A UML-Based Design Environment for Interactive Applications

Paulo Pinheiro da Silva and Norman W. Paton

Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, England, UK.
e-mail: {pinheirp,norm}@cs.man.ac.uk

Abstract

The Unified Modeling Language (UML) can be used for modelling both the structure and behaviour of software applications. However, although UML supports many different modelling notations, minimal support is provided for user interface (UI) design. The Unified Modeling Language for Interactive Applications (UMLi) is an extension of UML that provides support for UI design. UMLi has a user interface diagram for modelling abstract UI presentations and an extended activity diagram that provides constructors for modelling common UI behaviours. This paper presents the support provided for UI design by the UMLi design environment. Designers can use the environment to model applications and their UIs using UML and its extensions in UMLi. The tool provides facilities for modelling interaction objects, and the collaboration of these interaction objects with domain objects.

1 Introduction

UML [4] has quickly become established as the principal object modelling language, and as an official standard, benefits from the endorsement of the OMG [15]. However, although UML can be seen as quite a comprehensive modelling language, with many interrelated diagrams for describing the structure and behaviour of an application at different levels of abstraction, very little attention has been paid to the needs of user interface designers in UML. As a result, attempts to develop user interfaces in UML tend to come up against modelling challenges that are a barrier to the natural expression of interface features [12, 7, 17].

The need for effective, abstract modelling facilities for user interfaces has long been recognised [14, 3], and a research community has been working on user interface management systems [11] and model-based user interface development environments (MB-UIDEs) [21, 6] for over 10 years. However, although this research activity has identified effective techniques for modelling tasks and dialogue, interface design methods have generally been poorly integrated with other aspects of application design, and there are few widely accepted models or notations.

The aim of the UML*i* project is to investigate techniques for easing the design of user interfaces in UML. A case study in the use of standard UML for modelling user interface applications [7] revealed various problems describing user interface functionalities using the existing facilities of UML. This has given rise to the design of some minimal extensions to UML, collectively known as UML*i*, that are specifically targeted at the needs of user interface modellers [8]. The emphasis in UML*i* is on supporting the development of form-based interactive interfaces to applications modelled using the existing facilities of UML. Indeed, most data intensive interactive systems are based on forms, e.g., database applications and web applications [25].

However, fully as important as the identification of appropriate modelling facilities is the development of effective environments in which to develop the models. Several tools have been developed for creating and managing UML models (e.g., [19, 20, 23]), and MB-UIDEs are themselves often associated with interactive model development tools (e.g., [1, 2, 13, 18, 22]). The focus of this paper is on the development of a modelling environment for UML*i*. As UML*i* is an extension of UML, most application development within UML*i* uses the existing facilities of UML. As such, it seems natural to develop a tool for UML*i* as an extension of an existing development environment for UML, and in fact the UML*i* environment is an extension of ARGO [20].

This paper demonstrates that a UML*i*-based tool can provide a design environment:

- where user interfaces and their mainstream applications can be modelled in an integrated way;
- that facilitates the design of activity diagrams that are simultaneously supported by interaction and domain objects.

Therefore, the implementation of the features of UML*i* in a tool can anticipate in the design the need to specify very precisely the relationship between interaction and domain objects. In the absence of such support, this integration is often a costly task performed in the implementation.

This paper has the following structure. An overview of UML*i* is provided in Section 2. Generic aspects of the UML*i* tool are presented in Section 3. Tool support for the modelling of UI presentations is described in Section 4. Tool support for modelling interactive application control-flow and data-flow using activity diagrams is presented in Sections 5 and 6. Tool support for modelling the collaboration between interaction and domain objects is presented in Section 7. Conclusions are presented in Section 8. Throughout the paper, some familiarity with UML notation and terminology is assumed.

2 UMLi Overview

The features of UML*i* are illustrated in this paper through the description of the models used for specifying the `ConnectUI` user interface presented in Figure 1. This user interface is where system users provide their login name and password to gain access to system's services. This is a rudimentary example, but it suffices to illustrates the UML*i* modelling environment. A more substantial UML*i* model is presented in [8].

A UML*i* model describing structural aspects of `ConnectUI` is presented in Section 2.1, and a UML*i* model describing dynamic aspects of `ConnectUI` is given in Section 2.2. Further details of the UML*i* can be obtained in [8] and in the UML*i* home-page at `http://img.cs.man.ac.uk/umli`, from which the modelling environment described in this paper can be downloaded.

2.1 Modelling User Interface Presentations

UML provides class and object diagrams for modelling structural aspects of software. The use of these diagrams for UI design presents some difficulties. For instance, the identification of interaction object *roles* and *containments* may not be easy in object diagrams [7].

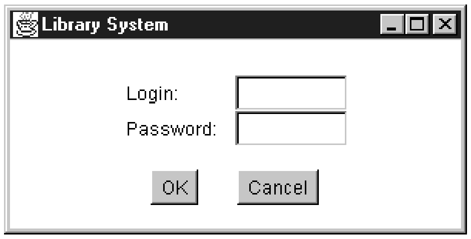


Figure 1. The *ConnectUI* user interface of a library system.

The *user interface diagram* in Figure 2 is an *abstract presentation model* of the `ConnectUI`. The UI diagram constructors are specialised UML classes called *InteractionObjects*. Thus, a UI diagram is a specialisation of a class diagram. The UI diagram in Figure 2 provides examples of most of the *InteractionObjects* specified in UML*i*. An *Inputter*, ∇, is responsible for receiving information from users. A *Displayer*, $\triangle$, is responsible for sending visual information to users. An *Editor*, $\diamond$, is simultaneously an *Inputter* and a *Displayer*. An *ActionInvoker*, ▷, is responsible for invoking an object operation or raising an event. A *Container*, ⌸, is an *InteractionObject* that can contains other *InteractionObjects*. A *FreeContainer*, ⌸, is a top-level *Container* that cannot be contained by any other *Container*. *Inputters*, *Displayers*, *Editors* and *ActionInvokers* are *PrimitiveInteractionObjects*.

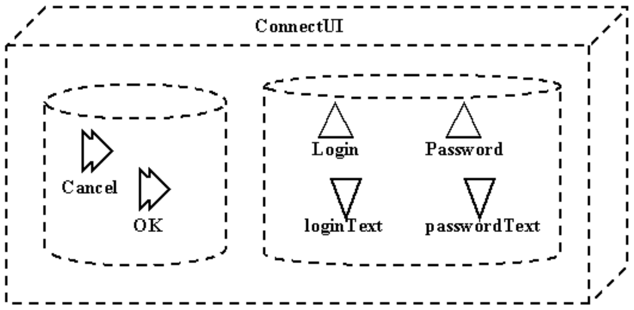


Figure 2. A user interface diagram describing structural aspects of the *ConnectUI*.

The problems relating to the description of groupings and roles using class diagrams are addressed in the UI diagrams: both the role and the grouping of interaction objects is depicted graphically. For example, we can see that the `passwordText` *Inputter* is contained by the `ConnectUI` *FreeContainer* in Figure 2. Further,

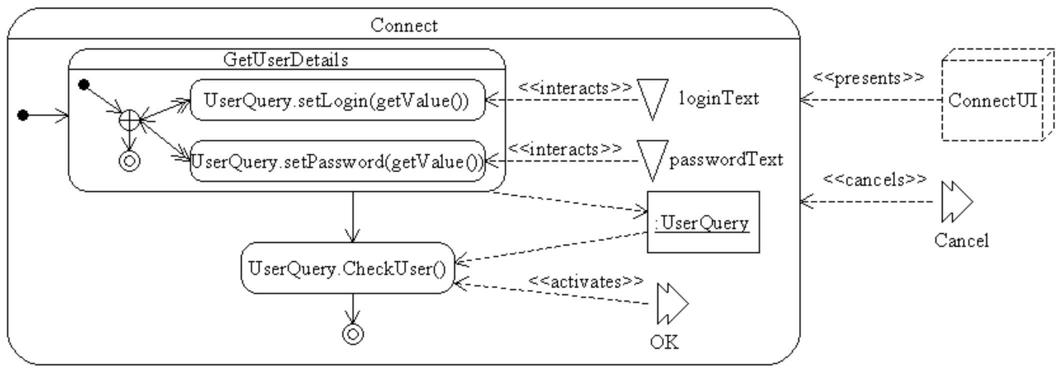


Figure 3. An extended activity diagram describing behavioural aspects of the *ConnectUI*.

`loginText` and `passwordText` play the same abstract role as *Inputters*.

2.2 Modelling User Interface Behaviours

The activity diagram in Figure 3 is a description of the behaviour of the `ConnectUI`. Broadly speaking, activities describe the actions that can be performed using the UI. The transitions describe the possible ways of executing activities. Further, object flows specify how objects are used by actions in activities.

UML*i* provides *stereotyped interaction object flows* to describe behaviours that are often performed in UIs. For example, the ≪*presents*≫ interaction object flow in Figure 3 specifies that the `ConnectUI` *FreeContainer* and its contained interaction objects are presented when the `Connect` *activity* is enabled, and that they are hidden when the `Connect` *activity* is disabled. Further, the ≪*interacts*≫ interaction object flow in Figure 3 specifies that users can interact with the `passwordText` *Inputter* when the `UserQuery.setPassword(getValue())` *ActionState* is enabled. UML*i* specifies three other stereotypes for interaction object flows in addition to the ≪*presents*≫ and ≪*interacts*≫ stereotypes: the ≪*cancels*≫ stereotype for cancelling the execution of an activity, the ≪*confirms*≫ stereotype for confirming the end of optional decisions, and the ≪*activates*≫ stereotype for triggering the execution of *ActionStates*.

The introduction of *SelectionStates* is another contribution of UML*i* for modelling UIs. For instance, the *OrderIndependentState*, $\oplus$, in Figure 3 specifies that its two *selectable states*, in this case, the two *ActionStates* connected to the *OrderIndependentState* by *ReturnTransitions*, $\longleftrightarrow$, must be executed once each, but may be carried out in any order. UML*i* specifies two other categories of *SelectionState* not used in Figure 3: *OptionalState* and *RepeatableState*.

Activity diagrams without object flows can be considered as *control-flow models*. Indeed, a description of the possible transitions between activities models the application workflow. Activity diagrams with object flows also model collaborations between objects. In this case, these activity diagrams can be considered as *data-flow models*. The distinction between control-flow and data-flow models is not normally relevant, since both of them are typically required during a UI design. However, the distinction is emphasised here because the UML*i* tool has distinct facilities for modelling control-flow and data-flow. The facilities of UML*i* for control-flow modelling are discussed in Section 5, and those for data-flow modelling are discussed in Section 6.

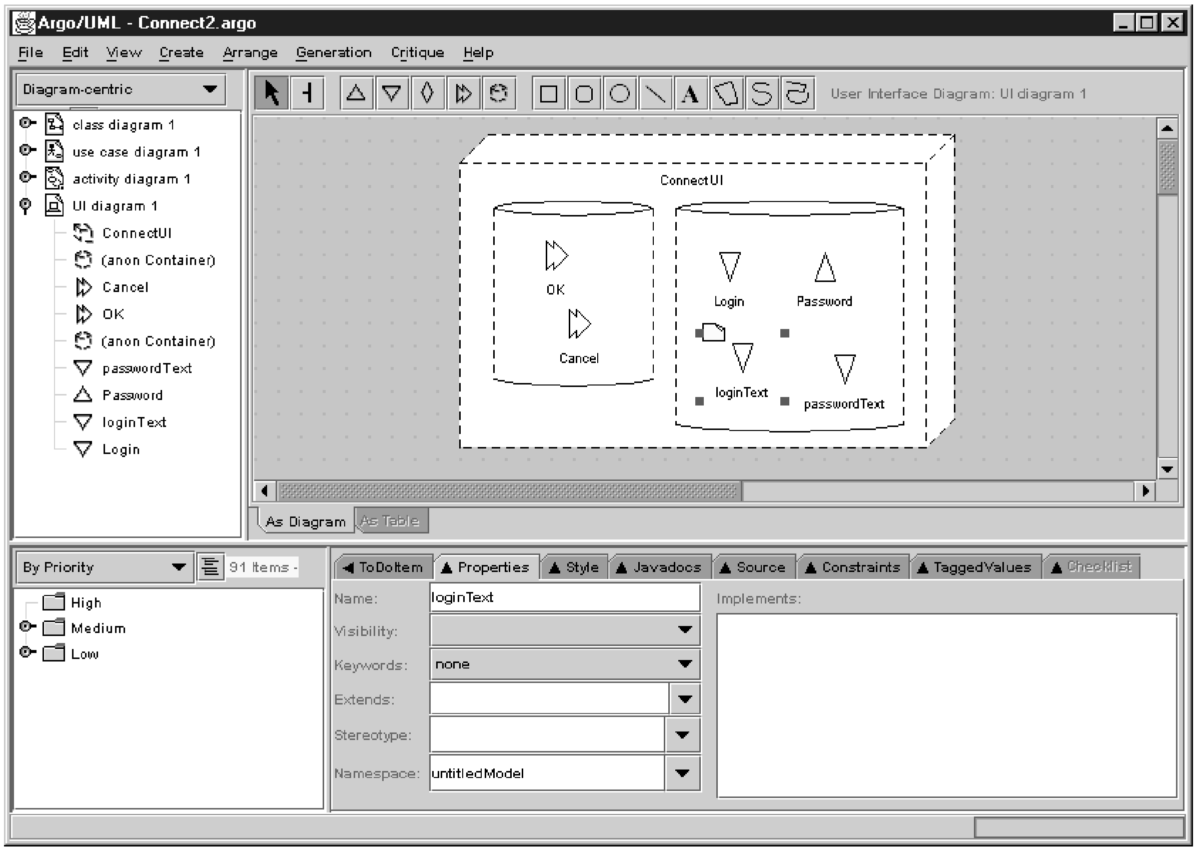
3 Implementing the UML*i* Tool

There are many computer-aided software engineering (CASE) tools for UML, e.g. Rational Rose [19], Together [23] and ARGO[20]. This section explains how the features of UML*i* have been implemented in ARGO [20].

3.1 ARGO Overview

There are two characteristics of ARGO that have guided our decision to build on this specific UML tool:

- ARGO is open source software. Thus, we may have the chance to implement the features of UML*i* without relying on the sometimes limited extension mechanisms occasionally provided by other UML-based tools.
- The ARGO object model used for handling UML models at runtime conforms with the OMG UML 1.3 specification [15].



Name: loginText
Visibility:
Keywords: none
Extends:
Stereotype:
Namespace: untitledModel

Figure 4. A snapshot of the ARGO user interface.

The ARGO user interface is shown in Figure 4, where there are four distinct panels:

- *Editing panel.* This panel is located at the top right of Figure 4, and is where UML*i* diagrams are constructed. This panel is composed of the *working area* and the *selection box*. The selection box is used for selecting a *constructor creator* or an *operator*. Constructor creators are used for adding new components to the working area. Operators are used for modifying constructors already created in the working area. The contents of the selection box, in terms of constructor creators and operators, depends on the kind of diagram that is being edited. For instance, the selection box in Figure 4 contains the constructor creators for UI diagrams, since this is the kind of diagram being edited.

- *Navigation panel.* This panel is located at the top left of Figure 4. From this panel designers can navigate through the entire UML model, switching from one diagram or diagram element to another. For instance, by selecting a new diagram in the navigation panel a designer can set the selected diagram as the current one in the editing panel.

- *Detail panel.* This panel is located at the bottom right of Figure 4. In this panel designers can interact with elements of the UML model and ARGO environment that may not be represented graphically in the editing panel. Different kinds of information can be specified in this panel. A different form is provided for each kind of information that can be specified. These forms are selected using the detail panel tabs, i.e. `ToDoItem`, `Properties` and `Style`, as shown in Figure 4. In the case of UML*i*, we are particularly interested in the *properties form*, where designers can provide additional information for UML*i* constructors. The content of the properties form is based on the selected component of the current diagram, if any. For example, the properties form in Figure 4 shows details about the selected `loginText` *Inputter*. If no component is selected, the property panel displays the property of the current diagram.

- *To Do panel.* This panel is located at the bottom left of Figure 4. Design critics provided by ARGO are presented in this panel. This is a possible place for implementing some constraints specified in the UML*i* model. Indeed, rather than enforcing the construction of consistent UML models from the beginning, ARGO provides non-compulsory criticism facilities that may provide guidance for building consistent models in an incremental way. The version of ARGO that implements the UML*i* extensions, v.0.8.1, does not make use of the ARGO criticism facilities.

From this description of the ARGO user interface we can see that UML*i* models are built in the editing panel. Moreover, detailed information concerning the models is normally provided in the property form. Therefore, UML*i* features are mainly based on extended functionalities implemented in the editing panel and property form.

3.2 Implementing ARGO*i*

The version of ARGO that provides the UML*i* facilities is called ARGO*i*. A high-level description of the ARGO architecture is required to explain the implementation of ARGO*i*. The components of ARGO are presented in Figure 5. There, the packages are composed of Java/Swing classes that may have their own sub-packages. These packages have the following roles in ARGO.

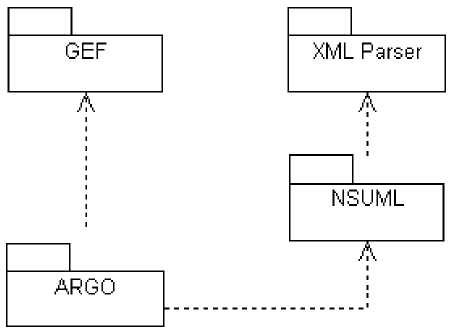


Figure 5. A top-level package view of the ARGO architecture.

- The Graph Editing Framework (GEF) package provides a generic set of graphical constructors for implementing diagrams, nodes and edges.

- The Novosoft UML (NSUML) package implements the UML object model. Additionally, the package provides the facility of saving and loading UML object models in the OMG XML metadata interchange (XMI) format [15].

- An XML parser package is used by NSUML classes for loading object models from XMI files.

- The ARGO package is composed of extend GEF classes. These ARGO classes provide editable graphical representations for the NSUML objects. Moreover, they are the graphical elements used in the editing panel.

The UML*i* implementation has extended classes in the ARGO and the NSUML packages described above. The ARGO package has been extended to provide the following facilities for modelling UML*i* diagrams.

- Editing facilities for user interface diagrams, as discussed in Section 4;

- Editing facilities in activity diagrams for modelling selection states, initial interaction states and interaction object flows, as in Sections 5 and 6;

- Wizards in activity diagrams for designing control-flow, as discussed in Section 5;

- Wizards in activity diagrams for modelling interaction object flows, as discussed in Section 6.

The NSUML package has been extended to support the UML*i* metamodel [5]. The NSUML package has also been extended to support the generation and reading of UML*i* XMI document type definition (DTD) conformant files [15]. This part of the implementation benefits significantly from the principle adopted in UML*i* of extending, but not modifying, the UML specification. Every XML file that conforms with the UML XMI DTD also conforms with the UML*i* XMI DTD.

At this point, we can recall that one of the difficulties of using existing MB-UIDEs is that each development environment uses a different set of notations for modelling UIs [6]. This means that UI models built by one MB-UIDE cannot be used by another one due to the intrinsic difficulty of translating models built over different notations. For instance, MASTERMIND [22], JANUS [1], Teallach [2] and MOBI-D [18] cannot interchange their models, even partially. With the use of XMI files, UML-based tools can interchange models. It is even expected that UML-based tools can eventually share models in a collaborative and distributed development environment. In terms of UML*i*, this model interchange ability means that UML*i*-based tools can use, without any translation, partial UML models of interactive applications, even when these have been developed using another environment.

4 Presentation Modelling Support

A snapshot of the ARGOi user interface when modelling a UI diagram is presented in Figure 4, which shows the modelling of the `ConnectUI` diagram from Figure 2. The implementation techniques used in this UMLi-specific diagram are the same as for other UML diagrams. For instance, the same technique is used for implementing interaction object containment in UI diagrams and state containment in statechart diagrams. In this section, we present the specifics of UI diagram editors, rather than generic implementation details of ARGO.

The decision about the content of each diagram is one of the first problems facing the implementor of a UI diagram editor. In fact, the *diagram* concept is not explicitly specified in UML. For instance, the classes of an application can be modelled in a single class diagram or in several class diagrams. In ARGOi, a UI diagram has exactly one *FreeContainer*. Indeed, a *FreeContainer* is automatically created when its UI diagram is created, and a UI diagram is deleted when its *FreeContainer* is deleted. For this reason, there is no *FreeContainer* creator in the selection box of the UI diagram editor, as we can see in Figure 4.

The decision that a UI diagram should contain exactly one *FreeContainer* may facilitate the selection of a *FreeContainer* in large-scale models. Indeed, navigation panels in UML-tools are usually organised around diagrams. Thus, *FreeContainers* can be selected through the selection of their UI diagrams in navigation panels. Otherwise, a search facility would be required to locate the UI diagram of a specific *FreeContainer*.

Interaction objects that are not *FreeContainers* are added to a UI diagram using one of the constructor creators in the selection box. Interaction object containment is initially specified by the position of the cursor on the working area of the editing panel when the pointer-device button is pressed. Thus, interaction objects are added into the innermost *Container* related to the selected position. Designers can modify interaction object containment by dragging and dropping interaction objects.

Interaction object placement (in contrast with containment) is not relevant in UI diagrams, since layout is normally more a concrete presentation concern than an abstract presentation one [6]. Therefore, the UI diagrams need to be refined into concrete presentations. The generation of concrete presentations from an abstract presentation model is partially described in [7]. The problem of how best to map abstract interaction objects onto concrete interaction objects, however, is still a research problem [24].

5 Control-flow Modelling Support

Activity diagram elements can be added to diagrams using the selection box. Alternatively, activity diagram elements can be added using the *temporal-relation wizard* presented in this section.

This wizard is based on task model techniques that exploit extensions to UML activity diagrams for modelling control-flow. In fact, task modelling is a well established technique for modelling the behaviour of interactive applications [10, 16]. Designers can build a task hierarchy that models the control-flow of the application by decomposing tasks into subtasks and specifying temporal relations between the subtasks. In UMLi, application control-flow can be modelled by activity diagrams. However, activity diagrams tend to be less abstract than task models. In particular, inter-object transitions in activity diagrams tend to be more complex to model than temporal relations in task models. In fact, the difficulty of modelling inter-object transitions using the statechart constructors was anticipated by Harel and Gery [9] when statecharts were adopted by UML. The temporal-relation wizard in ARGOi provides at least two benefits for modelling activity diagrams.

1. It can reduce the effort of modelling control-flow using UML. The selection of one of the wizard's options creates a complete set of constructors required for modelling the behaviour of temporal relations in a task model.

2. It exploits the potential of *SelectionStates* and *ReturnTransitions* (as introduced in Section 2.2) for modelling abstract inter-object transitions, simplifying the control-flow modelling process in UML-based tools. Indeed, a facility for modelling *OrderIndependentStates*, *OptionalStates* and *RepeatableStates* makes the control-flow modelling process more similar to the task modelling process in UI-specific development environments such as CTTE [16], MOBI-D [18] and Teallach [2].

The temporal-relation wizard appears every time a node in an activity diagram is selected, such as for the state `S` in Figure 6(b). The wizard is the iconographic menu to the right of `S`. We can see the same state `S` in Figure 6(a) before it was selected. The other wizards in Figure 6(b) are standard activity diagram wizards of

ARGO. The temporal-relation wizard has six options that perform the following actions:

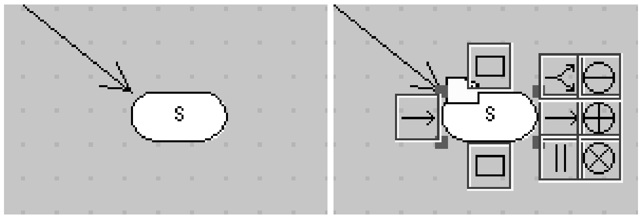


Figure 6. (a) The unselected S state. (b) The selected S state along with the ARGO*i* temporal-relation wizard on its right.

- *Sequential option* (→): This option builds an activity connected to the current node by a transition. This action is represented graphically in Figure 7(a).
- *Concurrent option* (||): This option builds two activities, a join, and a fork. A transition connects the current node to the fork. Each activity built has two transitions: one coming from the fork, and one going to the join. This action is represented graphically in Figure 7(b).
- *Choice option* ($\prec$): This option builds two activities and a branch. A transition connects the current node to the branch. Each activity built has a guarded transition coming from the branch. The transitions are built guarded to remind designers that these guards may be required. This action is represented graphically in Figure 7(c).
- *OrderIndependent option* (⊕): This option builds one *OrderIndependentState* and two activities. One transition connects the current node to the *OrderIndependentState*. Each activity built is connected to the *OrderIndependentState* by a *ReturnTransition*. This action is represented graphically in Figure 7(d).
- *Optional option* (⊖): This option builds one *OptionalState* and two activities. One transition connects the current node to the *OptionalState*. Each activity built is connected to the *OptionalState* by a *ReturnTransition*. This action is represented graphically in Figure 7(e).
- *Repeatable option* (⊗): This option builds one *RepeatableState* and one activity. One *ReturnTransition* connects the current node to the *RepeatableState*. A *ReturnTransition* connects the *RepeatableState* to the new activity. This action is represented graphically in Figure 7(f).

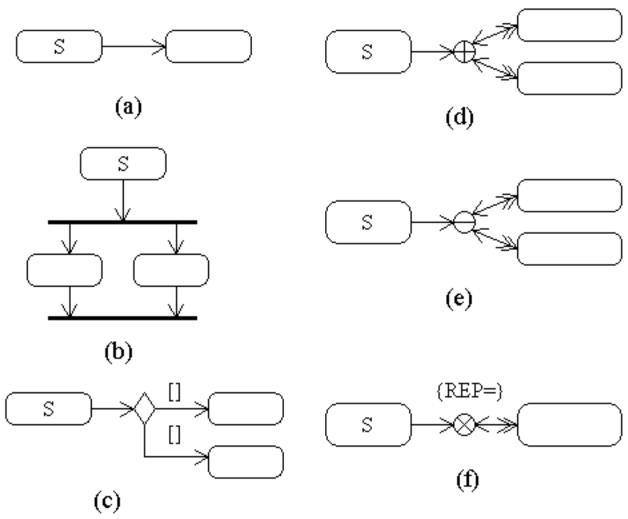


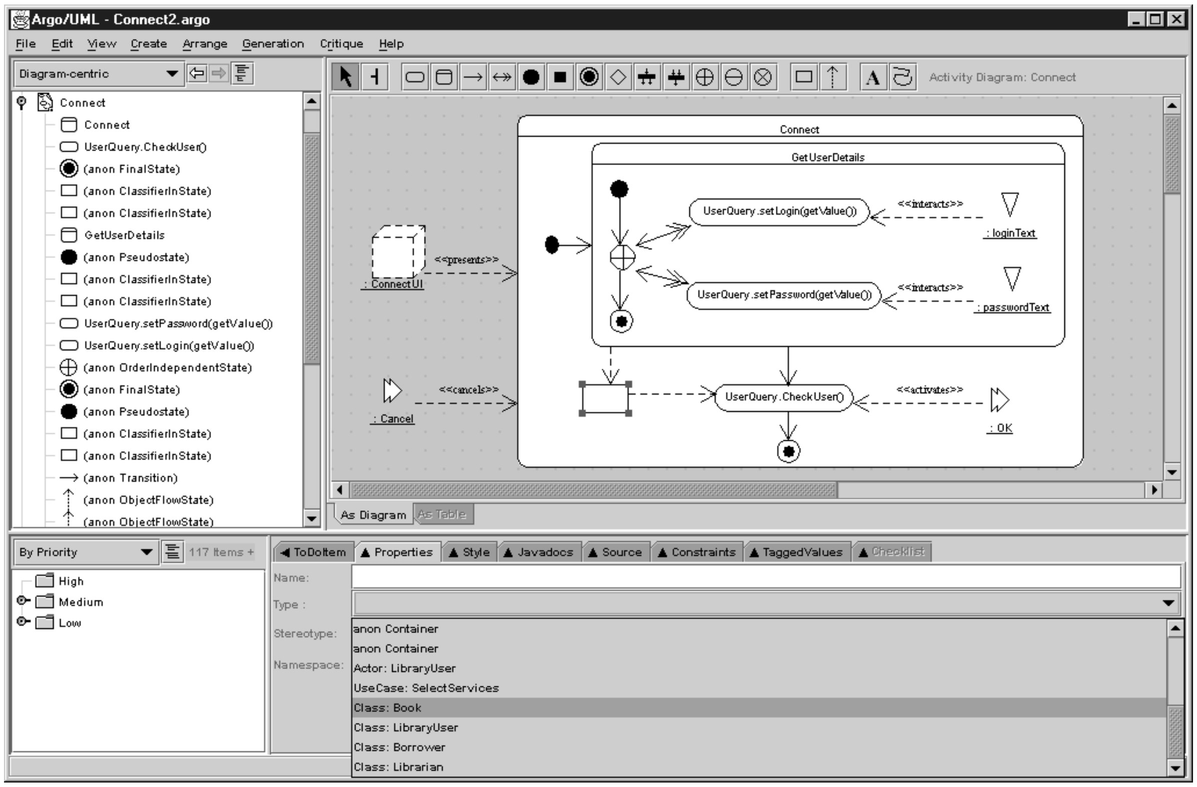
Figure 7. The result of each of the six actions that can be performed by the temporal relation wizard. The S state is that from which the wizard is invoked.

Figure 8 shows an snapshot of ARGO*i* during the modelling of the activity diagram in Figure 3. The *OrderIndependentState* in Figure 8 was built using the temporal-relation wizard. In fact, the *InitialState* of `GetUserDetails` corresponts to the `S` *State* in Figure 7(d), and the `UserQuery.setLogin(getValue())` and `UserQuery.setPassword(getValue())` *ActionStates* are refinements of the two selectable states created along with the *OrderIndependentState*.

In terms of control-flow modelling, UML*i* still specifies the *InitialInteractionState*. This constructor is a pseudo-state responsible for specifying the entry-point of interactive applications [8, 5]. ARGO*i* provides an *InitialInteractionState* creator in the selection box when editing an activity diagram.

6 Data-flow Modelling Support

Object flows provide the ability to specify data-flows in activity diagrams. For non-*InteractionObjects*, UML*i* specifies that object flows can be connected to *CompositeStates* in addition to *ActionStates*. For *InteractionObjects*, UML*i* specifies *interaction object flows*



Connect
Connect
UserQuery.CheckUser()
(anon FinalState)
(anon ClassifierInState)
(anon ClassifierInState)
GetUserDetails
(anon Pseudostate)
(anon ClassifierInState)
(anon ClassifierInState)
UserQuery.setPassword(getValue())
UserQuery.setLogin(getValue())
(anon OrderIndependentState)
(anon FinalState)
(anon Pseudostate)
(anon ClassifierInState)
(anon ClassifierInState)
(anon Transition)
(anon ObjectFlowState)
(anon ObjectFlowState)

anon Container
anon Container
Actor: LibraryUser
UseCase: SelectServices
Class: Book
Class: LibraryUser
Class: Borrower
Class: Librarian

Figure 8. The modelling of the activity diagram in Figure 3.

that: (1) can have an *interaction stereotype*; and (2) can be connected to *CompositeStates*, *SelectionStates* and *ActionStates*.

The modelling of object flows and interaction object flows may not be a simple task. Selecting a *Classifier*, i.e., a *Class*, *UseCase* or *InteractionObject*, to play the *type* role in an interaction object flow can be complex due to the number of *Classifiers* usually available in interactive application designs. Further, selecting a proper stereotype for an interactive object flow may be complex due to the rich semantics of the UML*i* interaction stereotypes. For instance, the ≪*presents*≫ interaction stereotype specifies a *FreeContainer context* [5].

ARGO*i* provides facilities to cope with the complexity of selecting types and stereotypes for interaction object flows. A precise description of these facilities, however, is intentionally avoided here since it requires a partial description of the UML*i* metamodel [5]. However, examples of these facilities can be provided using the activity diagram in Figure 3.

6.1 Selecting Types for Interaction Object Flows

Classifiers specified in the current UML*i* models are provided as options in the combo box of interaction object flows. For instance, Figure 8 shows how the combo box in the properties form can be used to specify the *type* of the object flow that is being added to the `Connect` *Activity*. There, the options in the combo box, e.g., `Book` *Class* and `SelectServices` *UseCase*, are *Classifiers* already specified in the models of the library system. Considering this type specification approach, ARGO*i* can analyse the current state of the UML*i* model to filter those *Classifiers* that cannot be a type for the selected interaction object flow. Two examples of the UML*i* model aspects analysed by ARGO*i* for filtering *Classifiers* are the following:

- *FreeContainer context.* In Figure 3 we can see the use of the `ConnectUI` *FreeContainer* as a ≪*presents*≫ interaction object flow of the `Connect` *activity*. This means that only the following interaction objects in the models can be made available for selection as a type of interaction object flows used by activities within the `Connect` activity:

  1. The *PrimitiveInteractionObjects* contained by `ConnectUI` (see Figure 2);

2. The *FreeContainers*. Indeed, this provides the ability to create new *FreeContainer* contexts.

- *ActionInvoker roles.* An instance of a *PrimitiveInteractionObject* should not play more than one role in a *FreeContainer*, e.g., ▷ `Cancel` in Figure 2 should not be associated with any other subactivity of the `Connect` *activity* in Figure 3 since it is already responsible for the cancelling behaviour within the `Connect` *activity*. Thus, ARGOi can notify designers when it identifies a *PrimitiveInteractionObject* playing more than one role.

6.2 Selecting Stereotypes for Interaction Object Flows

The selection of stereotypes for interaction object flows can be performed in a *property form*, like the selection of interaction object flow types. Once again, ARGOi can analyse the current state of the models in order to filter *interaction stereotypes* that do not suit the selected interaction object flow. An example of the UMLi model aspects analysed by ARGOi for filtering *interaction stereotypes* is presented as follows:

- *Associated state context.* If the state associated with the selected interaction object flow is a *CompositeState*, the interaction stereotype must be ≪*presents*≫, which creates a *FreeContainer context*, or ≪*cancels*≫. If the associated state is a *SelectionState*, the interaction stereotype must be ≪*confirms*≫, which allows users to indicate the finishing of an optional selection, or ≪*cancels*≫. If the state is an *ActionState*, the interaction stereotype must be ≪*interacts*≫, which enables the associated interaction object, or ≪*activates*≫, which makes the associated interaction object a trigger of the *ActionState*.

7 Interaction and Domain Object Collaboration Support

Interaction and domain objects collaborate in UMLi models when they are used by object flows sharing common *ActionStates*. For example, an instance of the `loginText` *Inputter* collaborates with an instance of the `UserQuery` *class* in the library system since they share the `UserQuery.setLogin(getValue())` *ActionState* in Figure 3. Indeed, UMLi makes explicit such collaboration since it provides a clear distinction between interaction and non-interaction objects. Moreover, UMLi makes explicit the problem of creating and preserving the integration between interaction and domain objects in the designs of interactive applications.

The problem of creating this integration can be partially addressed through the checking of UMLi models, a topic that is outside the scope of this paper. The problem of preserving this integration can be minimised through the use of the *integration wizard*. This wizard is triggered every time a designer deletes any class or interaction object. Thus, the wizard can check and notify the designer about *ActionStates* affected by such class or interaction object removal. Indeed, designers can have different motivations for modifying class diagrams and UI diagrams. However, they may not be able to evaluate the impact of modifying one diagram in other diagrams.

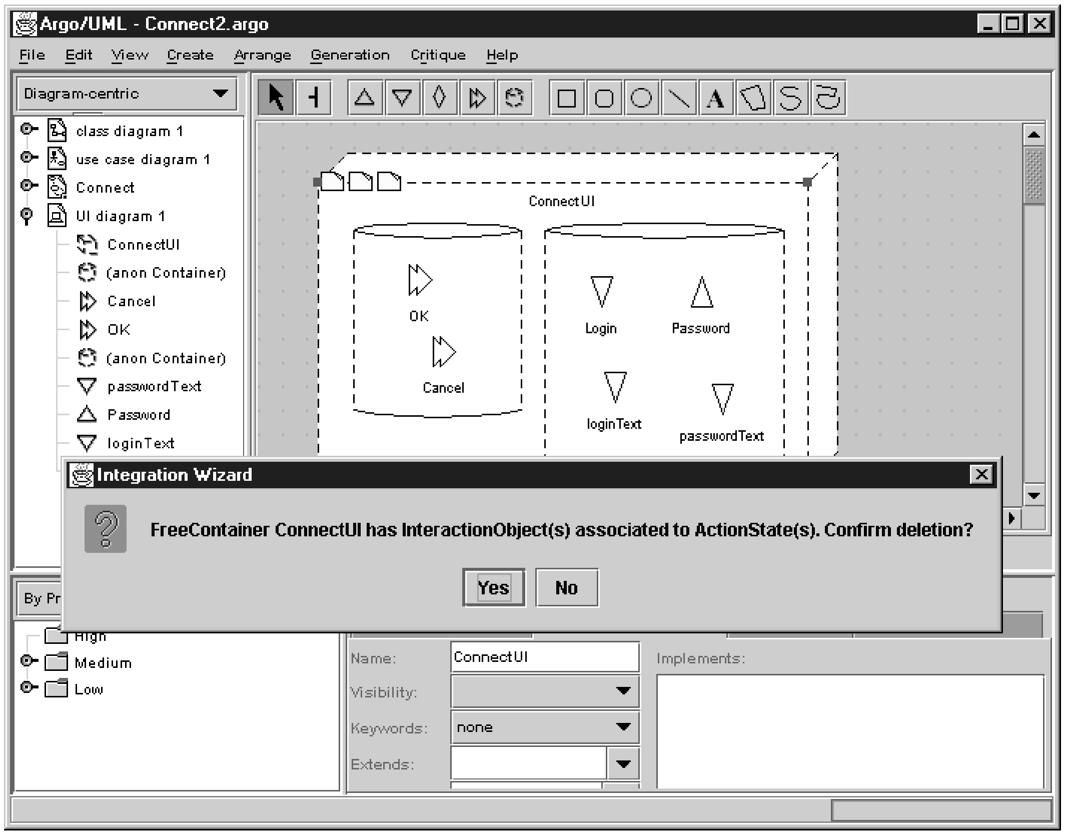
Considering this problem of preserving the collaboration of the interaction and domain objects, the *integration wizard* notifies designers about the effects of class and interaction object removal in the following design scenarios:

- *Deleting domain classes.* The *integration wizard* checks in the UMLi models to see if a domain class that is to be deleted is a type of an object flow that shares at least one common state with an *InteractionObject*.

- *Deleting interaction objects.* The *integration wizard* checks in the UMLi models to see if a *PrimitiveInteractionObject* to be deleted is a type of an object flow that shares at least one common state with a *Class*.

- *Deleting FreeContainers and Containers.* The *integration wizard* checks in the UMLi models to see if a *Container* or a *FreeContainer* to be deleted contains *PrimitiveInteractionObjects* that share *ActionStates* with domain classes, as described in *deleting interaction objects* above. The deletion of a *Container* or *FreeContainer* in ARGOi implies the recursive deletion of its contained *InteractionObjects*. Thus, the *integration wizard* also verifies recursively the side-effects of such a deletion in terms of interaction and domain object integration.

Figure 9 shows the *integration wizard* when deleting the `ConnectUI` *FreeContainer*. In this case, the *integration wizard* is activated since the `loginText`, `passwordText` and `OK` *InteractionObjects* of the `ConnectUI` *FreeContainer* are associated to *ActionStates*, as modelled in the activity diagram in Figure 8.



FreeContainer ConnectUI has InteractionObject(s) associated to ActionState(s). Confirm deletion?

Name: ConnectUI
Visibility:
Keywords: none
Extends:

Figure 9. An attempt to delete the `ConnectUI` FreeContainer can trigger the integration wizard.

8 Conclusions

This paper shows that the UML*i* specification can be effectively implemented in a UML-based design environment. Moreover, this paper shows that ARGO*i*, a UML*i*-based tool, can provide support for modelling a complete interactive application exploiting the UML*i* specification.

Compared with other UML tools (e.g., Rational Rose [19], Together [23], ARGO [20]), ARGO*i* provides additional tool support for:

- modelling abstract user interface presentations using the UML*i* user interface diagram (Section 4);
- modelling common UI behaviours using the *temporal-relation wizard* that exploits the use of the UML*i* *SelectionStates* (Section 5);
- modelling in an explicit and effective way the collaboration between interaction and domain objects (Section 6);
- preserving, when modelled, the integration between interaction and domain objects using the *integration wizard* (Section 7).

Compared with most MB-UIDEs, ARGO*i* can provide the following distinctive benefits:

- A graphical notation for modelling inter-model relationships, such as the ≪*presents*≫ object flow shown in Figure 3, which explicitly links the presentation model of a UI with the activity diagram modelling the UI's behaviour. Relationships between control-flow models (e.g., task models, activity diagrams) and structural models (e.g., class diagrams, UI diagrams) are modelled in a non-graphical way in Teallach [2] and MOBI-D [18].
- It allows the construction of domain models along with the construction of UI models.

There are MB-UIDEs that integrate a UI design environment with a mainstream CASE tool. For instance, JANUS [1] uses Together [23] for building its models, and AME [13] uses the OODevelopTool for building its models. However, ARGO*i* models can provide more comprehensive specification of UIs than

AME and JANUS models. Indeed, ARGO*i* provides a generic approach for modelling and relating structural and dynamic aspects of interactive applications (Sections 5 and 6). The AME and JANUS approaches for modelling UIs are quite limited for describing the behavioural aspects of UIs. Indeed, these approaches are based on the identification of the operations that can be executed by each interaction object rather than modelling the application workflow in an abstract way.

In the context of user interfaces to data intensive applications, UML*i* and ARGO*i* provide the following benefits:

- integration of UI design with data modelling through class diagrams;
- comprehensive support for the modelling of data flow during activities;
- support for form-based interfaces, which are predominant for most database interfaces;
- backward compatibility with existing UML models, in which many data intensive applications are already designed.

UML*i* has been developed as part of an ongoing project. The next activities of the project are:

- The modelling of a large-scale interactive application using ARGO*i*;
- The implementation of a UI software generator in ARGO*i*.

Acknowledgements. The first author is sponsored by Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Brazil) – Grant 200153/98-6.

References

[1] H. Balzert, F. Hofmann, V. Kruschinski, and C. Niemann. The JANUS Application Development Environment — Generating More than the User Interface. In *Computer-Aided Design of User Interfaces*, pages 183–206, Namur, Belgium, 1996. Namur University Press.

[2] P. Barclay, T. Griffiths, J. McKirdy, N. Paton, R. Cooper, and J. Kennedy. The Teallach Tool: Using Models for Flexible User Interface Design. In *Proceedings of CADUI'99*, pages 139–157, Louvain-la-Neuve, Belgium, October 1999. Kluwer.

[3] D. Benyon, T. Green, and D. Bental. *Conceptual Modelling for Human Computer Interaction using ERMIA*. Springer, London, UK, 1999.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[5] P. Pinheiro da Silva. On the Semantics of the Unified Modeling Language for Interactive Applications. In preparation.

[6] P. Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In *Proceedings of DSV-IS2000*, LNCS, pages 207–226, Limerick, Ireland, June 2000. Springer-Verlag.

[7] P. Pinheiro da Silva and N. Paton. User Interface Modelling with UML. In *Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Representation*, Saariselkä, Finland, May 2000. IOS Press. (To appear).

[8] P.Pinheiro da Silva and N. Paton. UMLi: The Unified Modeling Language for Interactive Applications. In *Proceedings of UML2000*, volume 1939 of *LNCS*, pages 117–132, York, UK, October 2000. Springer.

[9] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[10] P. Johnson. *Human Computer Interaction: Psychology, Task Analysis and Software Engineering.* McGraw-Hill, Maidenhead, UK, 1992.

[11] A. Kilgour. Theory and practice in user interface management systems. *Information and Software Technology*, 29:171–175, 1987.

[12] S. Kovacevic. UML and User Interface Modeling. In *Proceedings of UML'98*, pages 235–244, Mulhouse, France, June 1998. ESSAIM.

[13] C. Märtin. Software Life Cycle Automation for Interactive Applications: The AME Design Environment. In *Computer-Aided Design of User Interfaces*, pages 57–74, Namur, Belgium, 1996. Namur University Press.

[14] B. Myers. User Interface Software Tools. *ACM Trans. Computer-Human Interaction*, 2(1):64–103, March 1995.

[15] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. Version 1.3.

[16] F. Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer, Berlin, 1999.

[17] F. Paternò. Towards a UML for Interactive Systems. In *Proceedings of EHCI2001*, LNCS, Toronto, Canada, May 2001. Springer. (To appear).

[18] A. Puerta. A model-based interface development environment. *IEEE Software*, August:40–47, 1997.

[19] T. Quatrani. *Visual Modeling with Rational Rose and UML.* Addison-Wesley, 1998.

[20] J. Robbins, D. Hilbert, and D. Redmiles. ARGO: A Design Environment for Evolving Software Architectures. In *Proceedings of ICSE'97*, pages 600–601, Boston, MA, May 1997. ACM Press.

[21] P. Szekely. Retrospective and Challenges for Model-Based Interface Development. In *Computer-Aided Design of User Interfaces*, pages xxi–xliv, Namur, Belgium, 1996. Namur University Press.

[22] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative Interface Models for User Interface Construction Tools: the MASTERMIND Approach. In *Engineering for Human-Computer Interaction*, pages 120–150, London, UK, 1996. Chapman & Hall.

[23] TogetherSoft. http://www.togethersoft.com.

[24] J. Vanderdonckt. Advice-giving systems for selecting interaction objects. In *User Interfaces to Data Intensive Systems (UIDIS)*, pages 152–157, Edinburgh, United Kingdom, 1999. IEEE Computer Society.

[25] M. Zloof. Selected Ingedients in End-User Programming. In *Proceedings of the Working Conference on Advance Visual Interfaces (AVI'98)*, pages 30–35, L'Aquila, Italy, May 1998. ACM Press.