UMLi: The Unified Modeling Language 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

User interfaces (UIs) are essential components of most software systems, and significantly affect the effectiveness of installed applications. In addition, UIs often represent a significant proportion of the code delivered by a development activity. However, despite this, there are no modelling languages and tools that support contract elaboration between UI developers and application developers. The Unified Modeling Language (UML) has been widely accepted by application developers, but not so much by UI designers. For this reason, this paper introduces the notation of the Unified Modelling Language for Interactive Applications (UMLi), that extends UML, to provide greater support for UI design. UI elements elicited in use cases and their scenarios can be used during the design of activities and UI presentations. A diagram notation for modelling user interface presentations is introduced. Activity diagram notation is extended to describe collaboration between interaction and domain objects. Further, a case study using UMLi notation and method is presented.

1 Introduction

UML [9] is the industry standard language for object-oriented software design. There are many examples of industrial and academic projects demonstrating the effectiveness of UML for software design. However, most of these successful projects are silent in terms of UI design. Although the projects may even describe some architectural aspects of UI design, they tend to omit important aspects of interface design that are better supported in specialist interface design environments [8]. Despite the difficulty of modelling UIs using UML, it is becoming apparent that domain (application) modelling and UI modelling may occur simultaneously. For instance, tasks and domain objects are interdependent and may be modelled simultaneously since they need to support each other [10]. However, task modelling is one of the aspects that should be considered during UI design [6]. Further, tasks and interaction objects (widgets) are interdependent as well. Therefore, considering the difficulty of designing user interfaces and domain objects simultaneously, we believe that UML should be improved in order to provide greater support for UI design [3, 7].

This paper introduces the UML*i* notation which aims to be a minimal extension of the UML notation used for the integrated design of applications an their user interfaces. Further, UML*i* aims to preserve the semantics of existing UML constructors since its notation is built using new constructors and UML extension mechanisms. This non-intrusive approach of UML*i* can be verified in [2], which describes how the UML*i* notation introduced in this paper is designed in the UML meta-model.

UML*i* notation has been influenced by model-based user interface development environment (MB-UIDE) technology [11]. In fact, MB-UIDEs provide a context within which declarative models can be constructed and related, as part of the user interface design process. Thus, we believe that the MB-UIDE technology offers many insights into the abstract description of user interfaces that can be adapted for use with the UML technology. For instance, MB-UIDE technology provides techniques for specifying static and dynamic aspects of user interfaces using declarative models. Moreover, as these declarative models can be partially mapped into UML models [3], it is possible to identify which UI aspects are not covered by UML models.

The scope of UML*i* is restricted to form-based user interfaces. However, form-based UIs are widely used for data-intensive applications such as database system applications and Web applications and UML*i* can be considered as a baseline for non-form-based UI modelling. In this case, modifications might be required in UML*i* for specifying a wider range of UI presentations and tasks.

To introduce the UML*i* notation, this paper is structured as follows. MB-UIDE's declarative user interface models are presented in terms of UML*i* diagrams in Section 2. Presentation modelling is introduced in Section 3. Activity modelling that integrates use case, presentation and domain models is presented in Section 4. The UML*i* method is introduced in Section 5 when a case study exemplifying the use of the UML*i* notation is presented along with the description of the method. Conclusions are presented in Section 6.

2 Declarative User Interface Models

A modelling notation that supports collaboration between UI developers and application developers should be able to describe the UI and the application at the same time. From the UI developer's point of view, a modelling notation should be able to accommodate the description of users requirements at appropriate levels of abstraction. Thus, such a notation should be able to describe abstract task specifications that users can perform in the application in order to achieve some goals. Therefore, a *user requirement model* is required to describe these abstract tasks. Further, UI sketches drawn by users and UI developers can help in the elicitation of additional user requirements. Therefore, an *abstract presentation model* that can present early design ideas is required to describe these UI sketches. Later in the design process, UI developers could also refine abstract presentation models into *concrete presentation models*, where widgets are selected and customised, and their placement (layout) is decided.

From the application developer's point of view, a modelling notation that integrates UI and application design should support the modelling of application objects and actions in an integrated way. In fact, the identification of how user and application actions relate to a well-structured set of tasks, and how this set of tasks can support and be supported by the application objects is a challenging activity for application designers. Therefore, a *task model* is required to describe this well-structured set of tasks. The task model is not entirely distinct from the user requirement model. Indeed, the task model can be considered as a more structured and detailed view of the user requirement model.

The application objects, or at least their interfaces, are relevant for UI design. In fact, these interfaces are the connection points between the UI and the underlying application. Therefore, the application object interfaces compose an *application model*. In an integrated UI and application development environment, an application model is naturally produced as a result of the application design.

UML*i* aims to show that using a specific set of UML constructors and diagrams, as presented in Figure 1, it is possible to build declarative UI models. Moreover, results of previous MB-UIDE projects can provide experience as to how the declarative UI models should be inter-related and how these models can be used to provide a declarative description of user interfaces. For instance, the links (a) and (c) in Figure 1 can be explained in terms of state objects, as presented in Teallach [5]. The link (d) can be supported by techniques from TRIDENT [1] to generate concrete presentations. In terms of MB-UIDE technology there is not a common sense of the models that might be used for describing a UI. UML*i* does not aim to present a new user interface modelling proposal, but to reuse some of the models and techniques proposed for use in MB-UIDEs in the context of UML.

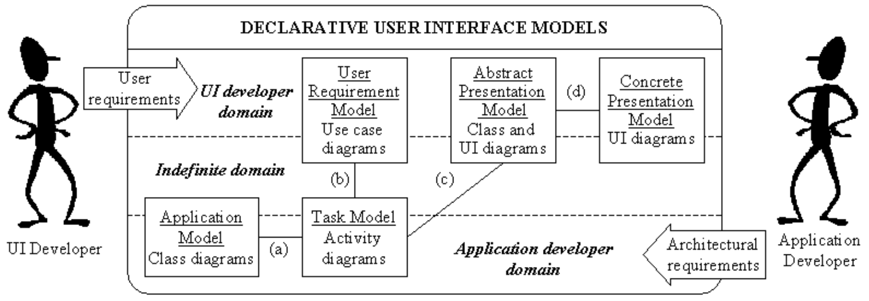

Figure 1: UML*i* declarative user interface models.

3 User Interface Diagram

User interface *presentations*, the visual part of user interfaces, can be modelled using object diagrams composed of *interaction objects*, as shown in Figure 2(a). These interaction objects are also called *widgets* or *visual components*. The selection and grouping of interaction objects are essential tasks for modelling UI presentations. However, it is usually difficult to perform these tasks due to the large number of interaction objects with different functionalities provided by graphical environments. In a UML-based environment, the selection and grouping of interaction objects tends to be even more complex than in UI design environments because UML does not provide graphical distinction between domain and interaction objects. Further, UML treats interaction objects in the same way as any other objects [3]. For instance, in Figure 2(a) it is not easy to see that the *Results* Displayer is contained by the *SearchBookUI* FreeContainer. Considering these presentation modelling difficulties, this section introduces the UML*i* *user interface diagram*, a specialised object diagram used for the conceptual modelling of user interface presentation.

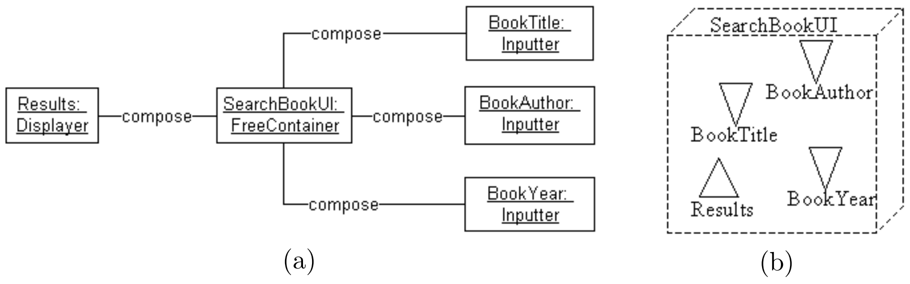

Figure 2: An abstract presentation model for the SearchBookUI can be modelled as an object diagram of UML, as presented in (a). The same presentation can alternatively be modelled using the UML*i* user interface diagram, as presented in (b).

3.1 User Interface Diagram Notation

The *SearchBookUI* abstract presentation modelled using the user interface diagram is presented in Figure 2(b). The user interface diagram is composed of six constructors that specify the role of each interaction object in a UI presentation.

- `FreeContainers`, , are rendered as dashed cubes. They are top-level interaction objects that cannot be contained by any other interaction object, e.g. top-level windows. They are also called *presentation units* since the interaction objects in a FreeContainer are always presented at the same time. An interaction object can be visible and disabled, which means that the user can see the object but cannot interact with it.

- `Containers`, , are rendered as dashed cylinders. They can group interaction objects that are not FreeContainers. Containers provide a grouping mechanism for the designing of UI presentations.

- `Inputters`, ∇, are rendered as downward triangles. They are responsible for receiving information from users.

- `Displayers`, $\triangle$, are rendered as upward triangles. They are responsible for sending visual information to users.

- `Editors`, $\diamond$, are rendered as diamonds. They are interaction objects that are simultaneously Inputters and Displayers.

- `ActionInvokers`, , are rendered as a pair of semi-overlapped triangles pointing to the right. They are responsible for receiving information from users in the form of events.

Graphically, Containers, Inputters, Displayers, Editors and ActionInvokers must be placed into a FreeContainer. Additionally, the overlapping of the borders of interaction objects is not allowed. In this case, the "internal" lines of Containers and FreeContainers, in terms of their two-dimensional representations, are ignored.

3.2 From an Abstract to a Concrete Presentation

The complexity of user interface presentation modelling can be reduced by working with a restricted set of abstract interaction objects, as specified by the user interface diagram notation. However, a presentation modelling approach as proposed by the UML*i* user interface diagram is possible since form-based presentations respect the *Abstract Presentation Pattern*[1] (APP) in Figure 3. Thus, a user interface presentation can be described as an interaction object acting as a FreeContainer. The APP also shows the relationships between the abstract interaction objects.

As we can see, the APP is environment-independent. In fact, a UI presentation described using the user interface diagram can be implemented by any object-oriented programming language, using several toolkits. Widgets should be bound to the APP in order to generate a concrete presentation model. In this way, each widget should be classified as a `FreeContainer`, `Container`, `Inputter`, `Displayer`, `Editor` or `ActionInvoker`. The binding of widgets to the APP can be described using UML [3].

Widget binding is not efficient to yield a final user interface implementation. In fact, UML*i* is used for UI modelling and not for implementation. However, we believe that by integrating UI builders with UML*i*-based CASE tools we can

[1]The specialised constructors under the Inputter, Displayer, Editor and ActionInvoker classes in Figure 3 indicate that many concrete interaction objects (widgets) can be bound to each one of these classes. This constructor is an adaptation of a similar one used in Gamma *et al.* [4] (see page 233).

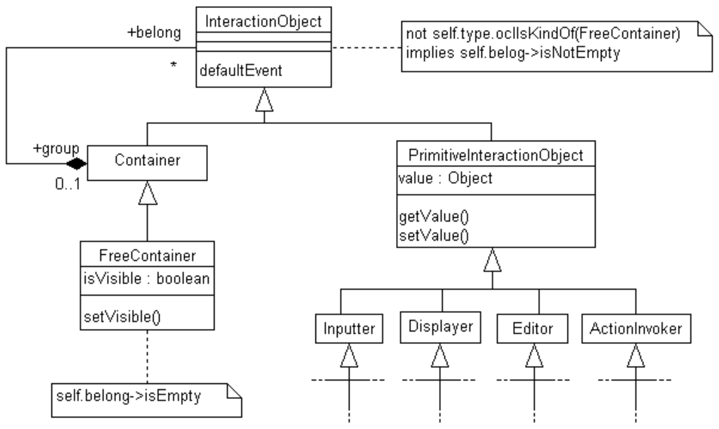

Figure 3: The Abstract Presentation Pattern

produce environments where UIs can be modelled and developed in a systematic way. For instance, UI builder facilities may be required for adjusting UI presentation layout and interaction object's colour, size and font.

4 Activity Diagram Modelling

UML interaction diagrams (sequence and collaboration diagrams) are used for modelling how objects collaborate. Interaction diagrams, however, are limited in terms of workflow modelling since they are inherently sequential. Therefore, concurrent and repeatable workflows, and especially those workflows affected by users decisions, are difficult to model and interpret from interaction diagrams.

Workflows are easily modelled and interpreted using activity diagrams. In fact, Statechart constructors provide a graphical representation for concurrent and branching workflows. However, it is not so natural to model object collaboration in activity diagrams. Improving the ability to describe object collaboration and common interaction behaviour, UML*i* activity diagrams provide greater support for UI design than UML activity diagrams.

This section explains how activities can be modelled from use cases, how activity diagrams can be simplified in order to describe common interactive behaviours, and how interaction objects can be related to activity diagrams.

4.1 Use Cases and Use Case Scenarios

Use case diagrams are normally used to identify application functionalities. However, use case diagrams may also be used to identify interaction activities. For instance, a ≪*communicates*≫ association between a use case and an actor indicates that the actor is interacting with the use case. Therefore, for example, in Figure 4 the `CollectBook` use case cannot identify an interaction activity since its association with `Borrower` is not a ≪*communicates*≫ association. Indeed, the `CollectBook` use case identifies a functionality not supported by the application.

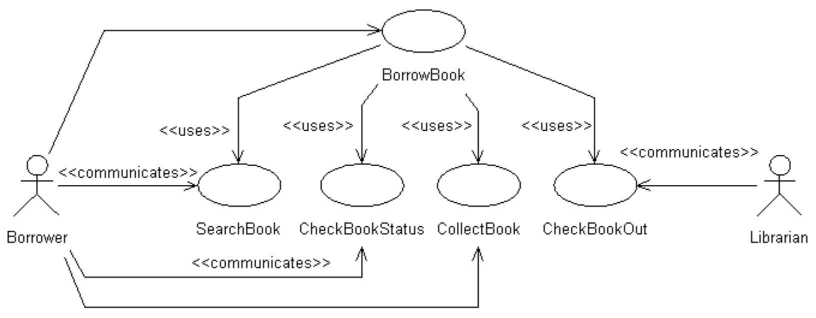

Figure 4: A use case diagram for the `BorrowBook` use case with its component use cases.

Use case scenarios can be used for the elicitation of actions [12]. Indeed, actions are identified by scanning scenario descriptions looking for verbs. However, actions may be classified as Inputters, Displayers, Editors or ActionInvokers. For example, Figure 5 shows a scenario for the `SearchBook` use case in Figure 4. Three interaction objects can be identified in the scenario: $\underline{\nabla \textit{providing}}$ that receives book's title, author and year information; $\underline{\nabla \textit{specify}}$ that specifies some query details; and $\underline{\triangle \textit{displays}}$ that presents the results of the query. Therefore, UMLi can start the elicitation of interaction objects, using this transformation of actions into interaction objects, during requirements analysis. These action transformations are often possible since the interaction objects of UMLi are abstract ones. The elicitation of these interaction objects early, as describe here, is important since it provides an initial description for abstract presentations. Indeed, user interface diagrams can initially be composed of interaction objects elicited from scenarios.

John is looking for a book. He can check if such book is in the library catalogue $\underline{\nabla \textit{providing}}$ *its title, authors, year, or a combination of this information. Additionally, John can* $\underline{\nabla \textit{specify}}$ *if he wants an exact or an approximate match, and if the search should be over the entire catalogue or the result of the previous query. Once the query has been submitted, the system* $\underline{\triangle \textit{displays}}$ *the details of the matching books, if any.*

Figure 5: A scenario for the `SearchBook` use case.

4.2 From Use Cases to Activities

UML*i* assumes that a set of activity diagrams can describe possible user interactions since this set can describe possible application workflows from application entry points. Indeed, transitions in activity diagrams are inter-object transitions, such as those transitions between interaction and domain objects that can describe interaction behaviours. Based on this assumption, those activity diagrams that belong to this set of activity diagrams can be informally classified as *interaction activity diagrams*. Activities of interaction activity diagrams can also be informally classified as *interaction activities*. The difficulty with this classification, however, is that UML does not specify any constructor for modelling application entry points. Therefore, the process of identifying in which activity diagram interactions start is unclear.

The *initial interaction state* constructor used for identifying an application's entry points in activity diagrams is introduced in UML*i*. This constructor is rendered as a solid square, ■, and it is used as the UML *initial pseudo-state* [9], except that it cannot be used within any state. A *top level interaction activity diagram* must contain at least one `initial interaction state`. Figure 6 shows a top level interaction activity diagram for a library application.

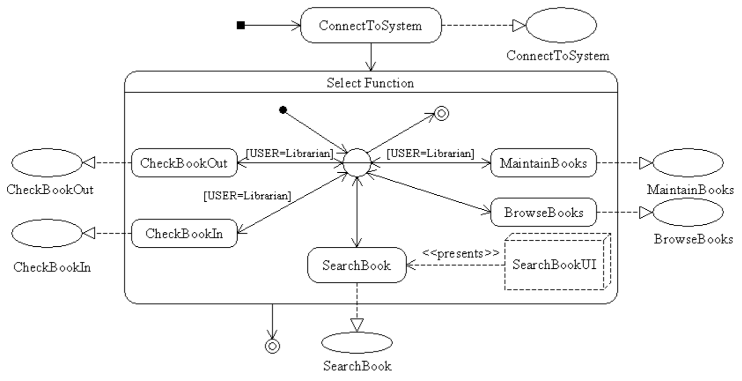

Figure 6: Modelling an activity diagram from use cases using UML*i*.

Use cases that communicate directly with actors are considered *candidate interaction activities* in UML*i*. Thus, we can define a *top level interaction activity* as an activity which is related to a candidate interaction activity. This relationship between a top level interaction activity and a candidate interaction activity is described by a realisation relationship, since activity diagrams can describe details about the behaviour of candidate interaction activities. The diagram in Figure 6 is using the UML*i* activity diagram notation explained in the next section. However, we can clearly see in the diagram which top level interaction activity realises which candidate interaction activity. For instance, the `SearchBook` activity realises the `SearchBook` candidate interaction activity modelled in the use case diagram in Figure 4.

In terms of UI design, interaction objects elicited in scenarios are primitive interaction objects that must be contained by FreeContainers (see the APP in Figure 3). Further, these interaction objects should be contained by FreeContainers associated with top-level interaction activities, such as the `SearchBookUI` FreeContainer in Figure 6, for example. Therefore, interaction objects elicited from scenarios are initially contained by FreeContainers that are related to top-level interaction through the use of a ≪*presents*≫ object flow, as described in Section 4.4. In that way, UI elements can be imported from use case diagrams to activity diagrams. For example, the interaction objects elicited in Figure 5 are initially contained by the `SearchBookUI` presented in Figure 6.

4.3 Selection States

Statechart constructors for modelling transitions are very powerful since they can be combined in several ways, producing many different compound transitions. In fact, simple `transitions` are suitable for relating activities that can be executed sequentially. A combination of `transitions`, `forks` and `joins` is suitable for relating activities that can be executed in parallel. A combination of `transitions` and `branches` is suitable for modelling the situation when only one among many activities is executed (choice behaviour). However, for the designing of interactive applications there are situations where these constructors can be held to be rather low-level, leading to complex models. The following behaviours are common interactive application behaviours, but usually result in complex models.

- The `order independent` behaviour is presented in Figure 7(a). There, activities `A` and `B` are called *selectable activities* since they can be activated in either order on demand by users who are interacting with the application. Thus, every selectable activity should be executed once during the performance of an order independent behaviour. Further, users are responsible for selecting the execution order of selectable activities. An order independent behaviour should be composed of one or more selectable activities. An object with the execution history of each selectable activity (*SelectHist* in Figure 7(a)) is required for achieving such behaviour.

- The `optional` behaviour is presented in Figure 7(b). There, users can execute any selectable activity any number of times, including none. In this case, users should explicitly specify when they are finishing the `Select` activity. Like the order independent behaviour, the optional behaviour should be composed of one or more selectable activities.

- The `repeatable` behaviour is presented in Figure 7(c). Unlike the order independent and optional behaviours, a repeatable behaviour should have only one associated activity. `A` is the associated activity of the repeatable behaviour in Figure 7. Further, a specific number of times that the associated activity can be executed should be specified. In the case of the diagram in Figure 7(c), this number is identified by the value of `X`.

An optional behaviour with one selectable activity can be used when a selectable activity can be executed an unspecified number of times.

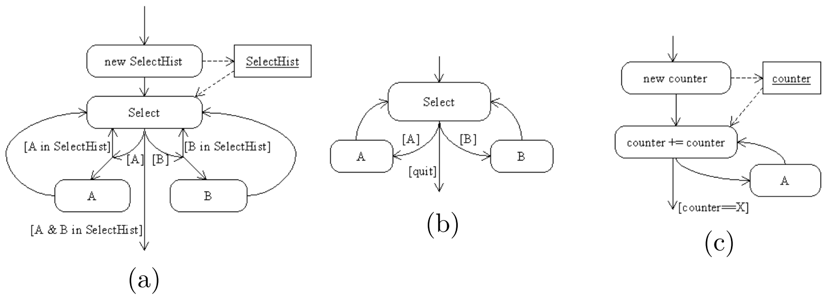

Figure 7: The UML modelling of three common interaction application behaviours. An `order independent` behaviour is modelled in (a). An `optional` behaviour is modelled in (b). A `repeatable` behaviour is modelled in (c).

As optional, order independent and repeatable behaviours are common in interactive systems [5], UML*i* proposes a simplified notation for them. The notation used for modelling an order independent behaviour is presented in Figure 8(a). There we can see an *order independent selector*, rendered as a circle overlying a plus signal, $\oplus$, connected to the activities `A` and `B` by *return transitions*, rendered as solid lines with a single arrow at the selection state end and a double arrow at the selectable activity end. The order independent selector identifies an *order independent selection state*. The double arrow end of return transitions identify the selectable activities of the selection state. The distinction between the selection state and its selectable activities is required when selection states are also selectable activities. Furthermore, a return transition is equivalent of a pair of Statechart transitions, one single transition connecting the selection state to the selectable activity, and one non-guarded transition connecting the selectable activity to the selection state, as previously modelled in Figure 7(a). In fact, the order independent selection state notation can be considered as a macro-notation for the behaviour described in Figure 7(a).

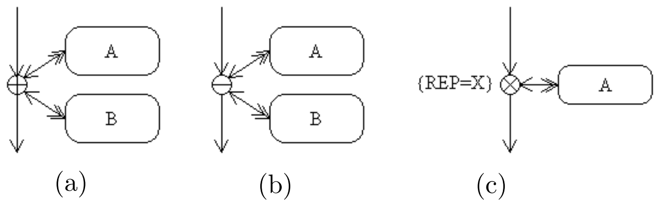

Figure 8: The UML*i* modelling of an `order independent` selection state in (a), an `optional` selection state in (b), and a `repeatable` selection state in (c).

The notations for modelling optional and repeatable behaviours are similar, in terms of structure, to the order independent selection state. The main difference between the notation of selection states is the symbols used for their selectors. The *optional selector* which identifies an *optional selection state* is rendered as a circle overlaying a minus signal, $\ominus$. The *repeatable selector* which identifies a *repeatable selection state*[2] is rendered as a circle overlaying a times signal, $\otimes$. The repeatable selector additionally requires a `REP` constraint, as shown in Figure 8(c), used for specifying the number of times that the associated activity should be repeated. The value `X` in this `REP` constraint is the `X` parameter in Figure 7(c). The notations presented in Figures 8(b) and 8(c) can be considered as macro-notations for the notation modelling the behaviours presented in Figures 7(b) and 7(c).

4.4 Interaction Object Behaviour

Objects are related to activities using *object flows.* Object flows are basically used for indicating which objects are related to each activity, and if the objects are generated or used by the related activities. Object flows, however, do not describe the behaviour of related objects within their associated activities. Activities that are action states and that have object flows connected to them can describe the behaviour of related objects since they can describe how methods may be invoked on these objects. Thus, a complete decomposition of activities into action states may be required to achieve such object behaviour description. However, in the context of interaction objects, there are common functions that do not need to be modelled in detail to be understood. In fact, UML*i* provides five specialised object flows for interaction objects that can describe these common functions that an interaction object can have within a related activity. These object flows are modelled as stereotyped object flows and explained as follows.

- An ≪*interacts*≫ object flow relates a primitive interaction object to an action state, which is a primitive activity. Further, the object flow indicates that the action state involved in the object flow is responsible for an interaction between a user and the application. This can be an interaction where the user is invoking an object operation or visualising the result of an object operation. The action states in the `SpecifyBookDetails` activity, Figure 9, are examples of Inputters assigning values to some attributes of the `SearchQuery` domain object. The △ `Results` in Figure 9 is an example of a Displayer for visualising the result of `SearchQuery.SearchBook()`. As can be observed, there are two abstract operations specified in the APP (Figure 3) that have been used in conjunction with these interaction objects. The `setValue()` operation is used by Displayers for setting the values that are going to be presented to the users. The `getValue()` op-

[2]UML*i* considers a *repeatable selection state* as a "selection" state since users might have the possibility of cancelling the repeatable state iteration.

eration is used by Inputters for passing the value obtained from the users to domain objects.

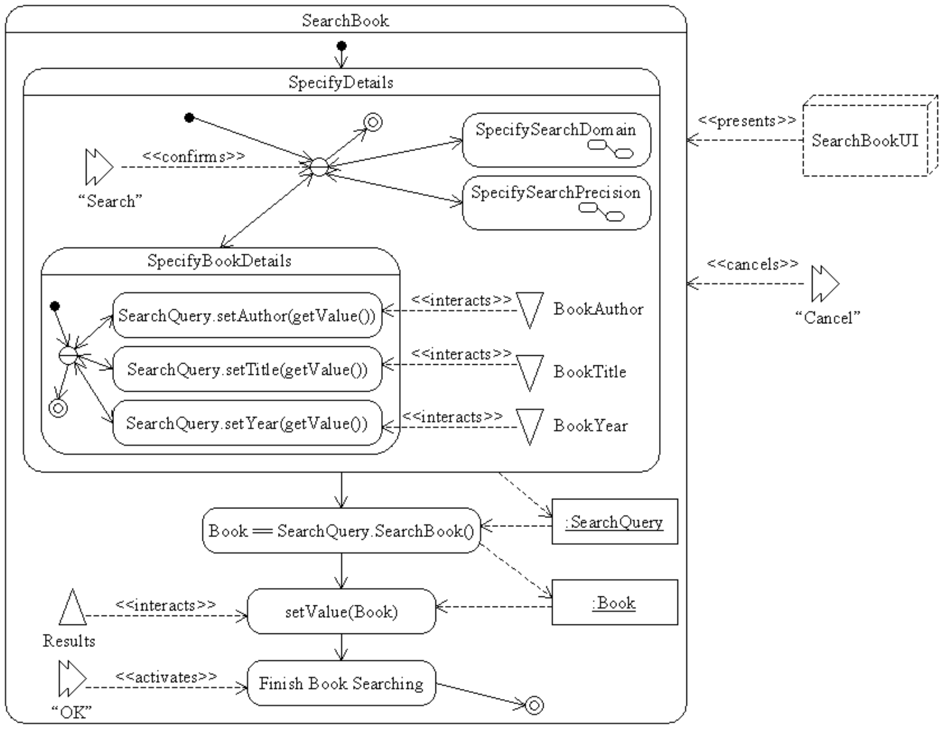

Figure 9: The `SearchBook` activity.

- A ≪*presents*≫ object flow relates a FreeContainer to an activity. It specifies that the FreeContainer should be visible while the activity is active. Therefore, the invocation of the abstract `setVisible()` operation of the FreeContainer is entirely transparent for the developers. In Figure 9 the `SearchBookUI` FreeContainer and its contents are visible while the `SearchBook` activity is active.

- A ≪*confirms*≫ object flow relates an ActionInvoker to a selection state. It specifies that the selection state has finished normally. In Figure 9 the event associated with the ▷"Search" is responsible for finishing the execution of its related selection state normally. An optional selection state must have one ≪*confirms*≫ object flow directly or indirectly related to it. The optional selection state in the `SpecifyDetails` activity in Figure 9 has the ▷"Search" directly related to it. The optional selection state in the `SpecifyBookDetails` relies on the ▷"Search" that is indirectly related to it. In fact, confirming the optional selection state in

`SpecifyDetails` a user is also confirming the optional selection state in `SpecifyBookDetails`.

- A ≪*cancels*≫ object flow relates an ActionInvoker to any composite activity or selection state. It specifies that the activity or selection state has not finished normally. The flow of control should be re-routed to a previous state. The ▷"Cancel" object in Figure 9 is responsible for identifying the user cancelling of the `SearchBook` activity.

- An ≪*activate*≫ object flow relates an ActionInvoker to an activity. In that way, the associated activity becomes a triggered activity, that waits for an event to effectively start, after being activated. This event that triggers the activity is the `defaultEvent` presented in the APP (Figure 3).

5 Using UML*i*: Method and Case Study

The UML*i* method is composed of eight steps. These steps are not intended to describe a comprehensive method for the modelling of a UI in an integrated way with the underlying application. For example, these steps could be adapted to be incorporated by traditional UML modelling methods such as Objectory and Catalysis.

A case study describing a Library Application [3] is used for exemplifying the use of the UML*i* method. Many results of this case study are used as examples of the UML*i* notation in previous sections.

Step 1 *User requirement modelling. Use cases can identify application functionalities. Use cases may be decomposed into other use cases. Scenarios provide a description of the functionalities provided by use cases.*

The use cases in Figure 4 identified some application functionalities. Scenarios can be used as a textual description of the use case goals. For instance, the scenario presented in Figure 5 is a textual description of the `SearchBook` use case in Figure 4. Further, scenarios can be used for the elicitation of sub-goals that can be modelled as use cases. Use cases that are sub-goals of another use case can be related using the ≪*uses*≫ dependency. Thus, the use of ≪*uses*≫ dependencies creates a hierarchy of use cases. For instance, `SearchBook` is a sub-goal of `BorrowBook` in Figure 4.

Step 2 *Interaction object elicitation. Scenarios of less abstract use cases may be used for interaction object elicitation.*

Scenarios can be used for the elicitation of interaction objects, as described in Section 4.1. In this case, elicited interaction objects are related to the associated use case. Relating interaction objects directly to use cases can prevent the elicitation of the same interaction object in two or more scenarios related to the same use case. Considering that there are different levels of abstraction for use cases, as described in Step 1, it was identified by the case study that interaction objects of abstract use cases are also very abstract, and may not be useful for exporting to activity diagrams. Therefore, the UML*i* method suggests that interaction objects can be elicited from less abstract use cases.

Step 3 *Candidate interaction activity identification.*

Candidate interaction activities are use cases that communicate directly with actors, as described in Section 4.1.

Step 4 *Interaction activity modelling. A top level interaction activity diagram can be designed from identified candidate interaction activities. A top level interaction activity diagram must contain at least one initial interaction state.*

Figure 6 shows a top level interactive activity diagram for the Library case study. Top level interaction activities may occasionally be grouped into more abstract interaction activities. In Figure 6, many top level interaction activities are grouped by the `SelectFunction` activity. In fact, `SelectFunction` was created to gather these top level interaction activities within a top level interaction activity diagram. However, the top level interaction activities, and not the `SelectFunction` activity, remain responsible for modelling some of the major functionalities of the application. The process of moving from candidate interaction activities to top level interaction activities is described in Section 4.2.

Step 5 *Interaction activity refining. Activity diagrams can be refined, decomposing activities into action states and specifying object flows.*

Activities can be decomposed into sub-activities. The activity decomposition can continue until the action states (leaf activities) are reached. For instance, Figure 9 presents a decomposition of the `SearchBook` activity introduced in Figure 6. The use of ≪*interacts*≫ object flows relating interaction objects to action states indicates the end of this step.

Step 6 *User interface modelling. User interface diagrams can be refined to support the activity diagrams.*

User interface modelling should happen simultaneously with Step 5 in order to provide the activity diagrams with the interaction objects required for describing action states. There are two mechanisms that allow UI designers to refine a conceptual UI presentation model.

- The inclusion of complementary interaction objects allows designers to improve the user's interaction with the application.
- The *grouping* mechanism allows UI designers to create groups of interaction objects using Containers.

At the end of this step it is expected that we have a conceptual model of the user interface. The interaction objects required for modelling the user interface were identified and grouped into Containers and FreeContainers. Moreover, the interaction objects identified were related to domain objects using action states and UML*i* flow objects.

Step 7 *Concrete presentation modelling. Concrete interaction objects can be bound to abstract interaction objects.*

The concrete presentation modelling begins with the binding of concrete interaction objects (widgets) to the abstract interaction objects that are specified by the APP. Indeed, the APP is flexible enough to map many widgets to each abstract interaction object.

Step 8 *Concrete presentation refinement. User interface builders can be used for refining user interface presentations.*

The widget binding alone is not enough for modelling a concrete user interface presentation. Ergonomic rules presented as UI design guidelines can be used to automate the generation of the user interface presentation. Otherwise, the concrete presentation model can be customised manually, for example, by using direct manipulation.

6 Conclusions

UML*i* is a UML extension for modelling interactive applications. UML*i* makes extensive use of activity diagrams during the design of interactive applications. Well-established links between use case diagrams and activity diagrams explain how user requirements identified during requirements analysis are described in the application design. The UML*i* user interface diagram introduced for modelling abstract user interface presentations simplifies the modelling of the use of visual components (widgets). Additionally, the UML*i* activity diagram notation provides a way for modelling the relationship between visual components of the user interface and domain objects. Finally, the use of selection states in activity diagrams provides a simplification for modelling interactive systems.

The reasoning behind the creation of each new UML*i* constructor and constraint has been presented throughout this paper. The UML*i* notation was entirely modelled in accordance to the UML*i* meta-model specifications [2]. This demonstrates that UML*i* is respecting its principle of being a non-intrusive extension of UML, since the UML*i* meta-model does not replace the functionalities of any UML constructor [2]. Moreover, the presented case study indicates that UML*i* may be an appropriate approach in order to improve UML's support for UI design. In fact, the UIs of the presented case study were modelled using fewer and simpler diagrams than using standard UML diagrams only, as described in [3].

As the UML*i* meta-model does not modify the semantics of the UML meta-model, UML*i* is going to be implemented as a plug-in feature of the ARGO/UML case tool. This implementation of UML*i* will allow further UML*i* evaluations using more complex case studies.

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

References

[1] F. Bodart, A. Hennebert, J. Leheureux, I. Provot, B. Sacre, and J. Vanderdonckt. Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide. In *Design, Specification and Verification of Interactive Systems*, pages 262–278, Vienna, 1995. Springer.

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

[3] 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).

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] T. Griffiths, P. Barclay, J. McKirdy, N. Paton, P. Gray, J. Kennedy, R. Cooper, C. Goble, A. West, and M. Smyth. Teallach: A Model-Based User Interface Development Environment for Object Databases. In *Proceedings of UIDIS'99*, pages 86–96, Edinburgh, UK, September 1999. IEEE Press.

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

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

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

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

[10] M. B. Rosson. Integrating Development of Task and Object Models. *Communications of the ACM*, 42(1):49–56, January 1999.

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

[12] R. Tam, D. Maulsby, and A. Puerta. U-TEL: A Tool for Eliciting User Task Models from Domain Experts. In *Proceedings of Intelligent User Interfaces'98*, pages 77–80, San Francisco, CA, January 1998. ACM Press.