Published in: Proceedings IFIP Working Groups 8.1/13.2 Conference, Domain Knowledge for Interactive System Design, Geneva 8-10th May, Chapman & Hall Publishers, 1996.

Reasoning about domain knowledge and user actions for interactive systems development

Ralf Möller
University of Hamburg, Computer Science Department,
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, moeller@informatik.uni-hamburg.de

Abstract

In this paper, an approach to specify interactive systems by modeling user actions and domain knowledge is presented. It is shown how Description Logics can be used to formally deduce action concepts at system development time when only incomplete conceptual descriptions of manipulated domain objects are known. We will discuss how types of parameters and values of application functions will be constrained by concepts for user actions which are either automatically derived or interactively selected by the UI designer.

1 INTRODUCTION

For a large class of applications, direct-manipulative interfaces with custom visualizations are required, i.e. the interfaces cannot easily be built with standard elements usually supported by interface builders. Though some UIMS allow static domain objects to be interactively drawn using a picture editor, this approach is not feasible when domain objects and their geometric representations are dynamically computed at runtime. To reduce system development time, applications of this class should be developed by high-level specification techniques instead of writing low-level code for managing the display of geometric objects and for handling mouse events (see Foley and Sukaviriya (1994) and Szekely et al. (1993)).

In this paper, an approach to specify interactive systems by formally modeling user actions and domain knowledge will be presented. It is shown how Description Logics can be used to deduce action concepts at system development time when only incomplete conceptual descriptions of manipulated domain objects are known. We consider an example application (called *XKL*) which is used to interactively configure the interior of an aircraft. In Figure 1 the insertion of a new object into the layout of the aircraft interior is sketched.

The main idea of the interface shown in Figure 1 is that the user selects a prototype object from a set of missing cabin objects. Once the object to be included into the cabin layout is known, possible placement areas are automatically computed according to several kinds of restrictions (weight restrictions, prefabrication restrictions, legal restrictions for emergency exits, etc.). The set of possible placement areas is presented on the screen and the *XKL* user localizes the selected prototype object within one of the placement areas.

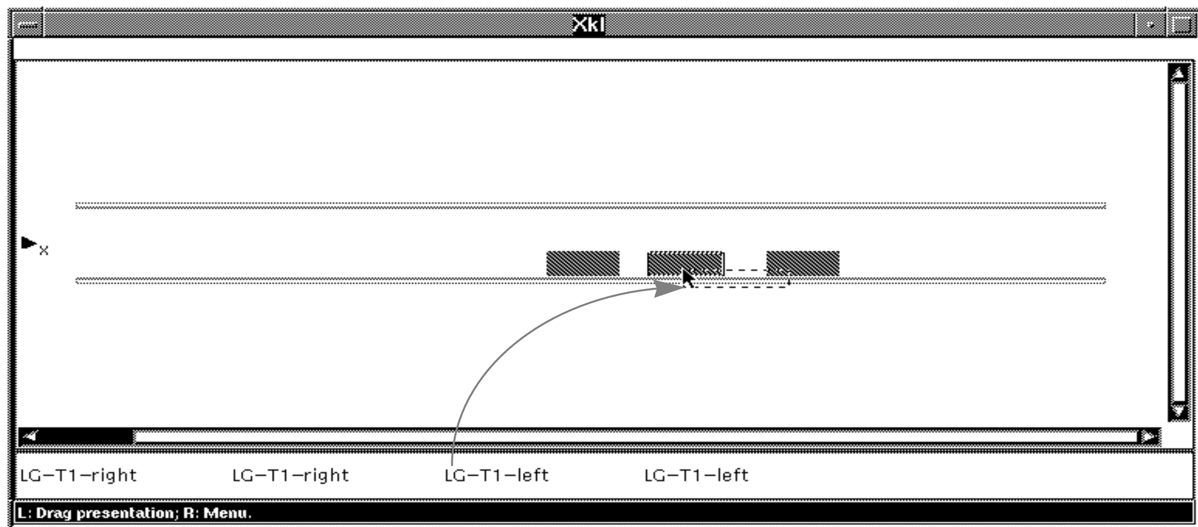

Figure 1 Creation of a new cabin object in *XKL*. The selection of a cabin object from a palette is followed by a localization action. The surface realization of this action sequence is done via drag and drop. After selecting an object from the palette, the possible placement areas for this object are computed and displayed.

Other user actions like the arrangement or movement of cabin objects must be supported as well. For this class of layout problems, a generic set of custom gadgets could be supplied as an extension to a UIMS library. For instance, a special gadget is required for supporting a movement within certain boundaries. However, as the library for the application class gets larger, it will become less and less obvious whether a certain interaction technique required to support user tasks in a specific application already exists (Hartson and Mayo 1995). Furthermore, in order to support the perception processes of the user, additional graphical objects must be presented (e.g. the aircraft body serving as a reference frame). These objects should be automatically derived from domain models at system **development** time. In addition, the presentation of these additional geometric objects must be adapted to the interaction gadgets for user actions. Compare this to intelligent multimedia presentation systems but note that these systems operate at runtime (cf. Wahlster et al. (1993), Arens et al. (1993)). In the spirit of other design environments (see also Johnson et al. (1995)), the goal of the framework presented in this paper (called HAMVIS, HAMburg VIsualization System) is to reduce runtime costs by exploiting models for user actions and domain objects as well as specifications of application functions (type signatures) to generate code for a UIMS.

Recent research on Human-Computer Interaction has shown the problems of prototyping approaches for system development (Neches et al. 1993). On the one hand, domain knowledge required for the implementation of application functions is influenced by interface design constraints (e.g. for a movement action, it must be possible to compute the possible positions in beforehand). On the other hand, domain knowledge also influences the selection of appropriate interaction techniques. This paper describes a new approach to formally model these influences using the derivation processes of Description Logics. In contrast to other approaches which also model user actions with task models for interface design (see e.g. Hartson et al. 1993), the approach presented in this paper is used (i) to structure the interface and (ii) to automatically derive the contents of graphical interfaces based on **geometric** information about domain objects rather than standard interaction gadgets (see e.g. Puerta et al. 1994) or objects known from drawing programs (see e.g. Szekely et al. 1993).

2 THE HAMVIS SYSTEM DEVELOPMENT SCENARIO

The idea of HAMVIS is to derive the contents of visualizations from communication knowledge and to constrain the development of models for domain knowledge by interface and presentation requirements. The actual presentation of domain objects is done at runtime, but the way the objects are presented and the whole dialog structure, i.e. the user actions supported within a specific window to be shown at runtime, can be defined at development time.

In order to derive domain objects required to support a direct-manipulative interaction style, user actions are not modeled at the UIMS level of gestures applicable to graphical objects (press mouse button, move mouse, release mouse button) but at the level of manipulations of domain objects. Because domain objects playing part in an interaction at runtime are usually not completely known at development time, HAMVIS must deal with conceptual information about domain objects which are actually computed at runtime by "application functions." In this paper, the term "application function" is used as a synonym for a procedure (defined with input parameters and values) that must not require user interaction. Types of parameters and values have to be specified at system development time but, from the viewpoint of HAMVIS, the body of an application function is treated as a black-box. Application Functions can be implemented using a programming language or by knowledge-based inference services. In the *XKL* example discussed in this paper, a knowledge-based configuration system could be used (see Kopisch and Günter 1992). An application function will compute the placement areas required for the interface in Figure 1. The input to this function is a cabin object selected by a user action prior to the evaluation of the application function at runtime. Thus, a user action (selection) provides the input for an application function which, in turn, produces values required for another user action (localization).

The basic knowledge base of HAMVIS (HAMVIS Upper Model) provides a generic model for user actions encountered in an application class. The combination of user actions and application functions is part of the domain knowledge to be acquired for a specific application. Figure 2 shows a system development scenario for *XKL*. In this paper, only the gray parts are discussed.

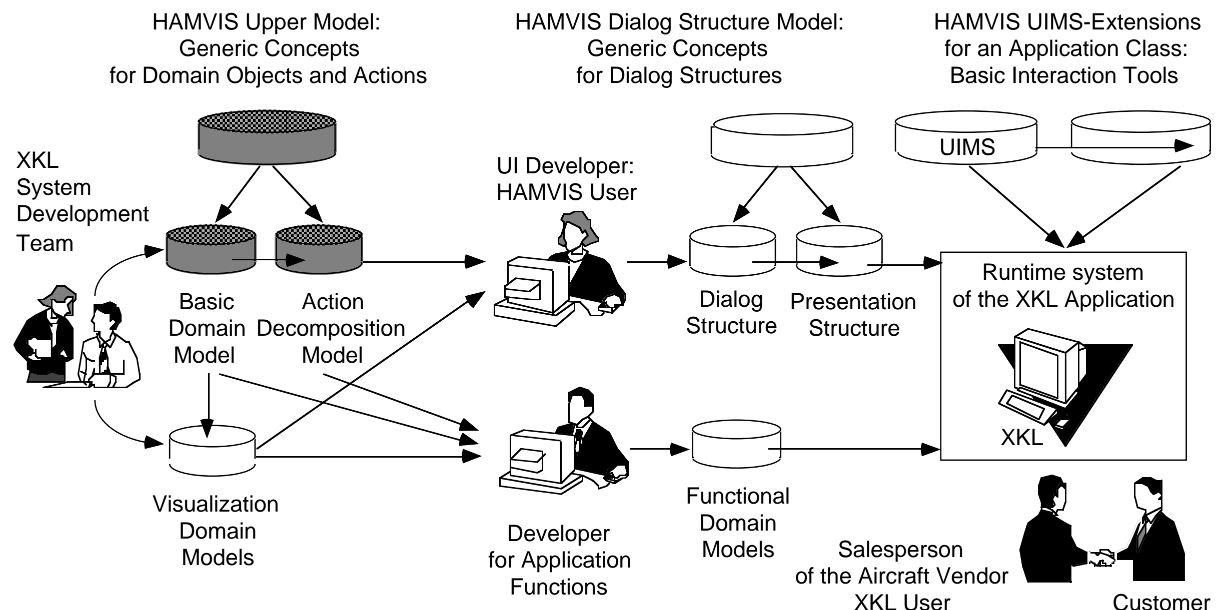

Figure 2 Complete HAMVIS system development scenario.

Domain knowledge is modeled by the System Development Team. Using concepts of the HAMVIS Upper Model, the System Development Team defines a Basic Domain Model (in the case of *XKL* with concepts like “Aircraft”, “Cabin”, “Cabin object”, “Galley” etc.). Afterwards, the structure of the application is defined in terms of actions. The System Development Team describes the combination of application functions and user actions in an Action Decomposition Model. User actions that can be performed at runtime are described at system development time using concepts from the HAMVIS Upper Model.

HAMVIS supports a division of labor between user interface development and implementation of application functions. Once the modeling of user actions is finished, types of parameters and values of application functions are defined. They can be implemented separately while the interface designer uses other services of HAMVIS and its interactive interfaces to structure the final application interface for *XKL*. Note that in the scenario there are several kinds of users: The aircraft salesperson is the end-user of the *XKL* application (at runtime) while the developer of the interface components of *XKL* is the HAMVIS user (at system development time).

The HAMVIS user models additional knowledge about the application (dialog structure, presentation structure) in such a way that action specifications can be mapped onto services of a UIMS. The UIMS is responsible for managing the interaction cycle: drawing and erasing objects on the screen, handling of gestures etc. The interface presented in Figure 1 has been created by the HAMVIS prototype. Due to space limitations, the additional models defined by the HAMVIS user and the associated inferences of HAMVIS are not presented in this paper (see Möller 1996). We will focus on the Action Decomposition Model which defines the basis for the mapping process.

2.1 Knowledge about actions in a class of applications: HAMVIS Upper Model

In order to provide conceptual knowledge about user actions, HAMVIS defines a basic model to structure conceptual information required for applications of a certain class. Domain objects and generic actions defined on these objects are modeled using a Description Logic (DL) representation formalism because, during development time, formal derivations with incomplete conceptual information must be supported. The paper assumes basis knowledge about DL (see e.g. Woods and Schmolze (1992) for an introduction). The knowledge bases presented in this paper are based on a concrete DL implementation: the CLASSIC system (Resnick et al. 1993). The basic knowledge base is called HAMVIS Upper Model. In the spirit of the Penman Upper Model (Bateman et al. 1990), this knowledge base structures basic concepts for an application class in a way that is suited to communication purposes. However, in this paper, we limit the presented section of the HAMVIS Upper Model to the part that is required for understanding action concepts used in the *XKL* application. Shortened excerpts of logical formulae from knowledge bases are presented using the KRSS-Syntax for CLASSIC (Patel-Schneider and Swartout 1993) with small extensions that are explained when required.

Spatial objects and their decomposition
For applications like *XKL*, knowledge about spatial objects and their decomposition is modeled as shown in Figure 3. Spatial objects are defined with a three-dimensional geometric representation (canonical form representation) and one or more projective forms. The decomposition of objects into parts is modeled with a role hierarchy. Currently, the meronymic Component/Object relation and a spatial inclusion relation are supported (cf. Winston

et al. 1987). For special spatial objects like rigid objects (physical objects) and spatial regions, the part-of relations are restricted (see Figure 3).
Domain concepts must be subconcepts of the predefined concepts as will be discussed below. The definitions of Figure 3 are required for defining conceptual knowledge about actions. Before we discuss detailed action concepts, we first have to deal with action decomposition.

```
(define-primitive-concept um-object) ; um-object ≈ upper-model-object

(define-primitive-role has-decomposition nil)
(define-primitive-role has-constituents has-decomposition)
(define-primitive-role spatially-encloses has-decomposition)
(define-primitive-role has-canonical-geometric-form-representation nil)
(define-primitive-role has-projective-form nil)

(define-primitive-concept spatial-position-constraint ...)

(define-disjoint-primitive-concept spatial-object (object-kinds)
    (and um-object
         (at-least 1 has-canonical-geometric-form-representation)
         (at-most 1 has-canonical-geometric-form-representation)
         (all has-projective-form projective-form)
         (at-least 1 has-projective-form)
         ...))

(define-disjoint-primitive-concept physical-object (kinds-of-spatial-objects)
  (and spatial-object
       (all spatially-encloses spatial-object)
       (all has-constituents spatial-object)))

(define-disjoint-primitive-concept spatial-region (kinds-of-spatial-objects)
  (and spatial-object
       (all spatially-encloses spatial-object)
       (all has-constituents spatial-object)))
```

Figure 3 Fragment of the HAMVIS Upper Model used to represent spatial objects.

Conceptual knowledge about action decomposition
User actions and application functions provide the basis for modeling an application. Application functions are evaluated at runtime and compute the objects that can be manipulated by user actions. In HAMVIS, elementary user actions do not have side effects on application objects but return new objects (e.g. a new position as shown in Figure 1) which are stored by subsequent application functions. The composition of user actions and their associated application functions can be interpreted as a higher-level user action (composite action), i.e. at this level, application functions are considered transparent.

An application can be composed of several activities, each of which is required for solving a different subproblem of the whole application. At runtime, a subproblem can be solved by selecting among several alternative composite actions until a certain goal is satisfied. For instance, the *XKL* user can either create a cabin object, move a cabin object, or delete a previously created cabin object etc. An activity models a very high-level user action with alternative subactions available in an interaction loop. Figure 4 shows the concept definitions used to model these different levels of actions in terms of Description Logic.

```
(define-primitive-role has-substep has-decomposition)

(define-disjoint-primitive-concept atomic-action (action-hierarchy-level)
  (atmost 0 has-substep))

(define-disjoint-primitive-concept composite-action (action-hierarchy-level)
    (and (atleast 1 has-substep)
         (all has-substep atomic-action)))

(define-disjoint-primitive-concept activity (action-hierarchy-level)
  (and (atleast 1 has-substep)
       (all has-substep composite-action)))

(define-disjoint-primitive-concept application (action-hierarchy-level)
  (and (atleast 1 has-substep)
       (all has-substep activity)))

(define-disjoint-primitive-concept user-action (action-agent)
  atomic-action)

(define-disjoint-primitive-concept application-function (action-agent)
  atomic-action)
```

Figure 4 Action types and their definitions.

The idea behind the four layers can be understood by considering the rough mapping onto UIMS services. Each activity requires its own interaction window, possibly with several panes (see Figure 1). Thus, at runtime, an activity is realized by a window of the UIMS host system and an interaction loop. Composite actions are mapped onto commands that are preferrably realized using custom gadgets or standardized interaction techniques like drag and drop. User actions define the visual “material” to be presented at runtime in various panes. In addition, the type of interaction pane required for a user action is also restricted by specific action concepts as will be discussed in the next section.

```
(define-primitive-role choice-set nil)
(define-primitive-role choice choice-set)

(define-primitive-action-concept selection
    (and user-action
         (at-least 1 choice-set) (all choice-set um-object)
         (at-least 1 choice) (all choice um-object))
  (:input choice-set)
  (:output choice)))

(define-action-concept single-object-selection
    (and selection (at-most 1 choice)))

(define-action-concept selection-from-small-palette
    (and single-object-selection (at-most 10 choice-set))
  (:parameters-to-present choice-set)
  (:pane-types (choice-set singe-choice-palette-pane)))
```

Figure 5 Modeling knowledge about selection actions using case roles.

Conceptual models for elementary user actions

Reasoning about actions is required at development time where only conceptual information about objects (to be manipulated at runtime) is available. We model actions using case roles by assuming that instances of actions will be related to instances of domain concepts. Figure 5 shows some of the declarations of the HAMVIS knowledge base for selection actions.The declaration `define[-primitive]-action-concept` is similar to `define[-primitive]-concept` from KRSS except that additional options required for modeling actions are supported. Considering the case role structure of an action it should be clear that some of the case roles must be filled before an action can actually be performed (the option `:input` specifies these case roles). Carrying out an action will set newly created objects into relation to the action instance using other case roles (option `:output`). The basic selection concept is primitive. However, more specific selection actions are modeled using defined concepts with necessary and sufficient conditions (see Figure 5). The same techniques are used to define localization actions (Figure 6).

```
(define-primitive-role has-manipulated-object nil)

(define-concept spatial-action
    (and user-action (all has-manipulated-object spatial-object)))

(define-primitive-role has-localized-entity has-manipulated-object)
(define-primitive-role has-reference-object nil)
(define-primitive-role has-previous-position-constraint nil)
(define-primitive-role has-new-position-constraint nil)

(define-primitive-action-concept localization
    (and user-action
         (at-least 1 has-localized-entity) (all has-localized-entity um-object)
         (at-least 1 has-reference-object) (all has-reference-object um-object)
         (at-least 1 has-new-position-constraint)
         (all has-new-position-constraint position-constraint))
   (:input has-localized-entity has-reference-object)
   (:output has-new-position-constraint))

(define-action-concept spatial-localization
    (and localization
         (all has-localized-entity spatial-object)
         (all has-reference-object spatial-object)))
(define-rule r1 spatial-localization
   (and (at-least 1 has-previous-position-constraint)
        (all has-previous-position-constraint spatial-position-constraint)
        (all has-new-position-constraint spatial-position-constraint))))

(define-action-concept spatial-localization-in-xy-bounding-rectangle
    (and spatial-localization
         (at-most 1 has-localized-entity) (all has-le-ro-relation in)
         (all has-previous-position-constraint xy-bounding-rect-pos-constraint))
  (:parameters-to-present has-reference-object))
(define-rule r2 spatial-localization-in-xy-bounding-rectangle
  (and (all has-new-position-constraint xy-bounding-rect-pos-constraint)
       (all has-reference-object non-overlapped-spatial-object)))
```

Figure 6 Modeling knowledge about localization actions using case roles.

Very specific actions are constrained in such a way that interface elements to support the action can be generated (see the definition of `selection-from-small-palette` in Figure 5). Case roles that contain objects that have to be presented at the interface are declared with `:parameters-to-present`. The pane type for the interaction object is given for each of these case roles with `:pane-types`. Rules are used to define additional constraints on actions. Note that a rule does not model a logical implication because of the unidirectional way of inference.

2.2 Modeling domain knowledge using the HAMVIS Upper Model

The input case roles of the action concepts defined above impose constraints on the objects that are manipulated or operated upon. Thus, if application objects are to be manipulated by these actions, they must inherit from the concepts defined in the HAMVIS Upper Model. In Figure 7 a small subset of the declarations found in the *XKL* knowledge base is shown.

```
(define-primitive-concept cabin-object spatial-object)

(define-primitive-role has-trolleys nil)
(define-primitive-concept trolley spatial-object)

(define-primitive-concept galley
  (and cabin-object physical-object (all has-trolleys trolley)))

(define-primitive-concept lavatory
  (and cabin-object physical-object))

(define-primitive-concept seat
  (and cabin-object physical-object))

(define-primitive-concept placement-area spatial-region)

(define-primitive-concept aisle (and cabin-object spatial-region))
```

Figure 7 Excerpt from the basic domain knowledge base for *XKL*.

These concepts are defined by the *XKL* System Development Team. Though generic knowledge about action concepts is defined in the HAMVIS Upper Model, the system development team must define the structure of the application by modeling the action decomposition. This will be discussed in the next chapter.

3 INFERENCE SERVICES FOR ACTION MODELING

HAMVIS supports a four-level model to structure interactive applications from the viewpoint of user actions (see also the approach of Philips et al. 1988). Actions at lower levels define the decomposition of higher-level actions. Since direct-manipulative interaction styles are supported, application functions found at the lowest level are transparent at the level above.

At system development time, the current application being modeled with HAMVIS can be treated as an instance of the concept `application` of the HAMVIS Upper Model (see Figure 4). The activities of the application are also modeled with instances (of the concept `activity`). These instances can be considered as development time prototypes used to gather information about action possibilities at runtime. An activity supports a set of alternative

actions. However, the actual action sequence (e.g., create a galley g1, create a lavatory l1, move galley g1, create a galley g2, move lavatory l1, delete galley g1, etc.) performed by the user at runtime is not known at development time. Saying "an activity has composite actions a,b,c as alternatives" means "the runtime action sequence is described by the regular expression (a | b | c)*." Again, information about composite actions must be gathered at development time. For each composite action (e.g. "create cabin object," "delete cabin object," "move cabin object") this can also be done using a prototype instance whose concept describes the corresponding runtime actions. For composite actions, the substeps are application functions or user actions (see again the definitions in Figure 4). The substeps of the composite action `create-cabin-object` discussed in Figure 1 are "compute a set of missing prototype cabin objects", "select a cabin object from this object", "compute possible placement areas from the selected object", "localize the cabin object with one of the possible placement areas", "store the new position constraint". Conceptual information about each of these actions can be described using a prototype action of type `application-function` or `user-action`. For the decomposition of a composite action we need additional information to represent the order of the five substeps mentioned above. The *XKL* application in general can be described by the tree in Figure 8.

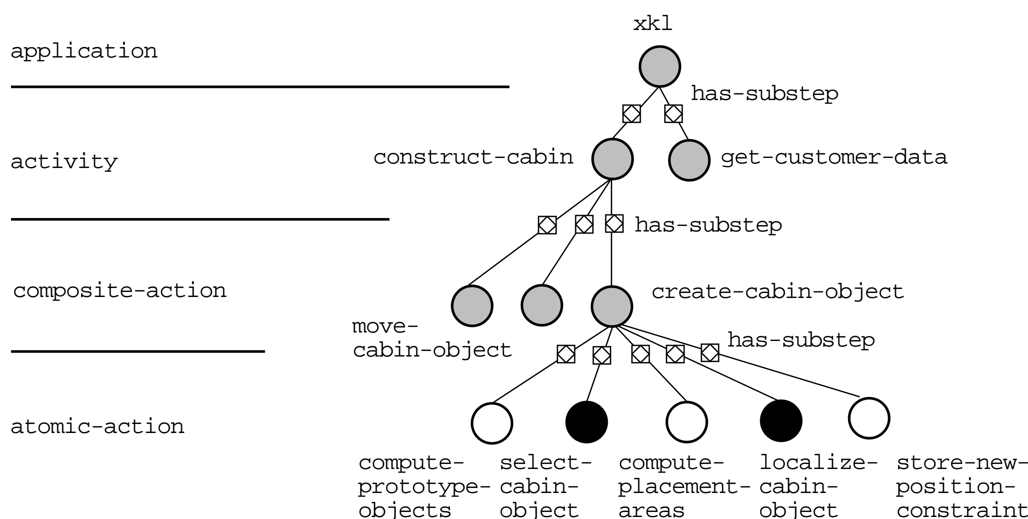

Figure 8 Decomposition of the *XKL* application (application functions and user actions are represented with white and black circles, respectively).

Actions of different modeling layers are represented by ABox instances indicated as circles (concepts of instances are represented by the names to the left). For user actions, more specific subconcepts are defined in the HAMVIS Upper Model. Thus, user actions must be modeled in a more detailed way. This information is required to generate adequate graphical objects required for actually carrying out an action at runtime.

In the next sections, we will see how conceptual information about parameters and values of application functions can be used to constrain the action concepts that can be chosen by the HAMVIS user (at development time!) for a detailed description of actions that can be performed by the *XKL* user (at runtime!). On the other hand, if application functions compute domain objects that are manipulated by a user action, the action concept constrains the concepts (or types) of the objects which application functions can compute. In the following, we

will see how Description Logic inference techniques can be used to model these interdependencies.

The action model for an application is a logical model stated in terms of ABox assertions. However, this ABox model can be interactively defined by the HAMVIS user with the HAMVIS Action Decomposition Interface. The graphical interface makes the assertions (and retractions) of logical formulae transparent to the HAMVIS user.

3.1 Interactive action decomposition

The Action Decomposition Interface supports the process of modeling user actions using graphical interaction techniques. The HAMVIS user defines the types of parameters and values of application functions (in terms of concepts from the basic domain model). If the values of application functions are to be manipulated by user actions, the set of actions applicable to these domain objects is restricted. The idea of HAMVIS is to let the HAMVIS user select the general type of action (e.g., selection, movement, localization) and to automatically compute the specific action concept that is suited to the objects to be manipulated. If the conceptual information given at development time is specific enough for determining by what graphical interaction technique an action can be realized at runtime, the model for a user action is complete.

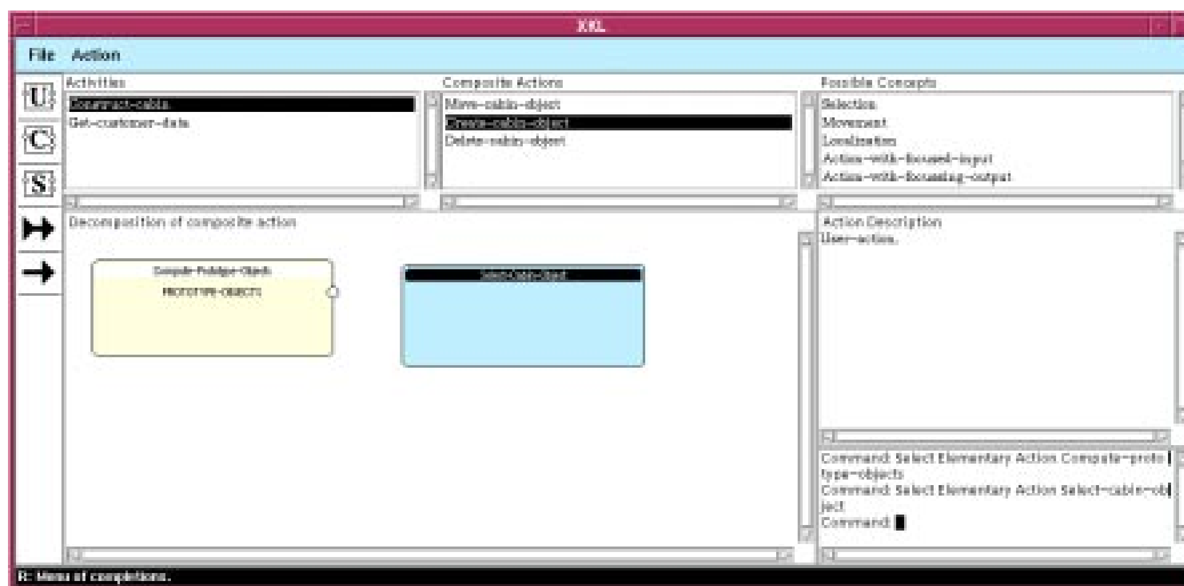

Figure 9 Decomposition of the *XKL* application.

In Figure 9, the decomposition of the *XKL* application into two activities (left upper table) is shown. This decomposition has been defined by the HAMVIS user. For the selected activity `construct-cabin`, the decomposition in terms of composite actions is presented in the middle table. In this table, the composite action `create-cabin-object` is selected. The final graphical interface for the composite actions has been presented in Figure 1. To define the decomposition of the selected composite action, application functions can be dragged as graphical objects (icon "C") from the toolbox into the main pane. Parameters to these functions can be defined by dragging the arrows (input-output arrow and output-only arrow) found in the toolbox onto an application function. Parameters and values of application functions are

treated in a similar way as input and output case roles of user actions, i.e. function values are called output parameters. The `Action Description` pane (middle pane to the right) serves as a viewer and editor for conceptual information about application functions and user actions. Let us assume, the HAMVIS user specifies that the function `compute-prototype-objects` returns domain objects (via the role `prototype-objects`) which are subsumed by the concept `cabin-object` (see Figure 7). Role restrictions inherited by `cabin-object` are also presented in this pane (minimum and maximum cardinalities in square brackets). Concept as well as role restrictions can be edited by clicking onto the names or numbers.

In Figure 9 a user action (icon "U") called `select-cabin-object` has also been dragged into the main pane. The topmost table to the right (`Possible Concepts`) presents the selectable possible specializations for the action. Let us assume, the HAMVIS user clicks at the concept `selection`. Selections impose constraints on the roles `choice-set` and `choice` (see the concept definition in Figure 5). The role `choice-set` is used to describe the input parameters and `choice` defines the output (or "effect") of the user action. These parameters are automatically shown in the graphical presentation (see Figure 10 which already shows the complete decomposition of `create-cabin-object`).

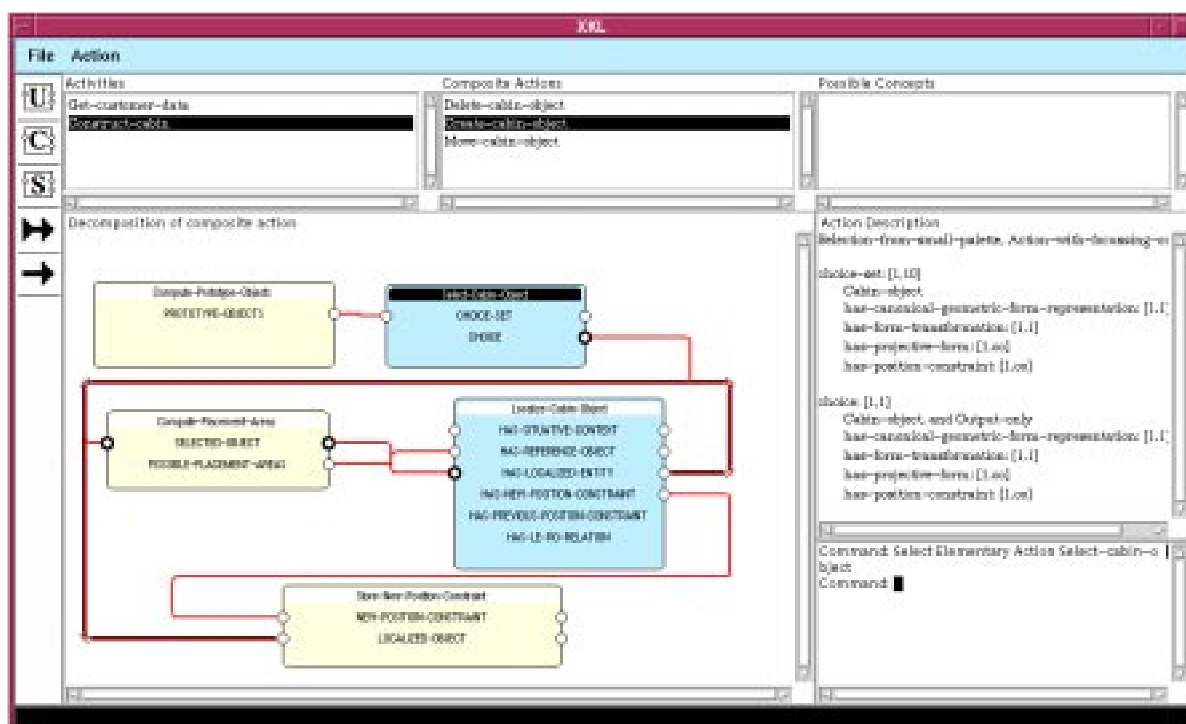

Figure 10 Connections between ports with propagation of conceptual information. The `Action Description` pane shows conceptual information about the selected action (indicated by a black bar).

The graphical representation of atomic actions, i.e. application functions and user actions, shows little ports with small circles to the left (input roles) and to the right (output roles). All input roles are also used as output roles. These ports can be interactively connected using the mouse indicating e.g. that the objects computed by `compute-prototype-objects` (and available in these ports at runtime) are exactly those objects that can be selected by the user (`choice-set`) once they are presented on the screen. The connection pipes are routed automatically.

Because the objects computed by the function play a certain role in the user action, the concept information (`cabin-object`) given for `prototype-objects` must be propagated to the case role `choice-set` of `select-cabin-object`. This is shown in Figure 10 in the `Action Description` pane. The pane shows the conceptual information that can be inferred for the selected action by connecting it with the application function.

3.2 Derivation of conceptual information about user Actions: The internal view

The inference steps behind the Action Decomposition Interface are realized by the ABox reasoning mechanism of the CLASSIC system. In Figure 11 the Abox statements modeling the knowledge about `create-cabin-object` are presented as a tree.

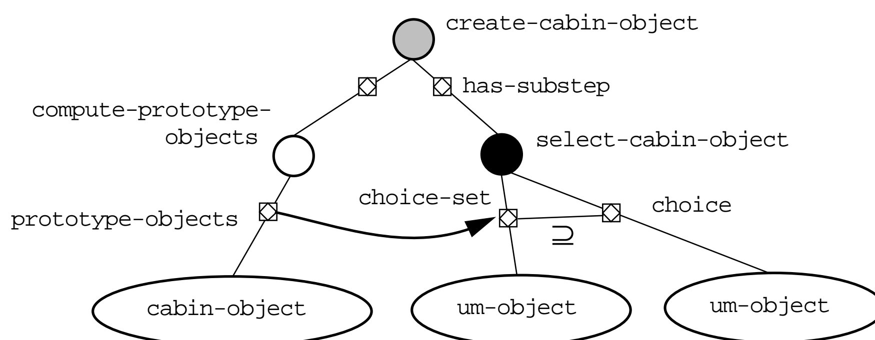

Figure 11 Representation of the *XKL* action model using ABox instances before role restriction propagation. `Um-object` is the root of the HAMVIS Upper Model. The graphical pipe between the roles `prototype-objects` and `choice-set` is indicated with an arrow.

The Abox declarations are automatically generated by the HAMVIS user interface as a consequence of dragging an icon from the toolbox into the main pane. For instance, the insertion of `select-cabin-object` results in the following statements:

```
(define-distinct-individual select-cabin-object)
(state (instance select-cabin-object user-action))
(state (related create-cabin-object select-cabin-object has-substep))
```

For each instance to be generated, the user is asked to enter a name which is used for identification purposes only. The mouse click at the action concept `selection` presented in the table `Possible Concepts` is expressed in ABox terms as:

```
(state (instance select-cabin-object selection))
```

For the action case roles, the concept restrictions shown in Figure 11 are either given by the HAMVIS user (e.g. for the role `prototype-objects`) or defined by the HAMVIS action knowledge base (roles `choice-set` and `choice`, see Figure 5).

The action knowledge base in Figure 5 defines the role `choice` as a subrole of `choice-set`. Thus, for every selection, the fillers of `choice` are a subset of the fillers of `choice-set`. Besides concept propagation, ABox reasoning ensures the correct propagation of cardinality information. Maximum restrictions on `choice-set` are propagated to `choice` and minimum restrictions of `choice` are propagated to `choice-set`.

Role restrictions of `compute-prototype-objects` for the role `prototype-objects` must also be asserted as role restrictions of `select-cabin-object` for the role `choice-set` and vice versa. The following ABox statements are automatically generated:

```
(state (instance select-cabin-object (all choice-set cabin-object)))
```

```
(state (instance compute-prototype-objects (and (all prototype-objects um-object)
                                                (at-least 1 prototype-objects))))
```

We can see that by generating these assertions, conceptual information is propagated along the pipes in both directions. Whenever an instance's restrictions on a role change, the conceptual information is propagated to the "partner role" at the other instance by computing corresponding ABox assertions as indicated above. The order of the substeps of a composite action is partially defined by the data-flow direction of the pipe between two roles (see the arrow in Figure 11).

The propagation mechanism has been implemented as an extension to the CLASSIC system using its metalevel facilities. The role restriction propagation mechanism described above ensures that the same restrictions are asserted to all roles (ports) connected with pipes. In the following we will see how the propagation mechanism will be used to model inference schemata for action modeling.

The complete description of the composite action `create-cabin-object` is also presented in Figure 10. The selected object to be found in the role `choice` of `select-cabin-object` is used as an input parameter to an application function `compute-placement-areas`. The type of the input parameter of this function is automatically constrained to be `cabin-object` because of the pipe connection to the role `choice` of `select-cabin-object`.

Let us assume, the HAMVIS user has defined `placement-area` as the type of the output parameter `possible-placement-areas`. The output parameters of `compute-placement-areas` are connected to a user action `localize-cabin-object`. Let us further assume, for this user action the HAMVIS user has chosen the concept `localization` from the set of possible concepts shown in the interface (see Figure 9). The role restriction propagation mechanism described above propagates the conceptual information known about the output parameters of `compute-placement-areas` to the respective roles of `localize-cabin-object` (see the pipe connections in Figure 10). Thus, role fillers of `has-localized-entity` are known to be instances of `cabin-object` and fillers of `has-reference-object` are `placement-areas`. Both concepts inherit from `spatial-object` (see the knowledge bases in Figure 7 and Figure 3). This is enough information to automatically infer that `localize-cabin-object` is not even a (simple) `localization` but even a `spatial-localization` (see the concept definition with necessary and sufficient conditions in Figure 6). The rule `r1` triggering on `spatial-localization` imposes additional constraints on the other roles of `localize-cabin-object` (see again Figure 6). The HAMVIS user can select between more specific subconcepts of `spatial-localization` (e.g., `spatial-localization-in-xy-bounding-rectangle`, see Figure 6). The additional constraints are propagated to the final application function `store-new-position-constraint`. The `localized-object` will be a `cabin-object` and the `new-position-constraint` will be at least a `spatial-position-constraint` because of the rule `r1`. If the HAMVIS user selected `spatial-localization-in-xy-bounding-rectangle` as a more specific concept, `new-position-constraint` would be `xy-bounding-rect-pos-constraint` (rule `r2`, see the knowledge base in Figure 6). Due to rule `r2`, the fillers of `has-reference-object` must inherit from `non-overlapped-spatial-object`. This information will be propagated along the pipes. As a consequence, the initial application function `compute-prototype-objects` must also return instances that are subsumed by this concept. Thus, action concepts determine the type signature of application functions.

At development time, the action decomposition model defines "what" has to be shown at runtime (see the `:parameters-to-present` declarations in the action knowledge bases in

Figure 5 and Figure 6). The general structure of the application (required panes and their classes, graphical perspectives, drawing attributes) can be defined in terms of actions such that a mapping to UIMS services is possible.

4 Conclusion

The paper has shown how Description Logic reasoning can be used to model the interdependencies between parts of the interface design model (user actions) and the implementation model (type signatures of application functions). More specific action concepts lead to interaction components that are better suited to the domain objects to be manipulated (see Figure 1: direct-manipulative interaction style rather than textual input of coordinates). More specific user action concepts provide more information on how to design adequate visualizations to support the user.

Note that the goal of HAMVIS is not to manually construct design environments (Fischer and Nakakoji 1991, Fischer et al 1994). The focus is on the automatic derivation of graphical interface components (see the HAMVIS knowledge bases in Figure 2) from a specification of user actions, domain objects and application functions. The example presented in Figure 1 which has been generated by the HAMVIS prototype indicates that this can be achieved. The final composition of visualizations is implemented by other components of the HAMVIS framework (see Figure 2 and Möller 1996).

In this paper, it has been shown how the role restriction propagation mechanism can be used to formally deduce specialized action concepts at system development time when only concepts of manipulated domain objects are known. Furthermore, types of parameters and values of application functions to be implemented will be constrained when specific user action concepts are interactively selected by the interface developer.

5 Acknowledgments

I would like to thank Bettina Brüning who implemented large parts of the Action Decomposition Interface of HAMVIS and the role restriction propagation mechanim as part of her diploma thesis. I am also grateful to Volker Haarslev, Bernd Neumann and the referees for thoughtful comments on this paper.

6 Bibliography

Arens, Y., Hovy, E.H., van Mulken, S., Structure and Rules in Automated Multimedia Presentation Planning, in *Proceedings IJCAI'93*, August 1993.

Bateman, J.A., Kasper, R.T., Moore, J.D., Whitney, R.A., A General Organization of Knowledge for Natural Language Processing: the Penman Upper Model, ISI-Report, March, 1990.

Fischer, G., Nakakoji, K., Empowering Designers with Integrated Design Environments, in *Proceedings of the First International Conference on Artificial Intelligence in Design*, Royal Museum of Scotland, Edinburg, 1992.

Fischer, G., McCall, R., Ostwald, J., Reeves, J., Shipman, F., Seeding, Evolutionary Growth and Reseedings: Supporting the Incremental Development of Design Environments, in *Proceedings CHI'94 Human Factors in Computing Systems*, ACM Press, 1994, pp. 292-298.

Foley, J.D., Sukaviriya, P.N., History, Results, and Bibliography of the User Interface Design Environment (UIDE), an Early Model-Based System for User Interface Design and Implementation, in *Interactive Systems: Design, Specification, and Verification*, 1st Eurographics Workshop, (ed. Paternó, F.), June 1994, Springer, 1995, pp. 3-14.

Hartson, H.R., Siochi, A.C., Hix, D., The UAN: A User-Oriented Representation for Direct-Manipulation Interface Design, in *ACM Transactions on Information Systems*, Vol. 8, Nr. 3, July 1990, pp. 181-203.

Hartson, H.R., Mayo, K.A., A Framework for Precise, Reusable Task Abstraction, in *Interactive Systems: Design, Specification, and Verification*, 1st Eurographics Workshop (ed. Paternó, F.), June 1994, Springer, 1995, pp. 279-297.

Johnson, P., Wilson, S., Johnson, H., Scenarios, Task Analysis and The ADEPT Design Environment, in *Scenario-Based Design* (ed. Carroll, J.), Addison-Wesley, 1995.

Kopisch, M., Günter, A., Configuration of a Passenger Aircraft Cabin Based on a Conceptual Hierarchy, Constraints and Flexible Control, in *Proceedings IEA/AIE-92 Industrial and Engineering Application of Artificial Intelligence and Expert Systems* (eds. Belli, F., Radermacher, F.J.), Springer, 1992, pp. 421-430.

Möller, R., HAMVIS: Generation of Visualizations in a Framework for User Interface Development, in German, Dissertation, University of Hamburg, 1996.

Neches, R., Foley, J., Szekely, P., Sukaviriya, P., Luo, P., Kovacevic, S., Hudson, S., Knowledgeable Development Environments Using Shared Design Models, in *Proc. ACM/ AAAI International Workshop on Intelligent User Interfaces*, Jan., 1993.

Patel-Schneider, P.F., Swartout, B., Description Logic Specification from the KRSS Effort, *http://ksl.stanford.edu:/pub/knowledge-sharing/papers/dl-spec.ps*, 1993.

Philips, M.D., Bashinski, H.S., Ammerman, H.L., Fligg, C.M., A Task Analytic Approach to Dialog Design, in *Handbook of Human-Computer Interaction* (ed. Helander, M.), Elsevier Science Publishers, B.V. (North Holland), 1988, pp. 835-857.

Puerta, A.R., Eriksson, H., Gennari, J.H., Musen, M., Beyond Data Models for Automated User Interface Generation, in *People and Computer IX, Proceedings of HCI'94* (ed. Cockton, G., Draper, S.W., Weir, G.R.S.), Glasgow, August 1994, pp. 353-366.

Resnick, L.A., Borgida, A., Brachman, R.J., McGuiness, D.L., Patel-Schneider, P.F., Zalondek, K.C., CLASSIC Description and Reference Manual for the Common Lisp Implementation, Version 2.2, 1993.

Szekely, P., Luo, P., Neches, R., Beyond Interface Builders: Model-Based Interface Tools, in *Proceedings of INTERCHI'93*, April 1993, pp. 383-390.

Wahlster, W., André, E., Finkler, W., Profitlich, H.-J., Rist, T., Plan-Based Integration of Natural Language and Graphics Generation, *Artificial Intelligence*, 63, 1993, pp. 387-427.

Winston, M.E., Chaffin, R., Herrman, D., A Taxonomy of Part-Whole Relations, *Cognitive Science*, 11, 1987, pp. 417-444.

Woods, W.A., Schmolze, J.G., The KL-ONE Family, in *Semantic Networks in Artificial Intelligence* (ed. Lehmann, F.), Pergamon Press, 1992, pp. 133-177.