et al., 1998]. Their approach uses a tabular notation to describe requirements which makes the technique relatively acceptable to developers. Combining simulation with model checking has also been a focus in, for example, [Gelman et al., 2013]. Recent work concerned with simulations of PVS specifications provides valuable support to this complementarity [Masci et al., 2013]. Had the specification been developed as part of a design process then a tool such as Event B [Abrial, 2010] might have been used. In such an approach an initial model is first developed that specifies the device characteristics and incorporates the safety requirements. This model is gradually refined using details about how specific functionalities are implemented.

In our previous work, we have introduced modelling patterns for decomposing interactive (human-machine) system models into a set of layers to facilitate models

reuse [Harrison et al., 2015a]. [Bowen and Reeves, 2015], who are concerned with design patterns for user interfaces, complements our work on modelling patterns. They have introduced four modelling patterns: the *callback* pattern, representing the behaviour of confirmation dialogues used to confirm user operations; the *binary choice* pattern, representing the behaviour of input dialogues used to acquire data from the user; the *iterator* pattern, representing the behaviour of parametric user interface widgets that share the same behaviour but have a different value parameter, such as the numeric entry keys 0–9; and the *update* pattern, for representing the behaviour of a numeric display.

7 Discussion and Conclusions

Two approaches to specification and proof are possible with the considered examples: model checking and theorem proving. Model checking is the more intuitive of the two approaches. Languages such as Modal Action Logic with interactors (MAL) [Campos and Harrison, 2008] express state transition behaviour in a way that is more acceptable to non-experts. The problem with model checking is that state explosion can compromise the tractability of the model so that properties to be proved are not feasible. Model checking, hence, is more convenient for analysing high level behaviour, for example when checking the modal behaviour of the user interface. Theorem proving, while being more complex to apply, provides more expressive power. This makes it more suitable when verifying properties requiring a high level of detail, such as those related to a number entry system, because the domain of numbers is relatively large.

To employ the strengths of the two approaches simple rules can be used to translate from the MAL model to the PVS model that is used for theorem proving. Actions are modelled as state transformations, and permissions that are used in MAL to specify when an action is permitted are described as predicates. The details of the specification carefully reflects its MAL equivalent. This enables us to move between the notations and verification tools, choosing the more appropriate tool for the verification goals at hand.

One aspect that has not been discussed in this chapter is the analysis and interpretation of verification results. In particular, the possibility of animation of the formal models to create prototypes of the modelled interfaces, and the possibilities these prototypes raise in terms of discussing the results of verification with stakeholders. Such prototypes can be used either to *replay* traces produced by a model checker or interactively to both discuss the findings of the verification or help identify relevant features of the system that should be addressed by formal analysis. This approach is described in [Masci et al., 2014].

Acknowledgements José Creissac Campos and Michael Harrison were funded by project ref. NORTE-07-0124-FEDER-000062, co-financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF),

through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese foundation for science and technology (FCT). Paul Curzon, Michael Harrison and Paolo Masci were funded by the CHI+MED project: Multidisciplinary Computer Human Interaction Research for the design and safe use of interactive medical devices project, UK EPSRC Grant Number EP/G059063/1.

References

AAMI. Medical devices - application of usability engineering to medical devices. Technical Report ANSI AMI IEC 62366:2007, Association for the Advancement of Medical Instrumentation, 4301 N Fairfax Drive, Suite 301, Arlington VA 22203-1633, 2010.

J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

M. L. Bolton, E. J. Bass, and R. I. Siminiceanu. Generating phenotypical erroneous human behavior to evaluate human-automation interaction using model checking. *International Journal of Human-Computer Studies*, 70:888–906, 2012.

J. Bowen and S. Reeves. Design patterns for models of interactive systems. In *Software Engineering Conference (ASWEC), 2015 24th Australasian*, pages 223–232. IEEE, 2015.

J. C. Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In N. Graham and P. Palanque, editors, *Interactive systems: Design, Specification and Verification, DSVIS '08*, number 5136 in Springer Lecture Notes in Computer Science, pages 72–85. Springer-Verlag, 2008.

J. C. Campos and M. D. Harrison. Interaction engineering using the IVY tool. In G. Calvary, T.C.N. Graham, and P. Gray, editors, *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 35–44. ACM Press, 2009.

J. C. Campos, G. Doherty, and M. D. Harrison. Analysing interactive devices based on information resource constraints. *International Journal of Human-Computer Studies*, 72:284–297, 2014.

J. C. Campos, M. Sousa, M. C. B. Alves, and M. D. Harrison. Formal verification of a space system's user interface with the IVY workbench. *IEEE Transactions of Human Machine Systems*, 46(2):303–316, 2016.

D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.

G.E. Gelman, K.M. Feigh, and J. Rushby. Example of a complementary use of model checking and agent-based simulation. In *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, pages 900–905, Oct 2013. doi: 10.1109/SMC.2013.158.

J. Gow, H. Thimbleby, and P. Cairns. Automatic critiques of interface modes. In S. Gilroy and M.D. Harrison, editors, *Proceedings 12th International Workshop on the Design, Specification and Verification of Interactive Systems*, number 3941

in Springer Lecture Notes in Computer Science, pages 201–212. Springer-Verlag, 2006.

M. D. Harrison, P. Masci, J. C. Campos, and P. Curzon. Demonstrating that medical devices satisfy user related safety requirements. In *Proceedings of Fourth Symposium on Foundations of Health Information Engineering and Systems (FHIES) & Sixth Software Engineering in Healthcare (SEHC) Workshop*. Springer-Verlag, 2014. in press.

M.D. Harrison, J.C. Campos, and P. Masci. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering*, 11(2):95–111, June 2015a. ISSN 1614-5046.

M.D. Harrison, J.C. Campos, and P. Masci. Patterns and templates for automated verification of user interface software design in pvs. Technical Report TR-1485, School of Computing Science, Newcastle University, 2015b.

M.D. Harrison, J.C. Campos, R. Rukšėnas, and P. Curzon. Modelling information resources and their salience in medical device design. In *Proceedings ACM Symposium Engineering Interactive Systems (EICS 2016)*. ACM Press, 2016. in press.

C. Heitmeyer, J. Kirby, and B. Labaw. Applying the SRC requirements method to a weapons control panel: an experience report. In *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP '98)*, pages 92–102, 1998.

A. L. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. Jetley, P. Raoul, P.L. Jones, and S. Weininger. An open test bed for medical device integration and coordination. In *ICSE Companion*, pages 141–151, 2009.

S. Konrad and B. H. C. Cheng. Requirements patterns for embedded systems. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 127–136. IEEE, 2002.

B. Larson, J. Hatcliff, S. Procter, and P. Chalin. Requirements specification for apps in medical application platforms. In *Proceedings of the 4th International Workshop on Software Engineering in Health Care*, pages 26–32. IEEE Press, 2012.

L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich. Models of computation for embedded system design. In *System-Level Synthesis*, pages 45–102. Springer, 1999.

T. Li, F. Tan, Q. Wang, L. Bu, J. Cao, and X. Liu. From offline toward real time: A hybrid systems model checking and CPS codesign approach for medical device plug-and-play collaborations. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):642–652, 2014.

P. Masci, H. Huang, P. Curzon, and M. D. Harrison. Using PVS to investigate incidents through the lens of distributed cognition. In A. E. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 273–278. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28890-6. doi: 10.1007/978-3-642-28891-3_27. URL `http://dx.doi.org/10.1007/978-3-642-28891-3_27`.

P. Masci, A. Ayoub, P. Curzon, I. Lee, O. Sokolsky, and H. Thimbleby. Model-based development of the generic PCA infusion pump user interface prototype in PVS. In F. Bitsch, J. Guiochet, and M. Kaâniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Springer Lecture Notes in Computer Science*, pages 228–240. Springer-Verlag, 2013. ISBN 978-3-642-40792-5.

P. Masci, Y. Zhang, P. Jones, P. Curzon, and H. W. Thimbleby. Formal verification of medical device user interfaces using PVS. In *ETAPS/FASE2014, 17th International Conference on Fundamental Approaches to Software Engineering*, Berlin, Heidelberg, 2014. Springer-Verlag.

P. Masci, P. Oladimeji, P. Curzon, and H. Thimbleby. PVSio-web 2.0: Joining PVS to Human-Computer Interaction. In *27th International Conference on Computer Aided Verification (CAV2015)*. Springer, 2015. Tool and application examples available at http://www.pvsioweb.org.

J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In J. Chew and J. Whiteside, editors, *ACM CHI Proceedings CHI '90: Empowering People*, pages 249–256, 1990.

P. G. Polson, C. Lewis, J. Rieman, and C. Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741–773, 1992.

N. Shankar, S. Owre, J. M. Rushby, and D. Stringer-Calvert. PVS System Guide, PVS Language Reference, PVS Prover Guide, PVS Prelude Library, Abstract Datatypes in PVS, and Theory Interpretations in PVS. Computer Science Laboratory, SRI International, Menlo Park, CA, 1999. Available at `http://pvs.csl.sri.com/documentation.shtml`.

M. Sorouri, S. Patil, and V. Vyatkin. Distributed control patterns for intelligent mechatronic systems. In *Industrial Informatics (INDIN), 2012 10th IEEE Intl. Conference on*, pages 259–264. IEEE, 2012.

W. Steiner and J. Rushby. TTA and PALS: Formally verified design patterns for distributed cyber-physical systems. In *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*. IEEE, 2011.

F. Tan, Y. Wang, Q. Wang, L. Bu, and N. Suri. A lease based hybrid design pattern for proper-temporal-embedding of wireless CPS interlocking. *Parallel and Distributed Systems, IEEE Transactions on*, 26(10):2630–2642, 2015.

J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.