10

Systematic methods for user interface design

O. Lauridsen
DDC-I A/S and Technical University of Denmark
DDC-I A/S, Gl. Lundtoftevej 1B, DK-2800 Lyngby. ol@ddci.dk

Abstract

This paper discusses the use of systematic software development methods in designing interactive systems, in particular in designing the user interfaces of interactive systems. The paper discusses justification of transformation of functional specifications into user interface specification, and the use of a verification tool for verifying system properties. We will demonstrate how some properties can be formulated, so they can be used together with a system specification in a justification.

Keywords

User interface design, formal specification, systematic software development.

1 INTRODUCTION

There have been a couple of approaches to verification of relations between specifications of the functional part of an application (the functional core) and the user interface part. Some of these build on the PIE model or the red-PIE model (Dix 1991) extended to a template model (Abowd 1991, Roast 1994). A slightly different approach has been taken by Duke and Harrison (1993), who consider the user interface (presentation) an abstraction of the functional core (internal state). With this approach it is possible to apply all of the proof obligations and coupling invariants between abstraction and implementation from the concept of data refinement (Jones 1986), mainly used for verification of functional correctness. Most of these approaches have considered the issue on a general and abstract level. In this paper, we will apply the data refinement concept to specifications of interactive systems in a more operational way, in the sense that it should be possible to justify system properties from the system specification alone. The user interface specifications are more concrete and detailed than in many other formal specification approaches by defining a set of user interface primitives, and it is therefore easier to generate an imple-

L. J. Bass et al. (eds.), *Engineering for Human-Computer Interaction*

mentation from the design specification. For specification of user interfaces, we will use a set of primitives (within an architectural model), which can be considered a refinement of Abowd's agent model (Abowd 1991). This allows us to use formal proof for justification of system properties. Duke and Harrison (1993) use the data refinement concept, considering the presentation to be the abstract specification and the functional specification to be the concrete implementation. We will take an opposite approach, letting the user interface be the implementation.

In order not to invent another specification language, we re-use an existing language to describe user interface primitives. The primitives are described as class modules within the language, implying that user interfaces may be specified in this language extended with the user interface primitive classes. The advantages are that the language is known; supporting tools for the language already exist, including a justification tool; and the same language may be used to specify both functionality and user interface. The specification language used, is the RAISE Specification Language* (RAISE 1992).

To illustrate the specification language, the use of the primitives, and the justification of properties described later, we will describe a database-like example: An object code information library for a compiler system. In this simplified example, the *library* is a mapping from a *name* to a *description*, consisting of a set of *attributes* and a set of *containers*. Four operations are defined on the library, a *list* operation to look up information in the library, a *remove* operation to remove entries in the library and two operations to disable and enable the remove operation, *lock* and *unlock*, respectively. This constitutes the specification of the functional core for which we would like to create a user interface (basic parts of the RAISE notation are explained in the last section).

```
object
  LibFC :
    class
      type
        Container == debug| obj | symtable | ... ,
        Containers = Container-set,
        Attr == locked | unlocked | ... ,
        Attributes = Attr-set,
        Decl == spec | rename | instance | subprog | body | subunit | generic,
        Name == mk_name(Text, Decl),
        Description == mk_desc(Attributes, Containers)

      variable library : Name ~m~> Description

      value
        /* lists library info. on a given name */
        list : Name → read library Description
        list(n) as d post d = library(n) pre n ∈ dom library,

        /* removes an entry in the library */
        remove : Name → read library write library Unit
```

*The language is developed on the basis of the VDM language, by a consortium of European companies and universities sponsored by the EC. The language is supported by a large set of tools, eg a context sensitive editor, a LaTeX formatter, proof assistance, Ada and C++ translators.

Figure 1 Layout example of the compiler library.

```
    remove(n) post library = library \ {n} pre n ∈ dom library,

    /* locks an entry for removal */
    lock : Name → read library write library Unit
    lock(n) ≡ ...,

    /* enables removal of an entry */
    unlock : Name → read library write library Unit
    unlock(n) ≡ ...
end
```

2 USER INTERFACE PRIMITIVES

The framework, in which the user interface primitives are used, is an architectural model of five layers, where an application consists of a functional core, functional core adapter, dialog control, logical interaction and physical interaction (the Arch/Slinky model (UIMS Workshop 1992) combined with the PAC model (Coutaz 1987) and the MVC model (Krasner and Pope 1988)), Figure 2. The dialog control and the logical interaction are specified by an agent model derived from Abowd's agent model (Abowd 1991). In order to be able to transform functional specifications into user interface specifications, some restrictions must be imposed on the functional specification. These restrictions are expected to be

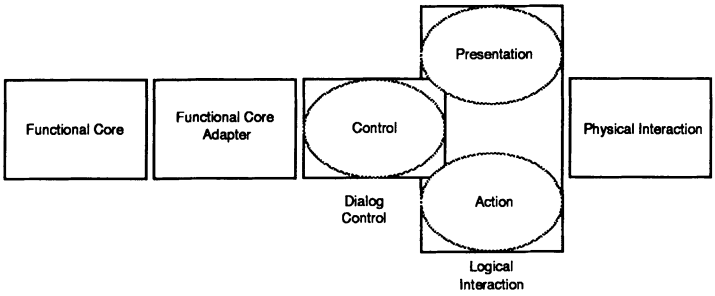

Figure 2 System Architecture.

'implemented' in the functional core adapter. One important restriction is that the functional core adapter must describe user tasks in order to map the user interface adequately to users' domain model, that is, only that which should be visible in the user interface, should be visible in the functional core adapter. Thus, the specification of the functional core adapter will be the basis of our discussion about the functional specification. We will refer to specifications of the functional core adapter as functional specifications. In the context of data refinement, we will consider the functional core adapter layer an abstraction of the functional core and of the user interface, so that the functional core is one implementation of the functional core adapter and the user interface is another implementation of the functional core adapter.

A simple abstract agent model consists of: A set of states S, one of which is identified as the initial state $s_0 \in S$; a set of input events I generated by the user or some agent; a set of output events O; the output for any state is determined by a *behavior* function $b : S \times I \rightarrow S \times O$. The model is:

```
M :
class
  type S, I, O
  variable state : S := s0
  value b : S × I → S × O
end
```

We will be using the agent model to model the dialog control and the logical interaction, and we will define a set of primitives for modeling these layers. The dialog control consists of *control* agents and the logical interaction consists of *presentation* agents and *action* agents. Agents are instances of classes: Control classes, presentation classes and action classes. All communication passes through a control component with no direct communication between action and presentation components.

Separating the user input from the system output provides flexibility to combine any kind of presentation with any kind of action, coordinated by a control agent. The communication between agents is as follows: An action agent sends a message to its control

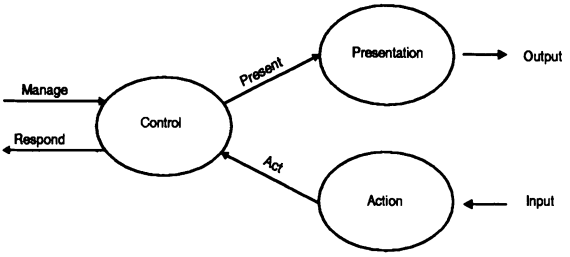

Figure 3 Agent Communication.

agent when some kind of user input is received (via the *act* channel). The control agent propagates the message to the connected presentation agent (via the *present* channel) and possibly to another control agent (via the *respond* channel). Eventually, the presentation agent modifies the physical appearance of the user interface and the functional core processes the input. A control agent receives messages from other control agents via the *manage* channel (Figure 3).

2.1 Dialog Control

A dialog control specification, *DC*, consists of a set of control agents. A control agent, *CtrlAgent*, is an instance of a control class. There are six control classes, each of them described below. A class is instantiated with parameters establishing the connection to the functional core adapter. Control agents do not describe physical presentation or interaction, they describe the connection between the functional core adapter and the user interface. These, in terms of abstract user interface primitives, describe the dynamic behavior of the user interface.

Agent classes are specified in RAISE and dialog control specifications are specified in RAISE using the agent classes. To give an overview of the content of agents and dialog control specifications, and for justification purposes later, we will here give a description of agents in an abstract syntax (also given in RAISE syntax):

```
type
  DC = CtrlAgent-set,

  /* control agent */
  CtrlAgent = Command | Entity | Exclusive | Inclusive | Group | View,

  Command == mk_cmd(id : Id, func : Func),
  Entity == mk_entity(id : Id, datatype : DataType),
  Exclusive == mk_excl(id : Id, datatype : DataType, al : CtrlAgent*),
  Inclusive == mk_incl(id : Id, datatype : DataType, al : CtrlAgent*),
  Group == mk_grp(id : Id, al : CtrlAgent*),
  View == mk_view(id : Id, al : CtrlAgent*)
```

```
value
  /* extracts the set of agents contained in an agent containing other agents */
  children : CtrlAgent → CtrlAgent-set,

  /* creates a set of all agents in the hierarchies of agents in a DC */
  flat_set : DC → CtrlAgent-set
```

The types *Id* and *Func* are defined in the abstract syntax for the functional specification below. The functions *children* and *flat_set* are used later to describe system properties. *Children* extracts all control agents contained in the hierarchy of control agents in a group or a view agent. *Flat_set* does the same as *children*, but for a full dialog control specification.

Six *Control classes* are defined, covering the different types of structuring and dynamic behavior in a user interface, without considering the actual physical interaction and presentation (the semantics of the user interface). The six control classes can be divided into three categories reflecting the semantic of the classes: simple classes, eg *command* and *entity*, classes containing agents to choose from/between, eg *exclusive choice* and *inclusive choice* classes, grouping classes, arranging agents of all classes as *group* and *view*. Command agents initiate operations in the functional core. Entities hold values of parameters, variables or results. Choice agents can contain entity or command agents. Choice agents can be a list of commands to choose from or a set of entities representing a range of values for a parameter or a set of inclusive options etc. Group agents group related agents of any class except views. Views contain groups of other agents. Views are considered independent sessions, as a separate application or a separate task within an application. For example, parameters for one function in the functional core adapter would typically be grouped together in one view and presented in the logical interaction as a separate window (a dialogbox). The complete set of control classes needed is listed below:

Simple classes:

- *Command*, an agent which initiates an operation in the functional core and implies a visible context shift, for example by opening or closing a dialogbox.
- *Entity*, any kind of value input or output.

Choice classes:

- *Exclusive choice*, selection of one value from a limited discrete range of values.
- *Inclusive choice*, selection of one or more values from a limited discrete range of values.

Grouping classes:

- *Group*, group of related agents could be presented as a frame in a dialog-box.
- *View*, a separate view, highest level of organizing agents could be presented as a dialogbox or a separate window.

Translation to Dialog Control Specification

A functional specification is systematically transformed into a dialog control specification in the following way: Each function generates a command control agent, because function invocation is an event like issuing a command, and for each function either two view agents are generated. One view for the input parameters and one for the output parameters. Parameters with simple interaction are translated into entity agents, while parameters as enumeration types are translated into exclusive or inclusive choice agents, etc.

Navigation, moving around between views, window navigation, and update of data objects in different views must be specified explicitly in the dialog control specification as the dynamic behavior of the user interface defined in the behavior functions of group and view agents. This is because it is not a part of the semantic specification of the functional core, nor is it a part of the logical interaction specification.

An abstract syntax for the functional specification to transform from, looks as follows: A functional specification, *FCA*, consists of a list of objects each containing an identifier *id*, a list of objects *objs*, a list of function declarations *funcs*, a list of type declarations *types* and a list of variable declarations *vars*. A function declaration *Func* consists of an identification, a list of input parameters *input*, and a list of output parameters *output*. Each parameter *Arg* has an identification and a type *datatype*, just as a variable *Variable* has an identification and a type.

```
object
  FuncSpec :
    class
      type
        FCA = Obj*,
        /* Object class declaration */
        Obj == mk_obj(id : Id, objs : Obj*, funcs : Func*, types : DataType*, vars : Variable*),

        /* Function declaration */
        Func == mk_func(id : Id, input : Arg*, output : Arg*),
        Arg == mk_arg(id : Id, type_def : DataType),
        /* Data type declaration */
        DataType = Null | Boolean | Integer | Float | Enumeration | String | Record | Set | List | ...,
        Null == mk_null,
        Boolean == mk_boolean,
        Integer == mk_integer_range(low : Int, high : Int) | mk_integer,
        Float == mk_float_range(low : Real, high : Real) | mk_float,
        ⋮

        Id,  /* left unspecified */

        /* Variable declaration */
        Variable == mk_var(id : Id, datatype : DataType)
    end
```

In the case of the library example, the specification is so simple (simplified) there is no need to distinguish between a functional core and a functional core adapter. A dialog control specification for the library example explained previously, could look as below

(described as a concrete dialog control specification as opposed to the abstract syntax just described). A command agent is specified for each function in the functional core (adapter), and given the name of the function. From the pre-conditions of the functions we know the legal set of input values is a finite set, hence the input parameter can be described by an exclusive choice agent *name*, containing all legal values. The set of legal values is dynamic as entries can be removed by the remove function (and added by a compile function not modeled here). The set of legal values is defined by the function *member*, which is passed as a parameter in the instantiation of the exclusive choice class. A variable *selected*, holding the currently selected name, is the one input parameter for all of the functions. The axioms describe invariants of the coupling between the functional core (adapter) and the dialog control. *Completeness* says that all names present in the dialog control should exist in the library and all names in the library should be present in the dialog control. *Selection* states that either there is no selected name or a selected name exists in the library.

```
LibDC :
  /* The dialog control consists of a view agent extended with the following specification */
  extend VIEW with
    class
      object
        M :
          class
            value
              member : LibFC.Name → read any Bool
              member(n) ≡ n ∈ dom LibFC.library
          end,

        /* Command agents */
        list : COMMAND(LibFC.list),
        remove : COMMAND(LibFC.remove),
        lock : COMMAND(LibFC.lock),
        unlock : COMMAND(LibFC.unlock),

        /* Entity agent for type LibFC.Description */
        description : ENTITY(LibFC{Description for Data}, DescEvents),

        /* Exclusive choice agent for selection of a name */
        name : EXCLUSIVE(LibFC{Name for Data}, NameEvents, M{member for item})

      type
        Selected == none | mk_name(LibFC.Name)

      variable selected : Selected := none

      value
        /* Behavior function receiving and sending messages from/to the agents above and LibFC */
        behavior : Unit → in any out any read any write any Unit
        behavior() ≡ ... /* omitted for space reasons */

      axiom
        [completeness]
```

```
    { n | n : name.Index } = dom LibFC.library

    [selection]
    selected ∈ {s | s :Selected • ∃ n : LibFC.Name •
                        n ∈ dom LibFC.library ∧ s = mk_name(n)} ∪ none

end
```

2.2 Logical Interaction

The logical interaction *LI* consists of presentation and action agents. A presentation agent *PresentationAgent* is an instantiation of a presentation class and an action agent *ActionAgent* is an instantiation of an action class. Any action and presentation class is instantiated with one control agent which controls the communication between action and presentation, and action and other control agents or the functional core adapter. Several presentation and action agents can be instantiated with the same control agent giving multiple views and/or multiple interaction forms for the same control agent.

```
type
  LI = (PresentationAgent × ActionAgent)-set,

  /* presentation agent */
  PresentationAgent = Push | Toggle | Slider | Field | Node | Edge | Menu | List |
     Panel | Table | Graph | Dialog | Window | ...,

  Push == mk_push(id : Id, ctrl_agent : CtrlAgent),
  ⋮

  /* action agent */
  ActionAgent = Select | Place | Move | FctKey | AccKey | TypeIn | None,

  Select == mk_select(id : Id, ctrl_agent : CtrlAgent),
  ⋮

value
  /* functions extracting the control agent from presentation and action agents */
  ctrl_agent : PresentationAgent → DialogControl.CtrlAgent,
  ctrl_agent : ActionAgent → DialogControl.CtrlAgent,

  /* extracts the set of presentation agents from LI */
  presentations : LI → PresentationAgent-set,

  /* extracts the set of action agents from LI */
  actions : LI → ActionAgent-set
```

The functions *ctrl_agent, presentations* and *actions* are used in the specification of properties.

A presentation agent may consist of any kind of audiovisual presentation. The set of presentation classes described here corresponds with the logical elements in a WIMP (Window-Icon-Menu-Pointer) user interface. *Command* control agents can, for example, be connected with presentation agents of class *push* or *field*. Whereas *view* agents can be connected with agents of class *dialog* or *window*. The connection is made by instantiating a presentation class with the control agent. A set of presentation classes are listed above in the abstract specification, new presentation classes may be created if necessary.

Action agents define the input activities to be performed by the user in order to execute an action. At this point, the purpose of the user action is not considered; nor is the kind of feedback to be given to a specific action. The definition of user actions is independent of semantics and presentation. Actions are partitioned according to the type of activity rather than device, for example, a select action can model both a mouse click and pressing the enter key on the keyboard or any other kind of selection mechanism. Connection to the control agent is made by instantiating an action class with the control agent. A set of action classes are listed above in the abstract specification, new action classes may be created if necessary.

The action and presentation agents defined here are very similar to the set of *abstract interaction objects* defined in TRIDENT (Vanderdonckt and Bodart 1993) and other systems for automatic generation of user interfaces.

Translation to Logical Interaction Specification

A specification of the dialog control is translated into a logical interaction specification by converting control agents into presentation and action agents. This is done according to guidelines and heuristics.

In the library example, the dialog control specification can be translated into a logical interaction specification as follows: There is a push-button presentation *list*, and a select action agent *list_action* for each command agent. The exclusive agent is presented by a list agent *name*. Items in the list agent are presented by toggle-button agents *item*, and selection of a list item is handled by a select action agent *item_action* for each item. The description agent is translated into a field presentation agent *description* and a none-action agent *description_action*, as there is no way of interacting with the output of the list function. In this example, we do not distinguish between the different elements (attributes and containers) within the description type, they all goes into the same field. An alternative presentation of the description type could be a group or a view containing an entity for each of the elements.

```
object
  LibLI :
    class
      object
        list : PUSH(LibDC.list),
        list_action : SELECT(LibDC.list),

        :

        item : TOGGLE(LibDC.name.LIST_ELEMENT),
        item_action : SELECT(LibDC.name.LIST_ELEMENT),
```

```
    name : LIST(LibDC.name),
    name_action : PLACE(LibDC.name),

    description : FIELD(LibDC.description),
    description_action : NONE(LibDC.description)
end
```

3 JUSTIFICATION OF PROPERTIES

Having specified the dialog control and the logical interface it should be possible to verify certain properties of the specification. There are two types of verification of interest: one is the verification of invariants and axioms describing the actual correspondence between the dialog control and the functional specification, and between the dialog control and the logical interface. Such axioms can be explicitly stated in the specification. The other type of verification is the verification of some general system properties and system invariants, as defined below. Some of these properties can be formalized, and some of these formalizations can be used on concrete specifications for proving that the system does have the properties. The approach taken here are similar to Abowd (1991) and Duke and Harrison (1993), while Palanque, Bastide, Dourte and Sibertin-Blanc (1993) have used Petri nets for justification of some of the same properties, and Systä (1994) has used a special purpose method DisCo, for reactive systems to justifying system properties.

3.1 Data refinement and Proof Obligation

The relationship between functional specification and user interface specification can be considered a data refinement, considering one specification an implementation of another specification, due to greater details or more concrete data types. We can use the proof obligations of data refinement (Jones 1986) to justify the transformation from functional specification to dialog control and logical interface specifications. The transformation raise an implementation proof obligation, verifying that the user interface presents (implements) the functional core (adapter) adequately. Invariants are used to state the relation between the elements in the functional specification and the dialog control specification, and between dialog control and logical interface.

The RAISE justification tool defines an implementation relation (Brock and George 1990). When this relation is used, the validity of the relation is controlled automatically. This relation is too restrictive for our purpose and we will therefore use the implementation relation called data refinement.

The *adequacy* proof obligation of data refinement says that at least one dialog control/logical interaction value must exist, which can be mapped onto any possible functional core/dialog control value. The mapping is described by a retrieve function *retr-*T_{FC}, mapping back from dialog control specification to functional specification, and from logical interaction to dialog control. T_{FC} is any type in the functional specification, and T_{DC} is any type in the dialog control specification. In particular, it has to hold that an initial state

value in a dialog control specification p_0 represents an initial state value in the functional specification a_0, and the same holds between logical interaction and dialog control.

[retrieve function]
retr–T_{FC}: $T_{DC} \rightarrow T_{FC}$

[adequacy]
$\forall$ a : T_{FC} •
  $\exists$ p : T_{DC} • retr–T_{FC}(p) = a

[initial state]
retr–$T_{FC}(p_0) = a_0$

Retrieve functions or coupling invariants are equivalent to the relation between result and display in the red-PIE model and the template model. The retrieve function is a simple decomposition, extracting the element in the dialog control/logical interaction agent describing a variable, a function or a function parameter. This element will be of the same type as in the functional core adapter. Nevertheless, it is quite complicated to prove that the dialog control type is an adequate refinement of the functional core adapter type.

The verification can be done on different abstraction levels. Some properties can be verified from a concrete specification, other properties have to be verified from a higher level, where the concrete specification is expressed in an abstract syntax.

Data refinement also raises *operation modeling* proof obligations, to verify the implementation of functions working on the refined data types. These proof obligations are not as important as the adequacy proof obligation, because operations are not implemented in the user interface, just represented by invocation (command) agents. But the domain rule can be used when considering range check of input in the user interface, to ensure that the range check is appropriate. This issue is not considered here. Instantiating classes with data types from the functional specification implies an implicit range check, or rather type check in the user interface.

3.2 Properties

The general system properties we want to justify, are taken from the sets of properties defined in IFIP WG2.7 (1995) and Dix, Finlay, Abowd and Beale (1993), which present very similar lists of properties, but with slightly different definitions. Some of these properties will be addressed in a formal way in order to justify a system's possession of these properties. Only properties, which can be given an operational definition, are considered here.

Predictability

Predictability is the user's ability to predict future states and response time from current and prior observable states. From the definition of predictability it seems impossible to formulate an operational proof obligation for predictability. Operation visibility, which

is connected to predictability, on the other hand, seems much simpler. *Operation visibility* or functional *completeness* refers to how the availability of operations, which can be performed, are shown to the user. The proof obligation could be to justify that all possible operations are visible, ie visible dialog control or logical interaction agents exist, representing the available operations.

Operation visibility, in dialog control specifications, is: For all functions f in the functional specification, a view agent v and a command agent c exist in the dialog control specification for which it holds that the command agent is contained in the view agent and the command agent is connected to the function f.

```
/* FC to DC operation visibility */
operation_visibility : FC × DC → Bool
operation_visibility(fc, dc) ≡
  (
    ∀ f : Func •
      ∃ v : View, c : Command •
        f ∈ funcs(fc) ∧
        v ∈ dc ∧
        c ∈ children(v) ∧ func(c) = f
  )
```

In the logical interaction specification operation visibility is expressed as follows: for all commands c, in the dialog control specification, a presentation agent p exists in the logical interaction specification, which is instantiated with to the command agent.

```
/* DC to LI operation visibility */
operation_visibility : DC × LI → Bool
operation_visibility(dc, li) ≡
  (
    ∀ c : Command •
      ∃ p : PresentationAgent •
        c ∈ flat_set(dc) ∧
        p ∈ presentations(li) ∧
        ctrl_agent(p) = c
  )
```

Observability

Observability is defined as all relevant information is available to the user, to evaluate the internal state of the system. Observability is connected to browsability, defaults and operation visibility. *Browsability* allows the user to explore the current state of the system via some kind of navigation through the user interface, the property might also be called *accessibility* or *reachability*. The model described here is not able to describe the difference between what is available in an entity or view, and what is currently observable in this entity or view, as for example, in a scrollable window of text. This relation between what is observable and browsable can be addressed with the template model as, for example, done by Roast (1994). Our model is only able to describe browsability, ie what is eventually observable.

Browsability in this model is, when a control class exists for all variables v in the functional specification, in the dialog control specification instantiated (ie an agent c) with the datatype of the variable v.

```
/* DC browsability */
browsability : FC × DC → Bool
browsability(fc, dc) ≡
  (
    ∀ v : Variable •
      ∃ c : CtrlAgent •
        v ∈ vars(fc) ∧
        c ∈ flat_set(dc) ∧
        datatype(c) = datatype(v)
  )
```

This holds for the predefined types. For complex types it is more complicated, as there may be a hierarchy of control agents to represent one complex variable in the functional specification. The hierarchy of control agents will correspond to the structure of the complex type.

Browsability in the logical interaction layer of elements in the dialog control layer is more simple, as it can be expressed as: For all control agents c in the dialog control, which are not a command agent (we do not want to look at command agents as they do not represent variables in the functional specification), at least one presentation agent p in the logical interaction layer must exist, which is connected to the control agent.

```
/* LI browsability */
browsability : DC × LI → Bool
browsability(dc, li) ≡
  (
    ∀ c : CtrlAgent •
      ∃ p : PresentationAgent •
        c ∈ flat_set(dc) ∧
        (~ ∃ id : Id, f : Func • mk_cmd(id,f) = c) ∧
        p ∈ presentations(li) ∧
        ctrl_agent(p) = c
  )
```

Honesty

Honesty, as defined by the IFIP WG2.7, relies on user perception: *Honesty*, the user is able to correctly interpret perceived information. While Dix et al.'s definition is based on system features only: Honesty relates to the ability of the user interface to provide an observable and informative description of internal state changes. If observable is relaxed to browsable, and if we do not consider the user's skills/ability to interpret different data representations, an 'informative description' would be any presentation that in some way adequately represents the internal data type. From this follows that the adequacy proof obligation can be used on the presentation of the internal state. In this formulation brows-

ability and honesty is exactly the same. The reason for this is the use of the same abstract data types in all layers, functional core adapter, dialog control and logical interaction.

Substitutivity

Substitutivity is to allow equivalent values to be substituted for each other. Related to substitutivity is *representation multiplicity*, alternative representations for both input and output,and *device multiplicity*, multiple input and output devices. Substitutivity can be justified by verifying that several types of presentations and actions for presenting information and receiving input exist.

The definition of this property is simple because of the model's partitioning of the user interface into control, presentation and action agents. It is only necessary to count the number of action agents connected to one control agent to know in which way a control agent (a parameter, a variable in the functional specification) can be given input in the logical interaction layer.

```
/* input substitutivity */
substitutivity : CtrlAgent × LI → Nat
substitutivity(c, li) ≡
  card { a | a : ActionAgent • a ∈ actions(li) ∧ ctrl_agent(a) = c }
```

For representation multiplicity the presentation agents connected to a control agent are counted.

```
/* output multiplicity */
output_multiplicity : CtrlAgent × LI → Nat
output_multiplicity(c, li) ≡
  card { a | a : PresentationAgent • a ∈ presentations(li) ∧ ctrl_agent(a) = c }
```

Some kinds of substitutivity cannot be represented in the model, for example when it is possible to enter a number as number literal and as a mathematical expression in the same field.

4 EXAMPLE

Properties are defined by means of the abstract syntax of specifications, whereas the example is stated in the concrete syntax of RAISE, implying that it is not directly possible to apply the property definitions to the example in a direct formal way. Instead, the example's properties will be discussed based on their definition without being strictly formal.

Operation visibility in the dialog control is easily justified by noticing that the dialog control contains a command agent for each function declared in the functional specification. This cannot be justified in the logical interaction specification, because the specification is incomplete.

The functional specification has a state variable *library*. The state of the system can be presented by listing all the contents of the variable library in the user interface. This

way, the system would be observable if all information could be on the screen at the same time. The state of the system is still presented by limiting the presentation to a list of names, the domain of the library variable, and a function to extract the library contents for each name. This way, the system will be browsable.

An important property of the library user interface is that it presents the correct set of unit names ie the set of unit names in the domain of the library variable. This is stated in the completeness axiom in the dialog control specification.

```
[completeness]
{ n | n : name.Index } = dom LibFC.library
```

This also states that adequate 'default' values exist to select from for the single operation input parameter in the this system. Furthermore, the axiom ensures that the domain of the library variable is browsable. The system will neither be honest, nor predictable or observable, if this invariant does not apply to the user interface's list of library names. This is because the user interface could present all possible names of type *Name*, ie present names not in the library or not present names actually in the library.

Expressed as browsability we could say that for all names n in the domain of library, an index exists i in the dialog control, which is equal to n.

```
∀ n : LibFC.Name •
  ∃ i : LibDC.name.Index •
    n ∈ dom LibFC.library ∧
    i = n
```

Proving that all information in the library variable are accessible from the user interface will prove the system to be browsable. Browsability of the library domain has already been justified, the browsability of the range of the library can be written as: for all descriptions *d*, which are in the range of the library, an index *i* and a description entity *desc* exist, for which library of index *i* gives description *d* and the data variable in description entity *desc* equals library of index.

```
[browsability]
∀ d : LibFC.Description •
  ∃ i : LibDC.name.Index, desc : LibDC.description •
    d ∈ rng library ∧
    library(i) = d ∧
    library(i) = desc.data
```

Furthermore, the *LibLI* contains a field class instantiated with LibDC.description, which means that there is a representation for the output in the logical interaction specification.

From the selection axiom it can be seen that there is no real default value, it is possible to invoke an operation without a legal name, when there is no name selected.

```
[selection]
selected ∈ {s | s :Selected • ∃ n : LibFC.Name •
                    n ∈ dom LibFC.library ∧ s = mk_name(n)} ∪ none
```

It can easily be seen from the logical interaction specification that there is no kind of substitutivity or representation multiplicity in the library example, as there is only one presentation agent and one action agent connected to each control agent, but alternative presentation and action agents could have been chosen, to support representation multiplicity and substitutivity.

5 CONCLUSION AND FUTURE WORK

This paper introduces a derived agent model and specification primitives, used for specifying interactive systems and system properties. The paper demonstrates a more operational use of formal specification and justification of interactive systems than what is common in other approaches. This is done by using an architectural model and a set of user interface primitives for specification of specific part of the application. The operational form could guide the construction of the supporting of tools for design of user interfaces, tools for automatic justification of system properties, and tools for automatic generation of design proposals from functional specifications. Our design method is intended for integration of the software design and the user interface design, and as a tool for software engineers.

Using the data refinement concept on the correspondence between the functional core adapter and the user interface has proven useful for justification of system properties, but it is complicated by the fact that the user interface is modeled by an imperative specification and the functional core is modeled by a 'more' applicative specification.

The justification could be extended by taking into account the dynamics of the user interface, for example justifying that operations are actually mapped, or can be mapped to the screen, that operations are enabled when they are available and disabled when they are not, etc.

The notation still needs some refinement, there are problems in writing an easy interpretable specification of agents conforming with the RAISE syntax. Partly because there are no types denoting functions, channels or classes, which would be useful in modeling agents and the dynamics of the user interface. This has made specification in RAISE cumbersome and clumsy.

6 RAISE NOTATION

This section gives a very short introduction to some parts of the RSL notation.

6.1 Basic Types

The RAISE language has a standard set of built-in basic types: **Bool**, **Int**, **Nat**, **Real**, **Char**, **Text**, and **Unit** the singleton type containing a single value '()', the type is used in function signatures of imperative functions, only depending on the state or where the only interesting effect of the function is the way it changes the state, ie functions with no input parameters and/or result parameters. The normal operations can be applied to expressions of the basic types. Complex types as set, list and map are available as well.

6.2 Type definition

A sort - or abstract data type - is a type with no predefined value literals and no predefined operations other than = and $\neq$. A variant definition is short for a sort definition, some value definitions and some axioms. Unions are a way of avoiding the constructors from layered variant definitions.

T	- Sort definition
$T_1 = T_2$	- Derived type definition
$T_1 = \{\mid t \mid P \mid\}$	- Subtype definition
$T == v_1 \mid \ldots \mid v_n$	- Variant definition
$T == c(T_1, \ldots, T_n)$	- Variant definition, with record constructor c
$T == c(d_1 : T_1, \ldots, d_n : T_n)$	- Variant definition, with record destructors $d_1, \ldots, d_n$
$T = VT_1 \mid \ldots \mid VT_n$	- Union definition, on a set of variant types

6.3 Functions

All functions must have a signature describing the types of values taken as input parameters and the types of the values returned as output parameters. Furthermore, the signature specifies whether a function accesses variables for read or write operations, and whether channel values are taken from or sent to channels. Functions can be described by pre- and post conditions, by axioms or explicitly by an algorithm.

$f : T_1 \times \ldots \times T_i \rightarrow T_o \times \ldots \times_n$	- Function signature
$f : T_1 \tilde{\rightarrow} T_2$	- Partial function
$f : T_1 \rightarrow$ **read** v_1 **write** v_2 T_2	- Function accessing variables
$f : T_1 \rightarrow$ **in** c_1 **out** c_2 T_2	- Function communicating on channels
$f(v) \equiv e$	- Function definition
$f(v)$ **as** r **post** e_1 **pre** e_2	- Function definition

REFERENCES

Abowd, G. D. (1991) *Formal Aspects of Human-Computer Interaction.* Ph.D Thesis. Oxford University Computing Laboratory, Oxford.

Brock, S. and George, C. (1990) *RAISE Method Manual.* Computer Resources International A/S.

Coutaz, J. (1987) PAC, an object oriented model for dialog design. In *Human-Computer Interaction INTERACT'87.* North-Holland, Amsterdam, 431-6.

Dix, A. J. 1991 *Formal Methods for Interactive systems.* Academic Press.

Dix, A., Finlay, J., Abowd, G. and Beale, R. (1993) *Human Computer Interaction.* Prentice-Hall.

Duke, D. and Harrison, M. (1993) *Rigorous Development of Interactive Systems.* Technical Report SM/WP16, ESPRIT BRA 7040 Amodeus-2.

IFIP WG2.7 (1995) *Design Principles for Interactive Software.* To be published by Chapman & Hall.

Jones, C.B. (1986) *Systematic Software Development Using VDM* Prentice-Hall International.

Krasner, G.E. and Pope, S.T. (1988) A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3).

Palanque, P.A., Bastide, R., Dourte, L. and Sibertin-Blanc, C. (1993) Design of user-Driven Interfaces Using Petri Nets and Objects. *Proceedings of Conference on Advanced Information Systems Engineering: CAISE'93, Lecture Notes in Computer Science no 685*, Springer Verlag, 569-85.

Roast, C. (1994) Modelling Interaction using Template Abstractions. *People and Computers IX, Proceedings of HCI'94*, 273-84.

The RAISE Language Group (1992) *The RAISE Specification language.* Prentice Hall.

Systä, K. (1994) Specifying User Interfaces in DisCo. *SIGCHI Bulletin* 26(2), 53-8.

UIMS Tool Developers Workshop (1992) A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin*, 24(1), 32-7.

Vanderdonckt, J.M. and Bodart, F. (1993) Encapsulating Knowledge for Intelligent Automatic Interaction Object Selection. *INTERCHI'93 Conference Proceedings*, 424-9.

7 BIOGRAPHY

Ole Lauridsen is currently involved in a research project on design of user interfaces for software development environments, in cooperation with the Technical University of Denmark, and as part of he PhD. He has previously worked on user interfaces for various programming tools, and integration of such tools. His interests are formal methods, programming languages, programming tools, and the design of user interfaces. He has been employed at DDC-I since 1989, and holds a masters degree in engineering.

Discussion

Joëlle Coutaz: You said that the FA was used to adapt the functional core to user needs. But adaptation is not always possible, in particular with regard to functional ordering.

Ole Lauridsen: You are right. I have not looked at sequencing.

Pedro Szekely: What evidence do you have to support your conclusion? What evidence do you have that you didn't fall in the same traps you claimed other system fell into?

Ole Lauridsen: I have no evidence at this time. I do not have any tools, so I do not know if I will fall into traps. I feel that I know many of these traps and that will help me avoid them. There may be others.

Reino Kurki-Suonio: VDM was basically designed with the transformational paradigm in mind. In reactive systems we often need to reason about interleaved or concurrent sequences of operations. Has RAISE got any additional features to allow this?

Ole Lauridsen: I describe agents as communicating processes. RAISE has no special provisions to reason about temporal properties.

Fabio Paterno': How do you perform specification of presentation or layout aspects even in an abstract way?

Ole Lauridsen: I have not yet solved how to do it. I am considering some possible solutions.

Gregory Abowd: The framework for mapping interactive properties onto an architecture is ideal for evaluating the utility of an architecture. This is one part of this work that is a contribution beyond existing formal methods work in this area.

Ole Lauridsen: Thank you.