OBJECT MODELLING OF INTERACTIVE SYSTEMS: THE UMLi APPROACH

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2002

By
Paulo Pinheiro da Silva
Department of Computer Science

Contents

Abstract **10**

Declaration **12**

Copyright **14**

Acknowledgements **15**

1 Introduction **17**

1.1 Object Modelling of User Interfaces 17
1.2 Object Modelling of Software Systems 18
1.3 User Interface Development Tools and Environments 19
1.4 Motivation for Work 20
1.5 The Unified Modeling Language for Interactive Systems (UML*i*) . 21
1.5.1 Principles 23
1.5.2 Contributions 24
1.6 Thesis Overview 25

2 UI Models and Development Environments **27**

2.1 Background 28
2.2 A Comparison Framework for User Interface Models 30
2.3 A Survey of User Interface Models 33
2.3.1 Modelled Aspects of User Interfaces 33
2.3.2 Constructs of User Interface Models 35
2.3.3 Notations Utilised in User Interface Models 35
2.3.4 Integration of User Interface Models 37
2.4 Summary 39

3 User Interface Modelling with UML **41**
3.1 Library System Case Study 41
3.2 Domain Modelling 43
3.3 Behaviour Modelling 45
3.3.1 Activity Refinement and Object Flows 47
3.3.2 Interactive Application Behaviour 49
3.4 Abstract Presentation Modelling 52
3.4.1 Abstract Presentation Pattern 53
3.4.2 Using the Abstract Presentation Pattern 55
3.5 Concrete Presentation Modelling 57
3.6 An Integrated View of UI Behaviour and Structure 59
3.7 Event Modelling 61
3.8 Summary 64

4 UML*i* Notation and Metamodel **67**
4.1 User Interface Diagram Modelling 68
4.1.1 User Interface Diagram Notation 69
4.1.2 Using the User Interface Diagram 70
4.2 Activity Diagram Modelling 72
4.2.1 From Use Cases to Activities: A Task Modelling Approach 73
4.2.2 Selection States 75
4.2.3 Interaction Object Flows 76
4.2.4 The UML*i* `SearchBook` Activity Diagram 78
4.3 UML Metamodel Architecture 80
4.3.1 The `Core` Package 82
4.3.2 The `StateMachines` and `ActivityGraphs` Packages 83
4.4 UML*i* Metamodel Architecture 85
4.4.1 The `UserInterfaces` Package 86
4.4.2 The `IntegratedActivities` Package 87
4.4.3 Extensions in the `DataTypes` and `StateMachines` Packages 89
4.4.4 Interaction Object Flows and Their Stereotypes 90
4.5 A Proposal for a UML*i* Method 90
4.6 Summary 93

5 UML*i* Semantics **95**
5.1 Approaches For a UML*i* Semantics 96

5.2 The LOTOS Specification Language 98
5.2.1 Basic LOTOS 98
5.2.2 Full LOTOS 99
5.3 From UML*i* Models to LOTOS Specifications: Foundations 103
5.3.1 The Activity Specification 103
5.3.2 The Transition and PseudoState Specifications 106
5.3.3 Type Mappings 108
5.4 Semantics for Structural Aspects of UML*i* 109
5.4.1 The Classifier Specification 109
5.4.2 The InteractionClass Specification 112
5.4.3 The PrimitiveInteractionClass Specification 120
5.4.4 The ActionInvoker Specification 122
5.4.5 The Container Specification 122
5.5 Semantics for Behavioural Aspects of UML*i* 124
5.5.1 The CreateAction ActionState Specification 124
5.5.2 The ObjectFlowState and ClassifierInState Specifications . . 125
5.5.3 The CallAction and SendAction ActionState Specifications . 126
5.5.4 The ≪*presents*≫ Stereotype Specification 127
5.5.5 The ≪*cancels*≫ Stereotype Specification 130
5.5.6 The OptionalState Specification 131
5.5.7 The ≪*Confirms*≫ Stereotype Specification 133
5.5.8 The DestroyAction ActionState Specification 134
5.5.9 The ≪*interacts*≫ Stereotype Specification 135
5.6 Verification of the Library System Specification 136
5.7 Summary 138

6 UML*i* Tool Support 141
6.1 ArgoUML: A UML-Based Development Environment 142
6.1.1 ArgoUML User Interface 142
6.1.2 ArgoUML Architecture 145
6.2 ARGO*i*: A UML*i*-Based Development Environment 146
6.3 User Interface Diagram Modelling Support 147
6.4 SelectionState Modelling Support 148
6.5 Interaction Object Flow Modelling Support 151
6.5.1 Selecting Types for Interaction Object Flows 152
6.5.2 Selecting Stereotypes for Interaction Object Flows 153

6.6 Domain Modelling Support 153
6.7 Summary 155

7 Metric Assessment of Resulting Models **158**
7.1 On the Assessment of Models using Metrics 159
7.2 Metric Definitions 161
7.3 Models Used in the Metric Study 164
7.3.1 UML*i*_UI Models 165
7.3.2 Mapping UML*i*_UI Models into UML_UI Models 166
7.3.3 Mapping UML_UI models into UML_noUI Models 168
7.3.4 Resulting Models of the Metric Study 169
7.4 A Metric Assessment of the `ConnectToSystem` Service Models . . 169
7.5 A Metric Assessment of the Library System Models 172
7.6 Summary 180

8 Conclusions and Future Work **182**
8.1 Work Overview and Contributions 182
8.2 Revisited UML*i* Principles 187
8.3 Conclusions 188
8.4 Future Work 189

A Additional Semantics for UML*i* **191**
A.1 Operation Specification 191
A.2 Association and ClassifierRole Specifications 194
A.3 Signal, Sender and Receiver Specifications 196
A.4 Generalisation Specification 197
A.5 A Generic Specification for Classifiers 198
A.6 The OrderIndependentState Specification 199
A.7 The ≪*activates*≫ Stereotype Specification 201
A.8 The RepeatableState Specification 201

Bibliography **203**

List of Tables

2.1 Component models of a user interface. 31
2.2 User interface model constructs. 32
2.3 Surveyed MD-UIDEs. 33
2.4 MB-UIDE's component models. 34
2.5 MB-UIDE's constructs. 36
2.6 Model notations. 37
2.7 Discrete representation of the inter-model relationships. 38

3.1 The UML diagrams used to model many aspects of UIs 66

5.1 Unary and binary operators of LOTOS. 100
5.2 Some type operators of LOTOS. 102
5.3 UCDs related to `PseudoState` constructs 107

7.1 Size of the models of the `ConnectToSystem` service. 169
7.2 CK metrics of the models of the `ConnectToSystem` service 170
7.3 Behavioural and visual metrics of the models of the `ConnectToSystem` service. 171
7.4 CK metrics of the models of the Library System 173
7.5 Behavioural and visual metrics of the models of the Library System. 178

8.1 UML*i* user interface model. 183

A.1 Association mapping. 195

List of Figures

2.1 Graphical representation of the inter-model relationships. 39

3.1 The use case diagram. 42
3.2 The domain model. 44
3.3 A partial top-level activity diagram of the Library System. 46
3.4 A decomposition of the `Connect` activity 49
3.5 The instantiation and destruction of `ConnectUI` 50
3.6 The UML modelling of common interaction application behaviours 51
3.7 The display of the `ConnectUI`. 53
3.8 The abstract presentation pattern. 54
3.9 The abstract presentation model of the `ConnectUI`. 56
3.10 The concrete presentation model. 58
3.11 A sequence diagram for the `ConnectToSystem` use case. 60
3.12 The event model. 61
3.13 The relationship between a UI and an exception handler. 63
3.14 Exceptions in activity diagrams. 64

4.1 UML*i* user interface models 68
4.2 The `ConnectUI` user interface diagram 69
4.3 A scenario for the `SearchBook` use case. 71
4.4 A preliminary version of the `SearchUI` user interface diagram . . 71
4.5 A refined version of the `SearchUI` user interface diagram 72
4.6 Modelling an activity diagram from use cases using UML*i*. 74
4.7 The modelling of UML*i* SelectionStates 76
4.8 The `Connect` activity. 77
4.9 The `SearchBook` activity. 79
4.10 Packages of the UML metamodel. 81
4.11 The Core Package of the UML metamodel 82

4.12 The StateMachines and ActivityGraphs packages of the UML metamodel 84
4.13 The UML constructs of an object flow. 85
4.14 Packages of the UMLi metamodel. 86
4.15 UserInterfaces package of the UMLi metamodel. 87
4.16 IntergratedActivities package of the UMLi metamodel. 88

5.1 Definition of a LOTOS process 99
5.2 Person type specification. 101
5.3 The LOTOS specification of the Library Activity. 108
5.4 Enumerate type specification. 109
5.5 Specification of EnumerateBookCopy from the Enumerate type. . . 109
5.6 BookCopy type specification. 110
5.7 Specification of Attributes in the BookCopy process. 111
5.8 An schematic view of the ADC interactor 113
5.9 The ADC interactor. 114
5.10 The ad type. 114
5.11 The ADU process. 117
5.12 The CC process. 119
5.13 The INITIATE_CONNECTUI_ACT process. 128
5.14 The INITIATE_CONNECT_ACT in the LIBRARY_ACT process. 130
5.15 Making the CONNECT_ACT cancellable. 132
5.16 A refinement of the CONNECT_ACT process 136
5.17 Snapshot of CADP 137

6.1 A snapshot of the ArgoUML user interface. 143
6.2 A package view of the ArgoUML architecture 145
6.3 The temporal-relation wizard 149
6.4 Resulting actions on temporal-relation wizards 150
6.5 The modelling of the Connect Activity in ARGOi 151
6.6 Triggering the integration wizard 154

7.1 The Connect activity of ConnectToSystem without its UI. 163
7.2 The ConnectToSystem-specific class diagram. 165
7.3 Distribution histogram of RFC 177

A.1 Specification of Operations in the BookCopy process. 193

A.2 Specification of Associations in the BookCopy process. 195
A.3 Specification of Signals in the BookCopy process. 196
A.4 Specification of the Reservation process 197
A.5 Specification of Borrower as a specialisation of Person. 198
A.6 A type for a generic Classifier. 199
A.7 A process for a generic Classifier. 200

Abstract

The Unified Modeling Language (UML) is a standardised notation for creating software application designs in a object-oriented manner. Developers should be able to model complete interactive systems using UML. However, interactive system models in UML describe few aspects of user interfaces (UIs). Model-based user interface development environments (MB-UIDEs) are a state-of-the-art approach to user interface development, which provide the ability to model and implement user interfaces in a systematic way. However, current MB-UIDE technologies can only be used to model user interfaces and not complete interactive systems. Therefore, there is a clear dichotomy between the state-of-the-art approaches to modelling user interfaces and their underlying systems.

The work conducted in this thesis investigates enhancing UML support for user interface design by incorporating MB-UIDE technologies into UML, providing in this way the necessary integration between user interfaces and their underlying systems. In particular, this thesis describes the specification and assessment of the Unified Modelling Language for Interactive Applications (UML*i*) – a conservative extension of UML. UML*i* provides a diagram notation for modelling user interface presentations. Further, it provides additional activity diagram notation for describing common behaviours of interactive systems and collaboration between interaction and domain objects. In fact, UML*i* addresses the difficulties identified in a case study where standard UML was used to model aspects of a user interface typically specified in surveyed MB-UIDEs.

By having a single modelling notation, common structures and behaviours of user interfaces and their underlying systems can be shared, verified and supported in an integrated way at design time. The description of UML*i* includes its grammar specified in terms of the UML metamodel, which provides the ability to implement ARGO*i*, a UML*i*-based tool that can import UML models from ArgoUML, a tool based on standard UML. This ability demonstrates that UML*i*

is a truly conservative extension of UML. The description of UMLi includes its semantics specified in terms of the Language of Temporal Ordering Specifications (LOTOS), which provides the ability to generate LOTOS specifications for complete interactive systems that can be model checked. This ability fully demonstrates that: UMLi constructs are integrated with UML constructs; and both UMLi and UML constructs can have a precise meaning. Finally, a metric study suggests that the inherent structural, behavioural and visual complexity of UMLi models of an interactive application are significantly lower than in UML models describing the same set of properties of the application.

UMLi is one of several approaches towards the integration of MB-UIDE facilities with a standard object-oriented modelling language. However, it is the most comprehensively specified, implemented and assessed approach to date.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning. However, some material presented in this thesis has been or will be published as follows:

1. Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In *Interactive Systems: Design, Specification, and Verification* (7th International Workshop DSV-IS, Limerick, Ireland, June, 2000). LNCS Vol. 1946, pages 207-226, Springer, 2000.

2. Paulo Pinheiro da Silva. A Proposal for a LOTOS-Based Semantics for UML. *Technical Report UMCS-01-06-1*, Department of Computer Science, University of Manchester. June, 2001. Also submitted for publication as a journal article.

3. Paulo Pinheiro da Silva and Norman W. Paton. UMLi: The Unified Modeling Language for Interactive Applications. In *Proceedings of the 3rd International Conference on the Unified Modeling Language ($\ll$UML$\gg$2000)*, York, United Kingdom, October, 2-6. LNCS Vol 1939, pages 117–132, Springer, 2000.

4. Paulo Pinheiro da Silva and Norman W. Paton, User Interface Modelling with UML. In *Information Modelling and Knowledge Bases XII* (10th European-Japanese Conference on Information Modelling and Knowledge Representation, Saariselkä, Finland, May, 8-11, 2000), pages 203-217, IOS Press, Amsterdam, 2001.

5. Paulo Pinheiro da Silva and Norman W. Paton. A UML-Based Design Environment for Interactive Applications. In *Proceedings of the 2nd International Workshop on User Interfaces to Data Intensive Systems (UIDIS'01)*, Zürich, Switzerland, Pages 60-71, IEEE Computer Society, 2001.

6. Paulo Pinheiro da Silva and Norman W. Paton. Object Modelling of User Interfaces in UML*i*. Submitted for publication.

Copyright

Acknowledgements

I would like to thank Norman W. Paton for his professional and dedicated work as my supervisor. In particular, I would like to thank him for his patience in coping with my "latin" way of discussing ideas.

I would like to thank Alvaro Fernandes for his support in the development of the metric study of UMLi, and for being constantly available for chatting about my current work, my future work and our homeland, Minas Gerais.

I would like to thank Enrico Franconi for his comments on the approach to providing a semantics for UMLi.

I would like to thank my parents and family that made me what I am today.

I would like to thank Pedro Sampaio for his support to my idea of pursuing a Ph.D. here in Manchester since the first time I contacted him through e-mail.

I would like to thank Martin Peim, Marcelo Aragão and Sergio Tessaris for their support on the use of LaTeX and Emacs.

I would like to thank my friends Anand, Ane, Akthar, Carole, Cornelia, Elon, Fouzia, Gary, Ian, Jeff, Luciano, Nassima, Norman Murray, Pierpaolo, Robert, Sandra, Sean, Steve, Tony and Veruska that I have met here in the Department of Computer Science.

I would like to thank all friends I have had opportunity to meet in the Brian Redhead Court, Hulme, Manchester, a place that has amazingly managed to be cozy.

This work was supported by a grant from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) – Brazil.

I DEDICATE THIS THESIS TO
MY WIFE ELISABETH AND MY DAUGHTER GIOVANA

Chapter 1

Introduction

UML [18, 99] is the notation most widely used for modelling entire software systems. An interactive software system consists of application functionality and the user interface. Therefore, the user interface, as an important part of an interactive system, should also be included as part of a UML model of the system. However, UML has not been widely accepted by user interface developers, which hints at its lack of support for modelling interactive systems [75, 97, 108]. In fact, it is by no means universally agreed how best to model user interfaces using UML. Therefore, there is a need to enhance UML with better support for the design of interactive systems.

1.1 Object Modelling of User Interfaces

Object modelling of user interfaces was introduced by Smalltalk [61, 44] during the 70s. In fact, Adele Goldberg, Daniel Ingalls and Alan Kay among other researchers at Xerox PARC introduced the use of graphical user interfaces (GUIs) in Smalltalk as an alternative to the textual user interfaces in use in those days. Furthermore, these developers incorporated into Smalltalk the ideas of Engelbart [35] on using windows to support the consultation of multiple sources and on using the mouse as the principal pointing device.

The object modelling and implementation of user interfaces is undoubtedly a story of success in computer science. This fact can be corroborated by observing the GUIs available in almost every computer these days. To reach this level of prevalence, several complex problems have been overcome relating to visual interface development. For instance, almost every large company developing software has spent many years developing user interface style guidelines, e.g., Microsoft Windows [90] and IBM Common User Access [60], to facilitate the development of interfaces with a consistent *look-and-feel*. Further, the development of interface architectures, e.g., Model-View-Controller (MVC) [76] and Presentation-Abstraction-Control (PAC) [29], have also demanded years of practical work and research to explain the dependencies among the different categories of objects used to compose user interfaces. Interaction objects that embed within them complex behaviours and graphical algorithms for rendering their visual appearance have been developed in parallel with the style guidelines and interface architectures. These interaction objects are popularly called *widgets* and they are collectively referred to as *toolkits*.

Object modelling of user interfaces could be defined as the specification of user interfaces based on the properties of the artifacts such as widgets, interface architects and style guidelines actually used to implement user interfaces. For a comprehensive discussion on the evolution of user interfaces in general we suggest Shneiderman [125]. For a discussion of the evolution of object modelling of user interfaces we suggest Collins [28].

1.2 Object Modelling of Software Systems

The non-object-oriented design methods used during the 80s, e.g., [43], started to present difficulties for specifying software systems that eventually would be coded in object-oriented programming languages (OOPLs). In fact, such specifications were not based on the concepts of classes and objects used by OOPLs. To cope with this mismatch, many object-oriented methods, e.g., Coad and Yourdon's Object Oriented Analysis and Design (OOAD) [26, 27], Rumbaugh et al.'s Object Modeling Technique (OMT) [121], Jacobson et al.'s Object-Oriented Software Engineering (OOSE) [66], and Booch's method [17], were developed during the 80s and early 90s. Each of these methods was based on a slightly different set of notations and processes. The large number of methods and notations, however, was confusing for organisations (and people) involved in the object modelling of software systems. The Unified Modelling Language (UML) [18] proposed in 1994 is the most significant attempt to establish a widely accepted object modelling method. UML is the result of integrating Booch's method, Rumbaugh's OMT and Jacobson's OOSE. Later, in 1997, the Object Management Group (OMG) of organisations working with object-oriented technologies became responsible for carrying out improvements to standardisation of the UML specification [99].

Modelling methods based on UML constitute the typical approach to object modelling of software systems these days. The object modelling of interactive systems could be defined as the object modelling of systems and their user interfaces, preferable performed in an integrated way. A noteworthy point, however, is that object modelling of user interfaces is largely ignored by object-oriented methods, including the UML.

1.3 User Interface Development Tools and Environments

The development of user interfaces is challenging due to the necessity of matching concepts understood by users with concepts implemented by systems [71]. In fact, users tend to make use of concepts that are much more complex than those implemented in user interfaces. The development of new interaction facilities such as GUIs and windows has improved the match between user concepts and system implementation of concepts. For instance, such facilities can help the implementation of concepts that are much more complex than those that could be implemented, for example, in textual user interfaces. From a user's point-of-view, these new interaction facilities may give rise to interfaces that are more flexible and easier to interact with. However, from a user interface developer's point-of-view, such facilities may indicate the necessity of modelling and implementing objects that can deal, for example, with multiple asynchronous input devices, multiple strategies for accepting commands, and complex graphics [93].

User interface development tools are widely used these days. Toolkits constitute a category of software for developing user interfaces. Many toolkits are supported by *user interface builders* that simplify the task of handling the large number of widgets normally available in toolkits. Indeed, developers can select, combine, and customise widgets directly in a GUI using interactive techniques such as direct manipulation. Toolkits and UI builders are probably the most popular tools for object modelling of user interfaces. The big shortcoming of such toolkits and UI builders, however, is that they do not specify the behaviour of widgets in interacting with users and collaborating with other objects. For example, using a UI builder a developer can specify the placement and visual appearance of a button in a user interface. However, the developer may not be able to specify when the button may be active and when it may be inactive. To cope with such limitations, UI development tools evolved into *user interface management systems* (UIMS). The main idea behind UIMS was to provide a runtime component responsible for managing running interfaces generated from user interface specifications. The use of some notations mainly for specifying UI behaviour, e.g., Statecharts [53, 52] and User Action Notation (UAN) [54, 56], was introduced by UIMSs. Thus, such a runtime component would promote some measure of *independence* between applications and their user interfaces. Nevertheless, UIMSs would be able to generate code or prototypes of user interfaces early in the development process. One of the major shortcoming of UIMSs, however, was the difficulty of abstracting their user interface specifications, making the task of creating these specifications too difficult.

Model-Based User Interface Development Environments (MB-UIDEs) are a state-of-the-art approach for modelling and implementing running user interfaces from user interface models. The possibility of specifying user interface properties in an abstract way has facilitated a more systematic modelling of user interfaces. Nevertheless, MB-UIDEs have inherited from UIMSs the benefits of prototyping and generating user interface code. Two distinct generations of MB-UIDEs can be identified. In the first generation of MB-UIDEs, user interface structures were much more abstract than those used in UIMSs. User interface behaviours, however, were mainly specified in terms of dialog diagrams based on concepts such as states, transitions, pre-conditions, post-conditions and side-effects. In the second generation of MB-UIDEs, the replacement of dialog models by task models has simplified the specification of user interface behaviours.

For a comprehensive discussion about user interface tools and development environments we suggest the reading of Myers [93]. Chapter 2 in this dissertation provides a comprehensive discussion of user interface models in MB-UIDEs.

1.4 Motivation for Work

From the point of view of a system designer, UML [18, 99] is the notation most widely used for modelling entire software systems. Therefore, the user interface, as a significant part of most software systems [95], should also be modelled using

UML. In fact, UML is a natural candidate notation for object modelling of user interfaces. However, it is by no means clear how best to model user interfaces using UML [75, 97, 103, 108].

From the point of view of a user interface designer, a MB-UIDE may be an appropriate approach for modelling user interfaces. In fact, the potential benefits of MB-UIDEs are apparent, but to date research has not yet matured sufficiently to make such systems widely adopted in practice. However, believing in the potential of the model-based approach, we still face the problem that they are not well-integrated with techniques for modelling mainstream software systems. In fact, the modelling of user interfaces is not at all well handled using standard object modelling techniques. As Collins [28] states, "published methods for user interface design largely ignore the fact that it is part of product development". This is still a problem that can be corroborated in classical object-oriented design methods [17, 26, 27, 66, 121] as in UML [18, 99].

Considering the poor integration of the facilities for modelling interactive systems, the main aim of this dissertation is to remove this dichotomy between object modelling of user interfaces and mainstream systems by integrating the new facilities provided by MB-UIDEs with the standard object modelling techniques provided by the UML. Moreover, this dissertation aims to amass as much evidence as possible that demonstrates the benefits of such an integration.

1.5 The Unified Modeling Language for Interactive Systems (UML*i*)

This dissertation proposes a set of extensions to UML in order to improve the support that the language provides for modelling interactive systems. The necessity of extending UML comes from the shortcomings of UML identified in a case study developed in Chapter 3. The case study is an attempt to model the user interface aspects usually described in MB-UIDEs, as identified in Chapter 2, using standard UML. The identified shortcomings can briefly be described as follows.

- The diagrams of UML for structural modelling, i.e., class diagrams and object diagrams, do not provide much support for modelling the structural properties of user interfaces, which correspond to the visual part (presentation) of user interfaces. For instance, both containment among interactive objects and the abstract roles that interaction objects can play in the composition of user interfaces are unclear in UML models.

- The diagrams of UML for dynamic modelling, i.e., sequence diagrams, collaboration diagrams and activity diagrams, do not provide much support for modelling certain categories of behaviour commonly observed in interactive systems. For instance, it is difficult to model functionalities achieved, for example, through the execution of an activity which can be performed many times (repeatable), or through the execution of a set of activities each of which must be performed once, but in any order (order independent).

- The diagrams of UML in general do not provide much support for modelling the collaboration among interactive objects. For instance, it is not trivial to specify that the activity or inactivity of an interaction object depends on the activity or inactivity of the interaction object that contains it.

- The diagrams of UML in general do not provide much support for modelling the collaboration between interactive objects and non-interactive objects. For instance, it is difficult to model the part of the system data-flow that includes the process of storing in domain (non-interactive) objects information provided by users using interactive objects. Furthermore, it is also difficult to keep structural and behavioural diagrams consistent with respect to the dependencies created by the collaboration of domain objects with interactive objects.

The proposed extension improves the support that UML can provide for the identified problems, as assessed by the implementation of a UML*i*-based tool (presented in Chapter 6) and by a metric comparison of UML and UML*i* diagrams (demonstrated in Chapter 7). The proposed extension is named the Unified Modeling Language for Interactive Systems (UML*i*). Its graphical notation and syntax specified in terms of the UML metamodel are presented in Chapter 4 and its semantics is presented in Chapter 5.

This dissertation, and UML*i* as a consequence, are developed within the following scope.

- **Focus on modelling rather than implementing user interfaces.** The identification of a better way of specifying user interfaces and generating running user interfaces from user interface specifications are two major concerns in the context of MB-UIDEs. In this dissertation the focus is on specifying user interfaces.

- **Focus on describing user interface aspects rather than on the process of constructing user interface descriptions.** The identification and validation of processes for modelling user interfaces is as important as the development of facilities for modelling user interface. In the case of UML*i*, the focus is on the development of facilities for describing user interface aspects. Nevertheless, a process describing the use of the UML*i* facilities is outlined in Section 4.5, although the specific merits of this process and its evaluation have not been a major focus.

- **Focus on the UML notation.** This is the object modelling notation widely used by practitioners and researchers. Further, UML is the language that is intended to be an industrial standard for modelling software systems. Thus, classical object modelling notations, e.g., OMT [121], are not considered in this dissertation.

- **Focus on form-based user interfaces.** This is the category of user interfaces most used in important data intensive applications such as database systems and web-based applications. Moreover, the problem of how best to model form-based user interfaces using UML is unsolved (as is the modelling of any other category of user interface in UML). Thus, the identification of significant benefits for modelling an impoverished UI style using UML can be even more significant for modelling more complex interfaces. Restricting the scope to form-based user interfaces means that interfaces for systems such as games, word processors and simulators are beyond the scope of this dissertation.

1.5.1 Principles

There are five principles which have guided the development of UML*i* [109].

Principle 1 *The UMLi proposal should be a conservative extension of UML – standard UML should be retained as a subset, in which existing constructs keep their roles and semantics.*

Principle 2 *The UMLi proposal should introduce as few new models and constructs into the UML as possible.*

Principle 3 *The UMLi proposal should support the expectations of current UML modellers, whose experience with UML should be of benefit when using user interface-specific extensions.*

Principle 4 *The UMLi proposal should support the expectations of user interface modellers who have experience using existing user interface modelling techniques. Such users should not feel that they are having to design interfaces with less supportive facilities than are provided by MB-UIDEs.*

Principle 5 *The UMLi proposal should support the modelling of complete interactive systems, so the links between user interface models and existing UML models should be well-defined and close.*

1.5.2 Contributions

The major contribution of this dissertation is the development and assessment of the UML*i*, a comprehensive proposal for improving the support that UML provides for modelling interactive systems. The modelling of interactive systems, however, is a complex task that creates many difficulties for model developers. Therefore, many contributions have been achieved during the process of developing, assessing and using UML*i*. These contributions are:

- The development of a comparison framework for MB-UIDEs used to identify a set of common modelled characteristics of user interfaces.
- The identification of the strengths and weaknesses of UML for modelling user interfaces.
- The development of a semantics for UML*i* based on the LOTOS specification language [16, 63], which brings the formalisation of interactive systems proposed by Markopoulos [82] and Paternò and Faconti [104] into the context of UML.
- The development of a UML*i* development environment:
  - Demonstrating how UML*i* can be implemented in a generic UML-based tool;

  - Identifying a set of tool facilities (wizards) that can be implemented for UML*i* in order to simplify some tasks frequently performed during the modelling of user interfaces;
  - Demonstrating that UML*i* is an conservative extension of UML. (1) UML*i* models can be developed from UML models developed originally in a standard UML tool. (2) The textual format for model exchange of UML*i*, UML*i* XMI, can be implemented by extending the textual format of UML, UML XMI [99].

- The development of a case study comparing the cost in terms of design metrics of modelling an interactive system with UML*i* and standard UML. This study demonstrates that (1) standard UML does not scale up well for modelling interactive systems, and that (2) UML*i* actually reduces the complexity of modelling interactive systems when compared with UML.

In addition, there is an auxiliary contribution that has resulted from the development of UML*i*.

- The development of a strategy for mapping UML models into LOTOS specifications. Further, using such a strategy it is demonstrated how some structural and behavioural constructs of UML can be mapped into LOTOS specifications.

1.6 Thesis Overview

The remainder of this dissertation is organised as follows:

Chapter 2 surveys the MB-UIDE literature identifying the aspects usually described by user interface models in MB-UIDEs. A comparison framework for MB-UIDEs identifies these common user interface aspects.

Chapter 3 exploits the facilities provided by the standard UML to model the user interface aspects identified in Chapter 2. The result of this study is the identification of UML's strengths and weaknesses for modelling user interfaces.

Chapter 4 proposes UMLi, a conservative extension of UML, that provides new constructs and models to cope with the weaknesses of UML identified in Chapter 3.

Chapter 5 proposes a syntax for UMLi based on the UML metamodel [99]. Further, a semantics based on the LOTOS specification language [16, 63] is provided for the UMLi constructs.

Chapter 6 presents the design and implementation of ARGOi, a UMLi-based modelling environment. Some modelling facilities which can be provided exploiting UMLi are also presented in this chapter.

Chapter 7 describes a metric evaluation of UMLi models when compared with their corresponding models described using standard UML. A metric evaluation is provided for each UMLi construct.

Chapter 8 concludes the dissertation by discussing the contributions achieved throughout this research. Finally, future work related to this dissertation is presented.

Chapter 2

UI Models and Development Environments

Model-based user interface development environments (MB-UIDEs) belong to a new category of tools for developing user interfaces (UIs) through the construction of user interface models (UIMs). These models describe, for example, the domain over which the user interface acts, the tasks that the user interface supports, and various aspects of the display presented to the user (e.g., the windows and interface components used). These models are typically at a conceptual level, and thus are potentially implementable in different ways on different delivery platforms. The ability to specify user interfaces in a conceptual manner, and to verify and deploy user interfaces from the models, are the major benefits of MB-UIDEs. Thus, MB-UIDEs can assist developers in the process of specifying UIMs, e.g., by using sophisticated techniques for exploiting the relationships between model's constructs, and can deploy user interfaces from UIMs, e.g., by generating user interface code.

Model-based user interface development technologies aim to provide an environment where developers can design and implement user interfaces in a professional and systematic way, more easily than when using traditional UI development tools. To achieve this aim, UIs are described using models. There are three major advantages that derive from use of user interface models.

- They can provide a more abstract description of the UI than those provided by UI development tools such as Visual Basic and Tcl/Tk [145, 115].

- They facilitate the creation of methods for designing and implementing the

UI in a systematic way, since they offer capabilities: (1) to model user interfaces using different levels of abstraction; (2) to incrementally refine the models; and (3) to re-use UI specifications.

- They provide the infrastructure required to automate tasks related to the UI design and implementation processes [135].

A major disadvantage of the model-based approach is the complexity of the models and their notations, which are often hard to learn and use [93, 135]. However, it is expected that an appropriate environment should help to overcome the complexity problem, by providing features such as graphical editors, assistants and design critics to support UI designers. Moreover, it is expected that the use of standard notations in this environment may also ease the process of learning how to construct UIMs. The development of model-based user interface development environments is still challenging since some problems related to this technology are not completely solved.

- It is difficult to demonstrate that UIMs describe the aspects of the UI required to generate running user interfaces, although there are a few examples of systems for generating running interfaces from UIMs [144, 131].

- The problem of how best to integrate UIs with their underlying applications is introduced in many papers [30, 31] but is not entirely addressed for user interfaces generated by MB-UIDEs.

- There is no consensus as to which set of models is the most suitable for describing user interfaces. Indeed, there is no consensus as to which aspects of user interfaces should be modelled.

This chapter is structured as follows. Section 2.1 describes the evolution of MB-UIDE. Section 2.2 introduces a framework for comparing and analysing the architectural components of the UIMs. Section 2.3 presents how user interfaces are described through models in 15 MB-UIDEs. Conclusions are presented in Section 2.4.

2.1 Background

The literature contains many papers describing MB-UIDEs and their UIMs. The following classification of MB-UIDEs is presented to show how this category of

UI development tool has been introduced and evolved by both academic and industry communities.

The first generation of MB-UIDE appeared as improvements to the earlier user interface management systems (UIMSs) since they sought to execute user interfaces represented in a more abstract way. The main aim of the MB-UIDEs of this generation was to provide a strategy to execute a UI from the UIM. Examples of the the first generation of MB-UIDEs are COUSIN [55], HUMANOID [133], MIKE [100], UIDE [72, 41, 40] and UofA* [129]. However, the UIMs of the first generation of MB-UIDEs did not provide a high-level of abstraction for the description of the UI. For instance, behavioural models focused on dialogue description required the specification of many operations in objects represented in UI models that may be embedded in widgets these days. Further, user interface aspects like layouts and widget customisation appeared early during the UI design process.

A second generation of MB-UIDEs provided mechanisms for describing UIs at a higher level of abstraction [146]. In particular, they consolidated the use of task models as the primary model for behavioural modelling. The dialogue models that were often used for behavioural modelling in MB-UIDEs of the first generation were either removed or preserved as a complementary mechanism for refining task specifications. Examples of the second generation of MB-UIDEs are ADEPT [85], AME [86], DIANE+ [137], FUSE [80], GENIUS [68] MASTERMIND [135], MECANO [111], MOBI-D [115], TADEUS [34], Teallach [47] and TRIDENT [13]. With MB-UIDEs of the second generation, developers have been able to specify, generate and execute user interfaces. Further, this second generation of MB-UIDE has a more diffuse set of aims than the previous one. Some MB-UIDEs, e.g., AME and TADEUS, consider the use of computer-aided software engineering (CASE) tools and notations such as OMT [121] in their development environment. Others, e.g., MODI-D and Teallach aim to achieve complete UI development.

A subset of the MB-UIDEs of the second generation is characterised by the adoption of UML or by an attempt to adopt UML as the underlying notation for UIMs. Examples of these UML-concerned MB-UIDEs are ConcurTaskTree [101], Markopoulos' approach [84] and WISDOM [97]. By contrast with the other MB-UIDEs of the second generation, the UML-concerned MB-UIDEs are less focused on tools and more focused in the exploitation of the UML notation for representing UIMs. One reason for this is that UML-based tools, e.g, Rational Rose [116] and Argo/UML [118], can be used for modelling UIs once it is explained how the UML notation can be used for modelling UIs. UML-concerned MB-UIDEs also address the problem of representing task models using UML constructs [103]. In fact, the use of task models has been sufficiently successful in many non-UML-concerned MB-UIDEs of the second generation to suggest that they should be retained in UML-concerned MB-UIDEs.

Most of the papers describing the MB-UIDEs listed above compare some of their features with other MB-UIDEs, showing the differences among them. However, they are focused more on introducing the new approach than on comparing MB-UIDE proposals. There are a few papers that provide overviews of the MB-UIDE field: Schlungbaum [122], Vanderdonckt [141] and Griffiths et al. [49] provide comparisons among many MB-UIDEs, and Szekely [135] provides an excellent insight into what an MB-UIDE is. Pinheiro da Silva [106] provides a generic description of user interface design processes in MB-UIDEs, on which this chapter is based.

2.2 A Comparison Framework for User Interface Models

User interfaces convey the outputs of applications to users and the inputs from application users. For this reason, UIs have to cope with the complexity of both the applications and the users. In terms of MB-UIDE's architectures, the problem of accommodating application complexity and user interaction complexity is reflected in the fact that MB-UIDEs usually have several models, referred to in this chapter as *component models* or *models*, describing different aspects of the UI. Table 2.1 presents the four categories of model considered in the framework, also presenting which aspects of the user interface are described by each model.

As the purpose of this framework is to provide a comparison among different UIMs, four points should be considered:

- Task models and dialogue models are classified within a single model called the *Task-Dialogue model*. Both task models and dialogue models describe the possible tasks that users can perform during the interaction with the

Component Model	Abbrev.	Function
Application model	AM	Describes the properties of the application relevant to the UI.
Task-Dialogue model	TDM	Describes the tasks that users are able to perform using the application, as well as how the tasks are related to each other.
Abstract presentation model	APM	Provides a conceptual description of the structure of the visual parts of the user interface. The UI is described in terms of abstract objects.
Concrete presentation model	CPM	Describes in detail the visual parts of the user interface. It is explained how the UI is composed in terms of widgets.

Table 2.1: Component models of a user interface.

application, but at different levels of abstraction. The reason for classifying them together is that UIMs often only have one of them. Further, the possible constructs of task and dialogue models may have similar roles.

- User models are supported in some UIMs (e.g, ADEPT, MECANO and TADEUS). Indeed, user models are important for model-based user interface technologies since they can provide a way to model preferences for specific users or groups of users. However, they are a challenging aspect of the UI that is generally not well-addressed in MB-UIDEs, and not clearly described in the literature. Moreover, in those UIMs that have a user model, it appears that the user model can be replaced by design guidelines. In fact, design guidelines usually contain user preferences that can be considered as a model of a group of users.

- Requirements models are supported by few MB-UIDEs. The elicitation of top-level functionalities and actors (or agents) related to these functionalities provided by use case diagrams is a bonus for UML-concerned MB-UIDEs. Furthermore, there are MB-UIDEs such as TRIDENT that analyse ergonomic, functional and user interface requirements to construct UIMs. However, it is not clearly described in the literature how requirements can be modelled in MB-UIDEs.

- Platform models, as in MODI-D [115], are not included in the framework since they are considered by few MB-UIDEs and they are not clearly described in the literature.

Comp. model	Construct	Abbrev.	Function
AM	class	CLASS	An object type defined in terms of attributes, operations and relationships.
	attribute	ATTR	A property of the thing modelled by the objects of a class.
	operation	OPER	A service provided by the objects of a specific class.
	relationship	RELAT	A connection among classes.
TDM	task	TASK	An activity that changes the state of specific objects, leading to the achievement of a goal. Tasks can be defined at different levels of abstraction, which means that a task can be a sub-task of a more abstract class.
	goal	GOAL	A state to be achieved by the execution of a task.
	action	ACTION	A behaviour that can be executed. Actions are the most concrete tasks.
	sequencing	SEQ	The temporal order that sub-tasks and actions must respect when carrying out the related high-level tasks.
	task pre-condition	PRE	Conditions in terms of object states that must be respected before the execution of a task or an action.
	task post-condition	POST	Conditions in terms of object states that must be respected after the execution of a task or an action.
APM	view	VIEW	A collection of abstract interaction objects (AIOs) logically grouped to deal with the inputs and outputs of a task.
	abstract interaction object	AIO	A user interface object without any graphical representation and independent of any environment.
CPM	window	WINDOW	A visible and manipulable representation of a a view.
	concrete interaction object	CIO	A visible and manipulable user interface object that can be used to input/output information related to user's interactive tasks.
	layout	LAY	An algorithm that provides the placement of CIOs in windows.

Table 2.2: User interface model constructs.

Models are primarily composed of constructs. Table 2.2 shows the constructs considered in the framework. The table also gives a possible distribution of these constructs into the component models, a concise description of each construct, and abbreviations for future reference. The distribution of the constructs into component models, as presented in Table 2.2, helps to clarify their function in the framework. Definitions of application model constructs are partially extracted from UML [18]. Definitions of task model constructs are partially extracted from Johnson [69]. Definitions of abstract and concrete interaction objects are partially extracted from Bodart and Vanderdonckt [15].

One point that should be considered in terms of constructs is that MB-UIDEs do not need to have all constructs presented in the framework. Further, constructs can be distributed in a different manner from that proposed in Table 2.2.

MB-UIDE	References	Generat.	Organisation
ADEPT	[85, 70, 146]	2nd	Queen Mary and Westfield College, UK
AME	[86]	2nd	Fachhochschule Augsburg, Germany
CTT	[101, 103]	2nd	University of Pisa, Italy
FUSE	[123, 124, 80]	2nd	Technische Universität München, Siemens AG, Germany
HUMANOID	[133, 134, 81]	1st	University of Southern California, USA
JANUS	[5, 6]	2nd	Ruhr-Universität Bochum, Germany
ITS	[144, 145]	1st	IBM T.J. Watson Research Center, USA
MASTERMIND	[135, 22, 131]	2nd	University Southern California, Georgia Inst. Tech., USA
MECANO	[111]	1st/2nd	Stanford University, USA
MODI-D	[115, 113, 114]	2nd	Stanford University, Redwhale Software, USA
TADEUS	[34]	2nd	Universität Rostock, Germany
TEALLACH	[47, 48]	2nd	Univ. Manchester, Univ. Glasgow, Univ. Napier, UK
TRIDENT	[13, 12, 15]	2nd	Facultés Universitaires Notre-Dame de la Paix, Belgium
Markopoulos	[84]	2nd	Technische Universiteit Eindhoven, The Netherlands
WISDOM	[98, 97]	2nd	Universidade de Madeira, Universidade do Porto, Portugal

Table 2.3: Surveyed MD-UIDEs.

2.3 A Survey of User Interface Models

This section provides a review of the model-based user interface technologies, summarising related information available from the literature. Relevant aspects of fifteen MB-UIDEs, as presented in Table 2.3, are compared and analysed. Details of how specific MB-UIDEs are implemented are not presented here, and nor are specific notations or tools.

2.3.1 Modelled Aspects of User Interfaces

Table 2.4 presents the terms used in the literature to identify the models of the MB-UIDEs according to the framework.

The application model is present in every user interface model. In fact, the MB-UIDE technology appeared initially as successors to user interface management systems (UIMS), where a clear distinction between the user interface and the mainstream application is required. For UML-concerned MB-UIDEs, the application model is the actual model of the application domain rather than a UI view of the application. This is a reason why UML-concerned MB-UIDEs tend to be more integrated with the design of the mainstream application than other MB-UIDEs.

The presentation model, like the application model, is always included in UI models. However, there are MB-UIDEs that do not have an abstract presentation

MB-UIDE	Application model	Task-Dialogue model
ADEPT	problem domain	task model
AME	application model	object-oriented design (OOD)
CTT	domain model	task model
FUSE	problem domain model	task model
HUMANOID	application semantics design	manipulation, sequencing, action side effects
JANUS	problem domain	(none)
ITS	data pool	control specification in dialog
MASTERMIND	application model	task model
MECANO	domain model	user-task model/dialog model
MOBI-D	domain model	user-task model/dialog model
TADEUS	problem domain model	task model/navigation dialogue
TEALLACH	domain model	task model
TRIDENT	application model	task model
Markopoulos	domain model	task model/activity diagram/use cases
WISDOM	information	dialogue

MB-UIDE	Abstract presentation model	Concrete presentation model
ADEPT	abstract user interface model	prototype interface
AME	object-oriented analysis	prototype
CTT	presentation model	concrete user interface
FUSE	logical user interface	user interface
HUMANOID	presentation	presentation
JANUS	(none)	user interface
ITS	frame specification in dialog	style specification
MASTERMIND	(none)	presentation model
MECANO	(none)	presentation model
MOBI-D	presentation model	presentation model
TADEUS	processing dialogue	processing dialogue
TEALLACH	presentation model	presentation model
TRIDENT	(not surveyed)	presentation model
Markopoulos	abstract interaction model	detailed interaction model
WISDOM	(none)	presentation

Table 2.4: MB-UIDE's component models.

model, such as MASTERMIND and MECANO. In other MB-UIDEs such as HUMANOID, TADEUS and Teallach, the distinction between the abstract and concrete presentation models is not clear. In the last case, designers normally have the flexibility to gradually refine the presentation description from an abstract model to a concrete model.

Finally, UIMs also include the use of a task-dialogue model to describe the possible interactions between users and applications using the presentation and application models. Some MB-UIDEs describe these interactions at a dialogue-level, such as HUMANOID, MASTERMIND and ITS. Other MB-UIDEs, especially those developed after ADEPT, describe the interactions at a task-level, which is more abstract than the dialogue-level. However, there are MB-UIDEs such as MECANO, TADEUS and Markopoulos' approach that describe the possible interactions at both dialogue and task levels.

2.3.2 Constructs of User Interface Models

Having identified the models, we need to identify the model constructs. As we did for models, Table 2.5 presents the model constructs using the terminology available in the literature for the specific proposals. The column *construct* refers to the abbreviation for constructs introduced in the framework (Table 2.2). Constructs not present in Table 2.5 are not used in the specific system, or at least were not identified in the literature.

2.3.3 Notations Utilised in User Interface Models

While Section 2.3.1 has indicated what models are present in different proposals, the semantics of the individual models in different contexts has not yet been touched on. Table 2.6 shows the several different notations used by the models of different proposals.

We notice in Table 2.6 that there are UIs entirely described by models using a single notation. In general, these notations have been developed specifically for the MB-UIDE. They can be completely new as in ITS's style rules [144, 145], or they can be extensions of other notations, as in MASTERMIND's MDL, which is an extension of CORBA IDL [128]. The use of a single notation can be useful to describe how the models collaborate with each other. However, especially due to the requirement of graphical notations, UI models tend to use different notations. For example, JANUS, TADEUS, TRIDENT, Teallach and Adept models use more than one notation. It is not feasible to provide a categorisation of these UIMs in terms of their notations here because they tend to be specific to each proposal. For instance, there are many MB-UIDEs that use a hierarchical task notation to model their task-dialogue models, however, the notation may not be precisely formalised, as in Teallach.

The use of standard notations is a tendency guiding the development of UML-concerned MB-UIDEs such as CTT and Markopoulos' approach, which are trying to integrate task models with UML. This tendency to use standard notations can also can be observed in MB-UIDEs of the second generation. For instance, MASTERMIND's notation is based on CORBA IDL, and AME and TADEUS apply OMT [121] in some of their component models, since these are notations available for describing other parts of the application. In fact, OMT can be used during the analysis and design of the mainstream application, and CORBA IDL

MB-UIDE	Construct	Name
Adept	TASK	task
	GOAL	goal
	SEQ	ordering operator + sequencing
	AIO	user interface object
	CIO	UIO
AME	CLASS	OOA class
	ATTR	slot/OOA attributes
	OPER	OOA operation
	RELAT	relation type
	ACTION	behaviour
	AIO	AIO
	WINDOW	OOD class
	CIO	CIO
	LAY	layout-method
CTT	CLASS	class
	RELAT	association, generalisation
	TASK	task
	ACTION	basic task
	SEQ	temporal-relationship
	AIO	presentation object
	CIO	widget
Humanoid	CLASS	object type
	ATTR	slot
	OPER	command
	TASK	data flow constraints
	GOAL	goal
	ACT	behaviour
	SEQ	guard slots' constraints, triggers
	PRE	sequential pre-condition
	POST	action side-effect
	AIO	template
	WINDOW	display
	CIO	display, interaction technique
	LAY	layout
Janus	CLASS	class
	ATTR	attribute
	OPER	operation
	RELAT	association, aggregation
	CIO	interaction object
	WINDOW	dialog widow (UIView)
ITS	CLASS	data table
	ATTR	field
	VIEW	frame
	AIO	dialog object
	EVENT	event
	ACT	action
	WINDOW	root unit
	CIO	unit
	LAY	style attribute
Mastermind	CLASS	interface
	ATTR	attribute
	OPER	method
	TASK	task
	GOAL	goal
	SEQU	connection type
	WINDOW	presentation
	CIO	presentation part
	LAY	guides, grids, conditionals
MOBI-D	CLASS	object
	ATTR	attribute
	RELAT	relationship
	TASK	task
	ACTION	action
	SEQ	subtask ordering
	AIO	presentation element
	WINDOW	window
	CIO	widget
	LAY	presentation attribute
Teallach	CLASS	class
	ATTR	attribute
	OPER	operation
	TASK	task
	SEQ	task temporal relation
	VIEW	free container
	AIO	AIO
	WINDOW	window
	CIO	CIO
TRIDENT	CLASS	entity
	RELAT	relationship
	TASK	dialog object
	GOAL	goal
	SEQ	link
	VIEW	presentation unit
	AIO	abstract interaction object
	CIO	concrete interaction object
	LAY	placement of CIOs
Markopoulos	TASK	task activity
	ACTION	elementary task activity
	SEQ	temporal ordering
	AIO	interactive object
WISDOM	CLASS	entity class
	TASK	task class
	SEQ	temporal dependency
	CIO	interaction space class
	LAY	user interface style

Table 2.5: MB-UIDE's constructs.

MB-UIDE	Notation	Models
ADEPT	task knowledge structures (TKS) [65] LOTOS [16] Communicating Sequential Process (CSP) [57]	TDM TDM TDM, APM
AME	OOA/OOD [27] OMT [121]	AM AM
CTT	CTT [101] UML [99]	TDM,APM, CPM AM
FUSE	algebraic specification [147] HTA [69] Hierarchic Interaction graph Template (HTI)	AM TDM, UM APM, CPM
HUMANOID	uses a single notation which was not specified	all models
JANUS	JANUS Definition Language (extended CORBA IDL and ODMG ODL)	AM
ITS	Style rule [144, 145]	all models
MASTERMIND	MDL [131] (extended CORBA IDL [128])	all models
MECANO	MIMIC [111] (extended C++)	all models
MOBI-D	MIMIC (see MECANO's notation)	all models
TADEUS	specialised HTA OMT [121] Dialogue Graph (specialised Petri net)	TDM AM, UM TDM
TEALLACH	hierarchical tree with state objects hierarchical tree	TDM AM, APM, CPM
TRIDENT	Entity-Relationship-Attribute (ERA) Activity Chaining Graph (ACG)	AM TDM, APM, CPM
Markopoulos	CTT UML UAN [54, 56]	TDM TDM, AM TDM
WISDOM	UML	AM, TDM, CPM

Table 2.6: Model notations.

can be used during the implementation of the mainstream application.

A comprehensive explanation of the semantics of these notations is outside of the scope of this survey. The references required to find out more about these notations are provided in Table 2.6.

2.3.4 Integration of User Interface Models

Models are integrated, although it is not unusual for the literature to be unclear on the precise nature of the integration. Indeed, Puerta and Eisenstein [114] said that there is a lack of understanding of UIM integration, denoting this problem as *the mapping problem*.

One strategy to finding out how these models are integrated is through the compilation of the relationships of constructs in different component models. Table 2.7 shows some of those inter-model relationships, relating the relationship constructs with their multiplicity. The multiplicity between brackets is described in UML notation [18]. Additionally, Figure 2.1 shows graphically how the models are related to each other in the MB-UIDEs.

MB-UIDE	Inter-model relationship	
	Construct	Construct
ADEPT	ACTION (1)	AIO (*)
AME	CLASS (1)	AIO (1..*)
	CLASS (1)	WINDOW (0..1)
	ATTR (1)	AIO (1)
	WINDOW (1)	AIO (1..*)
	ACTION (1)	AIO (1)
CTT	TASK (1)	CLASS (0..*)
	CLASS (1)	AIO (0..*)
HUMANOID	CLASS (1)	CIO (1)
	AIO (1)	CIO (1)
JANUS	WINDOW (1)	CLASS (*)
	AIO (1)	ATTR (1)
ITS	VIEW (1)	ATTR (*)
	CLASS (1)	AIO (*)
	AIO (1)	CIO (1..*)
MASTERMIND	TASK (1)	OPER (0..1)
	TASK (1)	CIO (0..1)
	root TASK (1)	WINDOW (1)
MECANO	WINDOW (1)	CLASS (1)
	AIO (1)	ATTR (1)
MOBI-D	CLASS (*)	TASK (*)
	ATTR (1)	AIO (1)
	TASK (1..*)	WINDOW (1)
TEALLACH	TASK (1)	CLASS (0..*)
	TASK (1)	AIO (0..*)
	TASK (1)	VIEW (0..1)
	WINDOW (1)	AIO (0..*)
	AIO (1)	CIO (1..*)
TRIDENT	TASK (1)	VIEW (1)
	WINDOW (1..*)	VIEW (1)
	AIO (1)	CIO (0..*)
Markopoulos	ACTION (1)	CLASS (0..*)
	TASK (1)	AIO (1)
WISDOM	CLASS (1..*)	TASK (1)
	TASK (1..*)	CIO (1)

Table 2.7: Discrete representation of the inter-model relationships.

The presentation model can be considered as a set composed of the APM, the CPM, and the relationships between the APM and CPM. Our strategy for analysing Table 2.7 is based on the identification of how AMs relate to presentation models. There are two approaches to relating AMs and presentation models. The first approach, mainly used by MB-UIDEs of the first generation, is creating direct relationships between the two models, such as in HUMANOID, JANUS, ITS and MECANO. The second approach, mainly used by MB-UIDEs of the second generation, is using the TDM. In this case, there are relationships between the AM and the TDM, and between the TDM and the presentation model, such as in CTT, MASTERMIND, Teallach and Markopoulos' approach.

In AME and ADEPT, for instance, there are relationships between the APM and the TDM, but these relationships do not provide a link with the AM that is

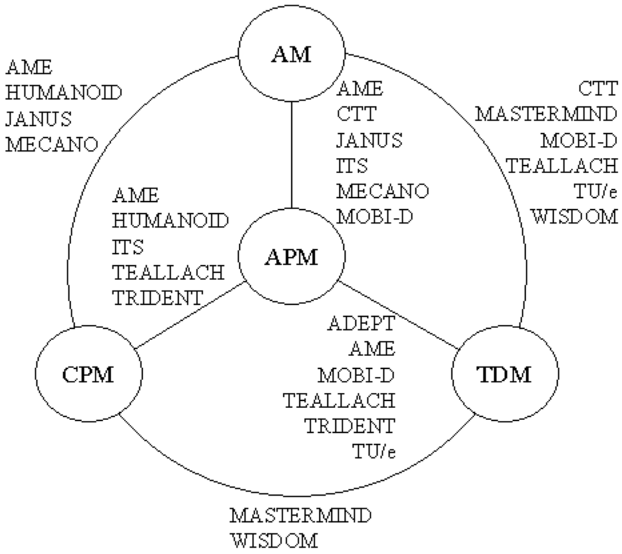


Figure 2.1: Graphical representation of the inter-model relationships.

directly linked with the APM. In this case, the link is more between the AM and the TDM than between the AM and the presentation model.

2.4 Summary

MB-UIDEs seek to provide a setting within which a collection of complementary models can be used as a description of UI functionalities. The survey in this chapter has compared the models and tools provided by 15 MB-UIDEs. A comparative framework composed of the elements in Tables 2.1 and 2.2 was used to present the user interface models of the surveyed MB-UIDEs. Table 2.1 presents four categories of models often used to describe relevant aspects of user interfaces: *application models*, *task-dialogue models*, *abstract presentation models* and *concrete presentation models*. Table 2.2 presents 15 categories of constructs used to compose the models in Table 2.1. This framework is important for relating the MB-UIDE and techniques developed in the work described in this thesis to the research work on MB-UIDEs available in the literature.

The MB-UIDE technology is just now becoming stable enough to be commercialised as products e.g. Systemator [4]. Indeed, this is the result of practical experiences with this technology, e.g. ITS was used by IBM to produce the UI of the visitor information system of EXPO'92 [144, 145], and FUSE has been used by Siemens to simulate an ISDN telephone.

However, there are many aspects of MB-UIDE technology that must be studied in order to increase the acceptance of MB-UIDEs at the level of other specialised UI development tools [93].

- *Standard notations for UIMs.* The use of a standard notation may be useful for describing different UIMs using a common set of constructs. In fact, these constructs may facilitate the comparison of UIMs and their MB-UIDEs. For instance, the use of results achieved in one MB-UIDE by another MB-UIDE may be difficult these days since they are based on several notations, as presented in Table 2.6.

- *Mapping between models.* The aspects of UIs that it is relevant to model in UIMs are well-understood. In fact, most of the surveyed MB-UIDEs can describe a similar set of UI aspects, as observed in Table 2.4. However, there is less agreement on how best to model the relationships between the constructs of the models used to describe UIs, as observed in Table 2.7. The mapping between models in the context of UML is discussed in Chapters 4 and 5.

- *UIM post-editing problem.* Automatically generated drafts of UI designs may be manually refined in order to generate final designs. However, manual refinements to generated designs are lost when developers regenerate other draft designs. Therefore, it is an open issue how best to cope with post-editing refinements. This is an aspect of MB-UIDEs not addressed in this thesis.

The use of UML for modelling UI aspects described in the framework presented in this chapter is discussed in the next chapter.

Chapter 3

User Interface Modelling with UML

This chapter presents a description of a comprehensive UI modelling case study using UML. This case study has the purpose of identifying:

1. common UI modelling difficulties when using UML;
2. a set of UML constructs and diagrams that may be used by application developers to design UIs.

From 1 we can identify some aspects of UIs that are not covered by the UML. From 2 we can identify the aspects of UIs that are covered by the UML. Therefore, the case study produces an insight into the ease with which the UML can be used to model UIs. Moreover, it provides elements that may be used to develop a strategy for extending UML in order to provide better support for user interface design.

3.1 Library System Case Study

A library case study is used to identify user interface modelling problems [49, 110]. A use case diagram in Figure 3.1 shows the actors of the Library System and their use cases. Actors are `Librarians` and `Borrowers`. The actor `LibraryUser` is a generalisation of `Librarian` and `Borrower`.

Librarians use the Library System to manage the loan records, the book catalogue and the user records. Librarians only need to inform to the Library System

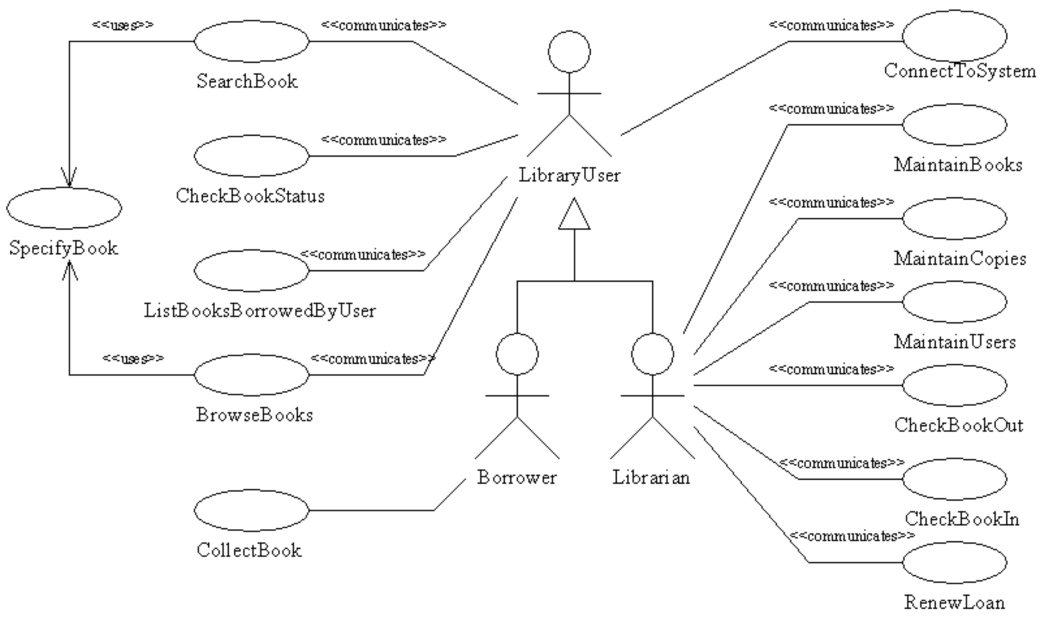


Figure 3.1: The use case diagram.

when books are checked into and checked out of the system to be able to manage loan records. Thus, the use cases `CheckBookOut` and `CheckBookIn`, associated with ≪*actor*≫ `Librarian`, are specified. Librarians can extend expiry dates of loans. Thus, the use case `RenewLoan`, associated with ≪*actor*≫ `Librarian`, is specified. Librarians need to add, update and remove book records and library user records from the Library System. Thus, the use cases `MaintainBooks` and `MaintainUsers` are specified, associated with ≪*actor*≫ `Librarian`.

`LibraryUsers` can connect to the system, list the books borrowed by a library user, check the availability of a book, browse the book catalogue without specifying any condition, and search for books by author, title, year or a combination of these. Thus the use cases `ConnectToSystem`, `ListBooksBorrowedByUser`, `CheckBookStatus`, `BrowseBooks` and `SearchBook`, associated with ≪*actor*≫ `LibraryUser`, are specified. `ConnectToSystem` is considered as a use case since a `LibraryUser` can login to the system just to check his/her password.

The use case `CollectBook`, associated with ≪*actor*≫ `Borrower`, is modelled to represent a task performed by `Borrowers`, although it is not implemented in the Library System. For this reason, the use case `CollectBook` does not have a ≪*communicates*≫ stereotype attached to it.

Some use cases have similar features in their behaviour. For instance, `BrowseBooks` and `SearchBook` are both use cases where books can be specified. Thus, a use case called `SpecifyBook` has been created to model this shared behaviour. Unidirectional associations are specified to model the ≪*uses*≫ relationship between `BrowseBooks` and `SpecifyBook`, and between `SearchBook` and `SpecifyBook`.

3.2 Domain Modelling

The class diagram in Figure 3.2 represents the *domain model* of the Library System. The class diagram is composed of ≪*entity*≫ classes that model things or objects that exist on their own right, and ≪*control*≫ classes that perform system behaviour. From a strict interpretation of the term *domain*, the model in Figure 3.2 could be composed of ≪*entity*≫ classes only. However, the specification of ≪*control*≫ classes in the model is relevant to the description of other models of the Library System presented in this thesis. The ≪*entity*≫ and ≪*control*≫ stereotypes used throughout this chapter were introduced by Ivar Jabcobson et al. [66] in their Object-Oriented Software Engineering (OOSE), and incorporated by UML. The ≪*entity*≫ stereotype identifies classes of the domain of the Library System and the ≪*control*≫ stereotype identifies classes that perform system behaviour. The ≪*boundary*≫ stereotype, although not used in Figure 3.2, is also introduced in OOSE and used later in this chapter. The ≪*boundary*≫ stereotype identifies classes that handle the interaction between system users and systems.

`LibraryUser`, `Librarian`, `Borrower`, `Book`, `BookCollection`, `Loan`, `LoanCollection` and `BookCopy` are the ≪*entity*≫ classes of the Library System. The three first ≪*entity*≫ classes correspond to the `LibraryUser`, `Librarian` and `Borrower` actors, respectively. The existence of an instance of `Book` means that the book has an entry in the library catalogue. A `BookCollection` object is a set of `Book` objects which can be empty. To manage its stock, the Library System has a `BookCopy` class that represents copy versions of the books the library has. It is possible, however, that some books in the library catalogue are not in stock, e.g. when newly ordered books have not yet been delivered, or when books are damaged. An instance of `Loan` is created by the process modelled by the `CheckBookOut` use case and destroyed in the process modelled by the `CheckBookIn` use case. A `Loan` object indicates essentially the day a book should

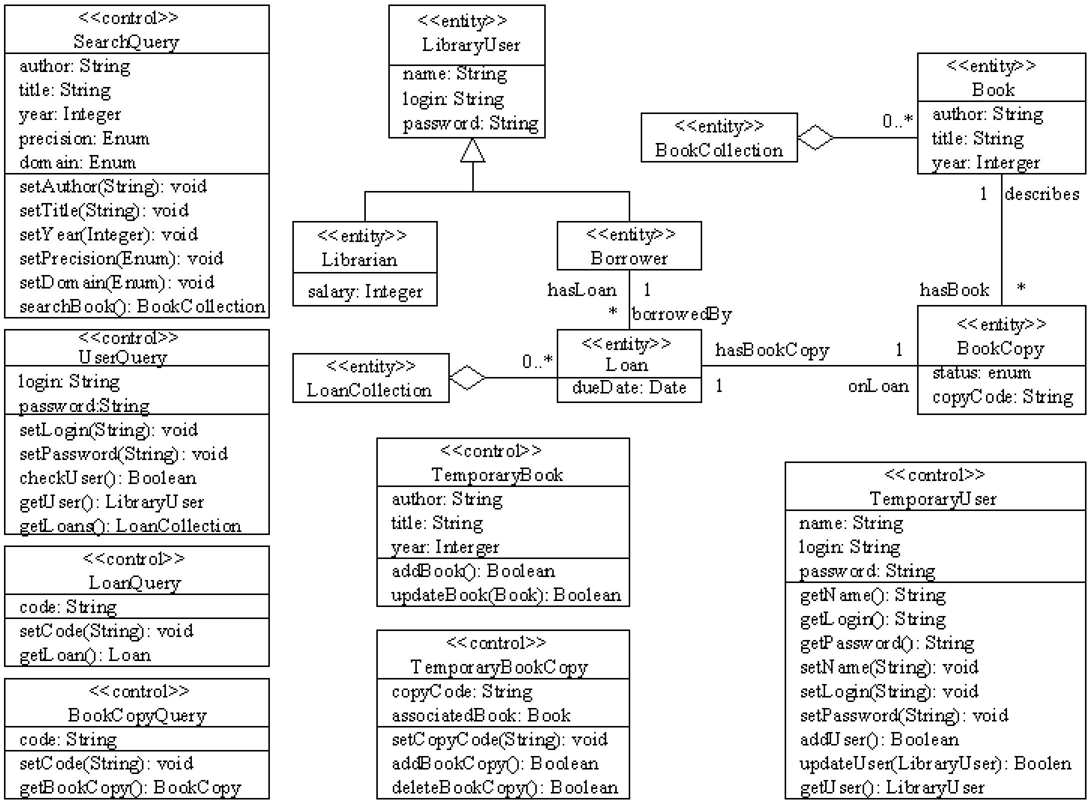


Figure 3.2: The domain model.

be returned. A `LoanCollection` object is a set of `Loan` objects, which can be empty.

`TemporaryBook`, `TemporaryBookCopy`, `TemporaryUser`, `SearchQuery`, `BookCopyQuery`, `LoanQuery` and `UserQuery` are responsible for supporting part of the behaviour elicited by the use cases in Figure 3.1. A `TemporaryBook` object supports the performance of the `MaintainBooks` functionality providing the `addBook()` and `updateBook()` methods to store new `Books` and update stored `Books`, respectively. In a similar way that the `MaintainBooks` functionality is supported by a `TemporaryBook` object, a `TemporaryBookCopy` object supports the performance of the `MaintainCopies` functionality for `BookCopies`, and a `TemporaryUser` object supports the performance of the `MaintainUsers` functionality for `LibraryUsers`. A `SearchQuery` object supports the performance of the `SearchBook` and `BrowseBook` functionalities. A `BookCopyQuery` object supports the performance of the `CheckBookStatus` functionality. A `LoanQuery` object supports the performance of the `CheckBookOut`, `CheckBookIn` and `RenewLoan` functionalities. Finally, a `UserQuery` object supports the performance of the `ConnectToSystem` and `ListBooksBorrowedByUser` functionalities.

The case study description provides the context for the introduction of the UI design using UML.

3.3 Behaviour Modelling

The `SearchBook` use case in Figure 3.1 shows that a library user can search for books. However, library users must be logged in to perform system functions. Using UML terminology, this means that the ≪*actor*≫ `LibraryUser` can only use the `SearchBook` use case if he/she previously used the `ConnectToSystem` use case. In fact, the situation is slightly more complex than that. The fact that a borrower has used `ConnectToSystem` does not mean that s/he has logged into the system since, for instance, the attempt to login could have failed. This difficulty in interpreting Figure 3.1 can arise because use cases were conceptualised for eliciting services (or functionalities), but not for providing control flow information related to tasks.

UI Modelling Difficulty 1 *Use cases do not provide temporal dependency features like pre-conditions and post-conditions often associated with user requirements.*

Temporal dependencies, however, can be specified between activities in activity diagrams. The activity diagram in Figure 3.3 shows how a user can interact with the user interface of the Library System. There, `Connect` is the first activity to be performed. The activity `SelectFunction` is reached if an object `cu` of the `LibraryUser` class is successfully instantiated[1] within the `Connect` activity. The application can be finished by having its control flow diverted to a final state if the object `cu` is not instantiated. Once the `SelectFunction` activity is reached, the `InitiateMainUI` activity responsible for instantiating and making visible the widgets of the `MainUI`, i.e., `qt` of the class `Quit`, `sb` of the class `SearchBook` and `cs` of the class `CheckBookStatus`, is performed. The invocation of the `qt.setActive(true)` and `sb.setActive(true)` operators allows users to interact with the `qt` and `sb` objects respectively. Thus, finishing the execution of the `InitiateMainUI`, the user can finish an interaction with the application by

[1] The explanation of how the object `cu` can be instantiated is presented in Section 3.3.1.

triggering the `qt.invokeAction()` operation. The user, however, may prefer to search for a book by triggering `sb.invokeActon()`, which results in the execution of the `SearchBook` activity.

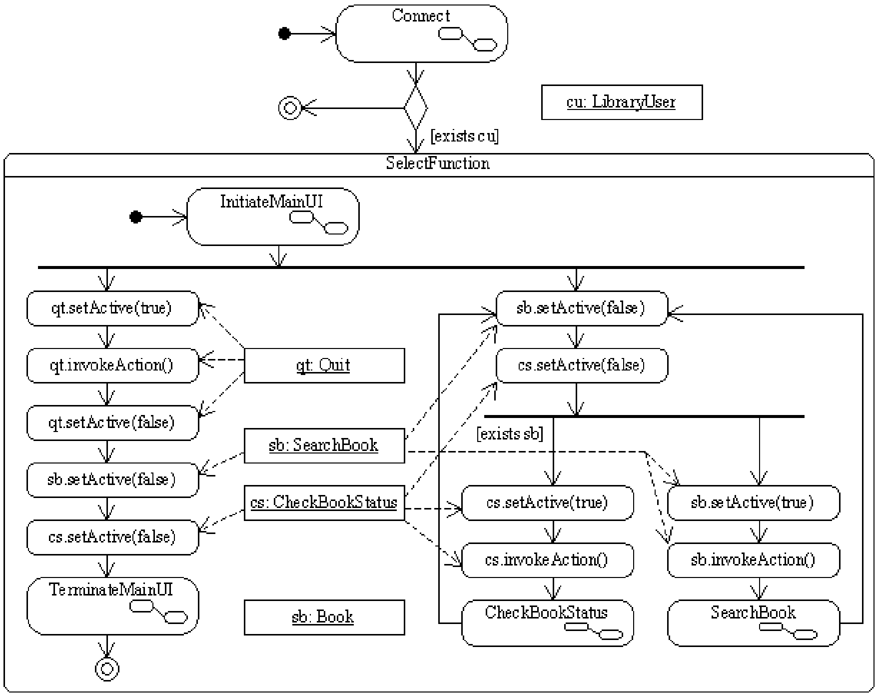


Figure 3.3: A partial top-level activity diagram of the Library System.

The activity diagram in Figure 3.3 is a partial view of the Library System, since it only specifies the `SearchBook` and `CheckBookStatus` activities corresponding to the `SearchBook` and `CheckBookStatus` use cases. The specification of a complete top-level activity diagram of the Library System can be achieved by including new activities and widgets in the model for modelling the other use cases in Figure 3.1, in the same way as the activities and widgets of the for the `SearchBook` and `CheckBookStatus` use cases are organised in Figure 3.3. In fact, a set of activity diagrams containing the diagram in Figure 3.3 could describe all possible user interactions, since it could describe all the possible ways the application control flow can navigate between activities where users may interact with the application. For example, transitions in activity diagrams are inter-object transitions, such as those transitions between interaction and domain objects that can describe interaction behaviours. However, initial states (or pseudo-states of kind `initial`) used to indicate the entry-points of activity diagrams do not identify application entry-points. For example, it may be impossible to assure that the initial state in Figure 3.3 is an entry-point of the Library System. Thus, the process of identifying in which activity diagram interactions start is unclear.

The identification of entry points is not a problem in activity diagrams defined in the scope of structural elements such as classes and packages. In fact, initial states identify the entry points of these activity diagrams. However, the identification of entry points is a problem in activity diagrams defined in the scope of entire systems, whether interactive or not. Although the problem of identifying entry points for this category of activity diagrams is easily solved by the introduction of a new construct, such entry points cannot be identified at all in standard UML models [99].

UI Modelling Difficulty 2 *UML does not specify any construct for modelling application entry points.*

Applying activity diagrams to control user interface navigation resembles traditional Hierarchical Task Analysis (HTA) [69, 73, 28], which is widely used to describe user task models. However, activities and tasks are not exactly the same thing although they have similar characteristics. For instance, the name of the `ConnectToSystem` use case suggests it represents a system's functionality where users can try to log into the system. The `Connect` activity, however, provides a more detailed specification for the functionality represented by the `ConnectToSystem` use case. The specification of the `Connect` activity in Figure 3.3 says that the activity is the first bit of functionality to be performed by the system. Furthermore, the specification says that if no `cu` object is instantiated at the end of the `Connect` activity then the application finishes. As a consequence of these observations, the `Connect` functionality is completed before the start of any other functionality in the system. Moreover, from what is presented in this section we can see that a combination of use cases and activities can allow designers to use the simplicity of use case diagrams to elicit system functionalities and top-level goals, and to use the expressiveness of activities to refine the specification of the elicited functionalities.

3.3.1 Activity Refinement and Object Flows

Activities can be decomposed to specify more details about the behaviour of systems. For instance, the `Connect` activity initially specified in Figure 3.3 can be decomposed into less abstract activities as presented in Figure 3.4. There, the decomposed `Connect` activity shows that the `cu` object which was not related to any activity as specified in Figure 3.3 can be, in fact, instantiated as a result of performing the `new UserQuery` activity. Moreover, the `InitiateConnectUI` and `TerminateConnectUI` activities in Figure 3.4 are still too abstract to explain how widgets of the `ConnectUI` user interface can be instantiated, made visible, made invisible and destroyed. Thus, the `Connect` activity can be further detailed by decomposing the `InitiateConnectUI` activity, as presented in Figure 3.5(a), and `TerminateConnectUI`, as presented in Figure 3.5(b). In fact, a similar modelling strategy for instantiating, making visible and invisible, and destroying the widgets of the `ConnectUI` user interface is implemented within the `InitiateMainUI` and `TerminateMainUI` activities in Figure 3.3 for the widgets of the `MainUI` user interface.

Returning to Figure 3.4, the `uq` object of class `UserQuery` is instantiated by `new UserQuery`, which is an *action state* performing a create action. Action states are activities that do not require further decomposition since they are responsible for the execution of a action that can, for example, be the invocation of an object operation or the raising of an event. Following the control flow in the diagram, the `cn.invokeAction()` and `ok.invokeAction()` operations can be triggered once the `cn` and `ok` objects are activated. Thus, the triggering of these operations results in the cancelling or confirming of an attempt to connect to the system, respectively. Concurrently, users can provide their login identification and password by interacting with the `pt` and `lt` objects. These interactions are performed during the execution of the `uq.setPassword(pt.getValue())` and `uq.setLogin(lt.getValue())` action states. The confirmation of an attempt to connect into the system results in the invocation of the `uq.CheckUser()` action state, which may instantiate a valid `cu` object of the `LibrarySystem`.

Many tasks require information from the domain model as well as information provided by the users [47]. Object flows are used for indicating which objects are related to each activity, and if the objects are generated or used by the related activities. Thus, object flows specify how information from the domain can be used by activities. However, object flows do not describe the behaviour of related objects within their associated activities. For example, in Figure 3.4 the `uq` object of class `UserQuery` instantiated as a result of the execution of the `new UserQuery` action state is used by the `uq.setPassword(pt.getValue())`,

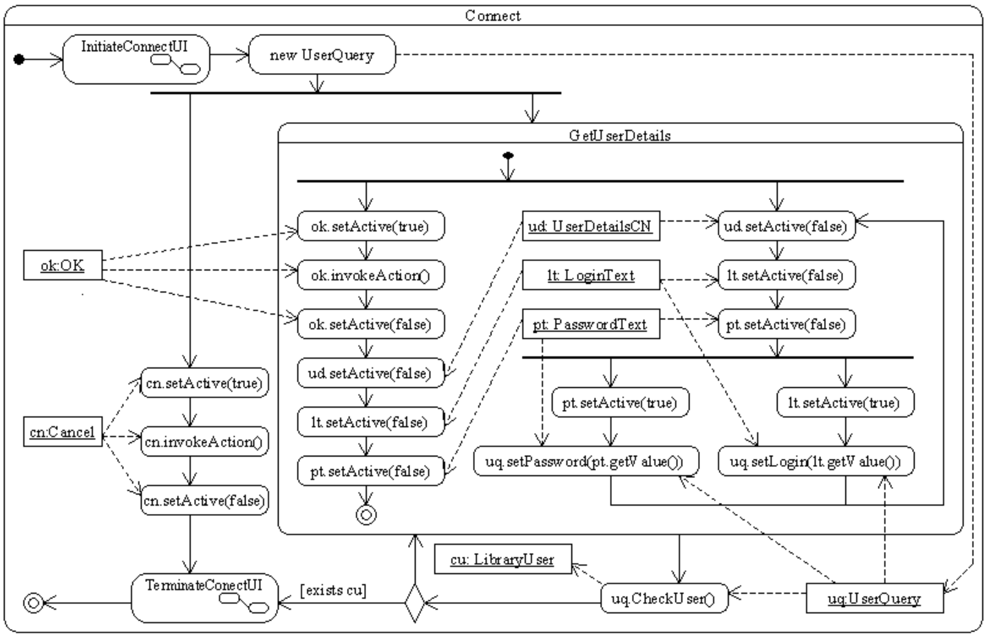


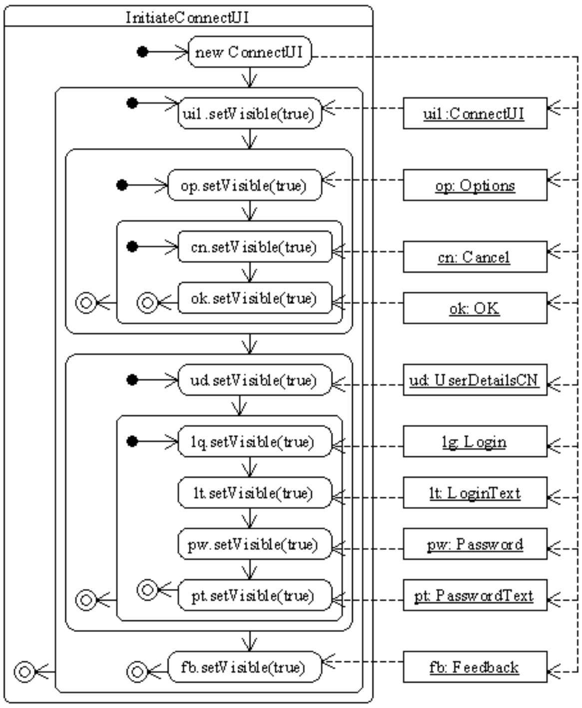
Figure 3.4: A decomposition of the `Connect` activity of Figure 3.3.

`uq.setLogin(lt.getValue())` and `uq.CheckUser()` action states later in the activity diagram. These action states explain how the `uq` object can be used. In fact, they specify when and how the operations of `uq` can be performed. Thus, a complete decomposition of activities into action states may be required to achieve such object behaviour description. However, the following problem is identified.

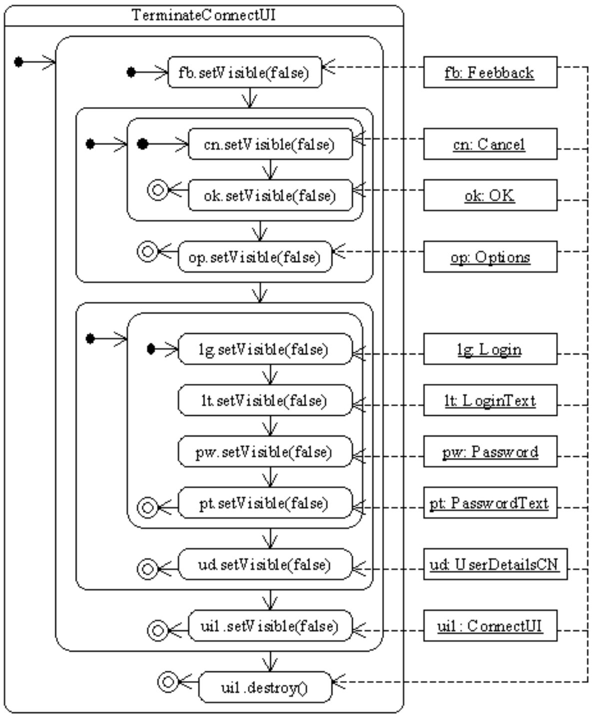
UI Modelling Difficulty 3 *The complete decompositions required to explain object behaviours tend to overload activity diagrams by increasing the number of graphical elements composing the diagrams, and consequently making the activity diagrams more complex.*

3.3.2 Interactive Application Behaviour

Activity diagram constructs for modelling transitions are powerful since they can be combined in several ways, producing many different compound transitions. 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).



(a)



(b)

Figure 3.5: The instantiation (a) and destruction (b) of `ConnectUI` and its components.

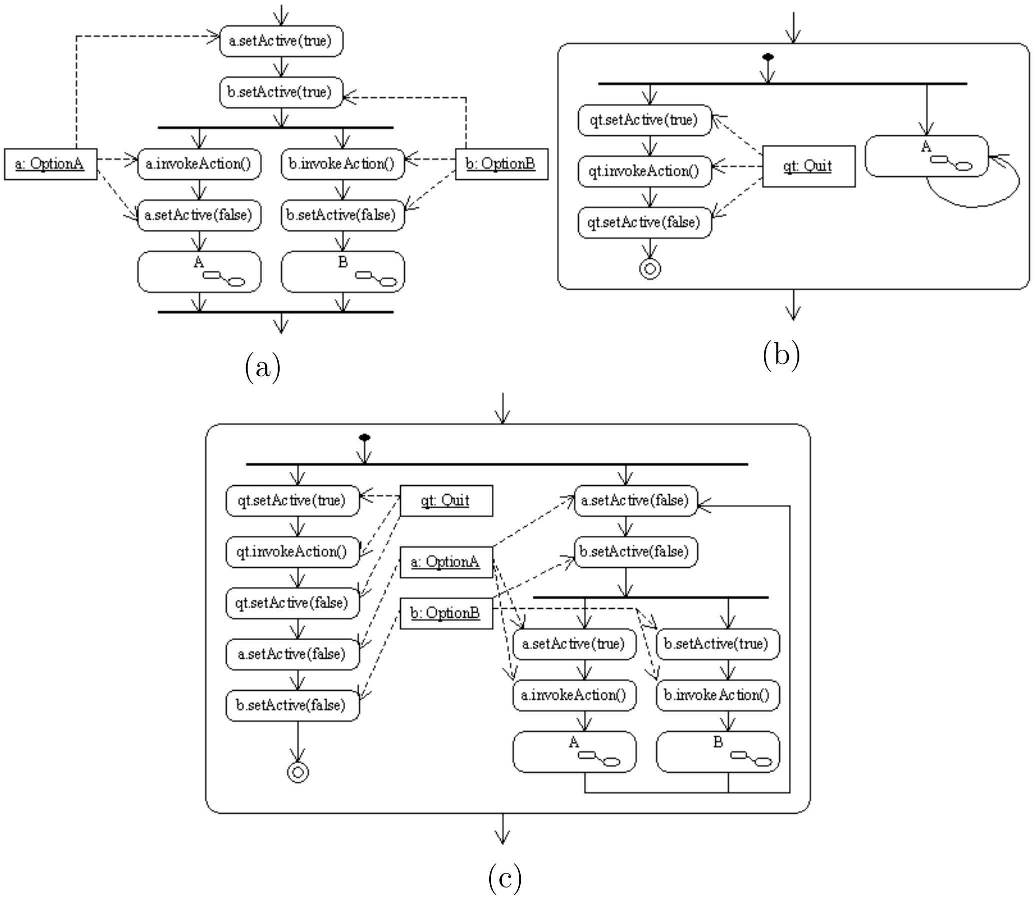


Figure 3.6: The UML modelling of three common interaction application behaviours. An `order independent` behaviour is modelled in (a). A `repeatable` behaviour is modelled in (b). An `optional` behaviour is modelled in (c).

The following behaviours are common interactive application behaviours [47].

- The `order independent` behaviour, as presented in Figure 3.6(a). There, activities `A` and `B` are called *selectable activities* since they can be activated in either order on demand by users triggering the `a.invokeAction()` and `b.invokeAction()` operations when 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 choosing the execution order of selectable activities since they are responsible for triggering `a.invokeAction()` and `b.invokeAction()`. An order independent behaviour should be composed of two or more selectable activities.

- The `repeatable` behaviour, as presented in Figure 3.6(b). A repeatable behaviour should have only one associated activity. `A` is activity associated with the repeatable behaviour in Figure 3.6(b). The looping created by the transition leaving activity `A` and going to activity `A` should not be considered a livelock, as the library user is able to leave activity `A` by triggering the `a.invokeAction()` operation.

- The `optional` behaviour, as presented in Figure 3.6(c). There, users can execute any selectable activity any number of times, including none, by triggering `a.invokeAction()` and `b.invokeAction()`. In this case, users should explicitly specify when they are finishing the activity by triggering `qt.invokeAction()`. Like the order independent behaviour, the optional behaviour should be composed of one or more selectable activities.

These behaviours are common in interactive systems. For instance, both activity diagrams in Figures 3.3 and 3.4 implement an order independent behaviour. Despite the fact that it is possible to model these behaviours in UML, a problem can be identified from them.

UI Modelling Difficulty 4 *Activity diagram constructs can be held to be rather low-level for modelling order independent, optional and repeatable behaviours, leading to complex models.*

3.4 Abstract Presentation Modelling

The need to model UI presentation arises naturally while modelling interactive applications. Even for very simple scenarios, the modelling of part of the UI presentation is essential. For instance, the `ok`, `cn`, `ud`, `lt` and `pt` objects in Figure 3.4 are elements of a UI presentation. At this stage we do not need a detailed model of the UI presentation, but only to know what kinds of components compose the UI, how many components there are, and how they may be grouped. We also need to know which abstract operations these UI elements should have.

The modelling of the `ConnectUI` user interface used by the Library System to interact with users trying to log into it can be used to exemplify the use of an *abstract presentation model*. Further, it can lead to an explanation of how the `ok`, `cn`, `ud`, `lt` and `pt` objects in Figure 3.4 are used to compose the `ConnectUI` presentation.

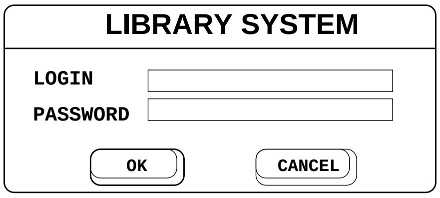


Figure 3.7: The display of the `ConnectUI`.

The `ConnectUI` object presents to the user a connection user interface requesting a login name and a password. This user interface can be something like the form shown in Figure 3.7, which is not a UML diagram. However, it would be good to have a notation that allows designers to specify widgets and their layout or to abstract over such details, should they choose to do so.

A description of how UML constructs can be used to model UI presentations is presented in the next section.

3.4.1 Abstract Presentation Pattern

The abstract presentation pattern (APP) in Figure 3.8 provides a generic description of classes and their relationships used to represent abstract widgets. There the APP has a top-level container, the `FreeContainer`, that can have many components from the classes `ActionInvoker`, `PrimitiveInteractionClass`, and `Container`. A `Container` defines an area in a presentation device, e.g. the screen. The visual presentation of the `Container` itself and the widgets contained by it are restricted to this area. A `FreeContainer` is a `Container` that cannot be contained by any other `Container`, e.g., a top-level frame (or window). In the model, `Containers` provide a grouping mechanism for the structural elements of the UI presentation. All such structural elements are represented by the abstract component `InteractionClass`. The `ActionInvoker` sub-category of `InteractionClass` represents those components that can receive information from users in the form of events, such as buttons. The `PrimitiveInteractionClass` sub-category of `InteractionClass` can be further specialised into `Displayer`, `Inputter` and `Editor`.

- The `Displayer` category represents those components that can present information to users, such as labels.

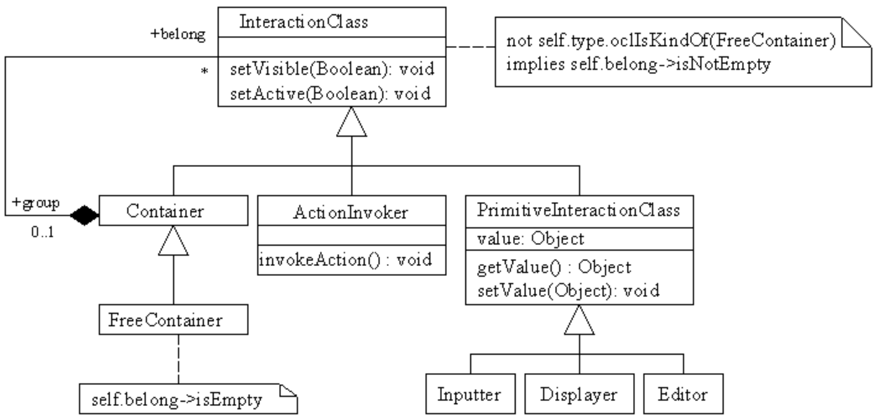


Figure 3.8: The abstract presentation pattern.

- The `Inputter` category represents those components that can receive information from users, such as text fields.
- The `Editor` category represents those components that have the properties of both `Displayers` and `Inputters`, such as combo boxes and selectable lists.

Five operations are defined in the APP: `setVisible()`, `setActive()`, `getData()`, `setData()` and `invokeAction()`. These are the abstract operations of UI presentation elements that should be implemented in some way by widgets.

- `setVisible()` has a boolean parameter. Invoked with a `true` value the method makes the `InteractionClass` visible for users. Invoked with a `false` value the method makes the `InteractionClass` invisible for users.
- `setActive()` has a boolean parameter. Invoked with a `true` value the method enables the `InteractionClass` to interact with users by updating its visual appearance, if it has one, and updating its state, if there is any. Invoked with a `false` value, the state of the `InteractionClass` is preserved unchanged from the time the method is invoked, preventing the `InteractionClass` from interacting with users. The visual appearance of the `InteractionClass`, if there is any, may change to notify users about its disabled status. This method has no effect if the `InteractionClass` is not visible.

- `setValue()` updates the state of the `PrimitiveInteractionClass`. As a result, the visual appearance of the `PrimitiveInteractionClass` is updated according to the updated state, if it depends on the state.

- `getValue()` returns the state of the `PrimitiveInteractionClass` that could be updated, collecting information provided by a user during an interaction.

- `invokeAction()` puts the `ActionInvoker` in a state waiting for a specific user event assigned to the `ActionInvoker`. The method finishes when the `ActionInvoker` senses the event during an interaction with the user.

3.4.2 Using the Abstract Presentation Pattern

The APP is the framework used to describe conceptual user interface presentations. A class diagram extending the APP provides a conceptual description of a user interface presentation. Thus, the `ConnectUI` can be conceptually described by the abstract presentation model shown in Figure 3.9. There the APP is the top of the hierarchy of classes. The other classes are specialisations of the APP's classes. For example, `ConnectUI` is a `FreeContainer`, `Options` is a `Container` and `OK` is an `ActionInvoker`. The specification of which `InteractionClass` is contained by each `Container` is explicitly specified by the compositions between the subclasses of the APP in Figure 3.9. Thus, it can be observed that `ConnectUI` contains `Options`, `UserDetailsCN` and `Feedback`, and that `Options` contains `Cancel` and `OK`. Although the compositions between interaction classes in Figure 3.9 are correct, it is difficult to see that they are actually representing the containment among interaction classes. Thus, it is difficult to group interaction classes in UML class diagrams, one of the essential tasks for modelling UI presentations.

UI Modelling Difficulty 5 *The notion of containment among classes is not represented graphically in UML class diagrams.*

The selection of interaction classes is another essential task for modelling UI presentations. However, it is usually difficult to perform this task due to the large number of interaction classes with different functionalities provided by graphical environments. In a UML-based environment, the selection of interaction classes

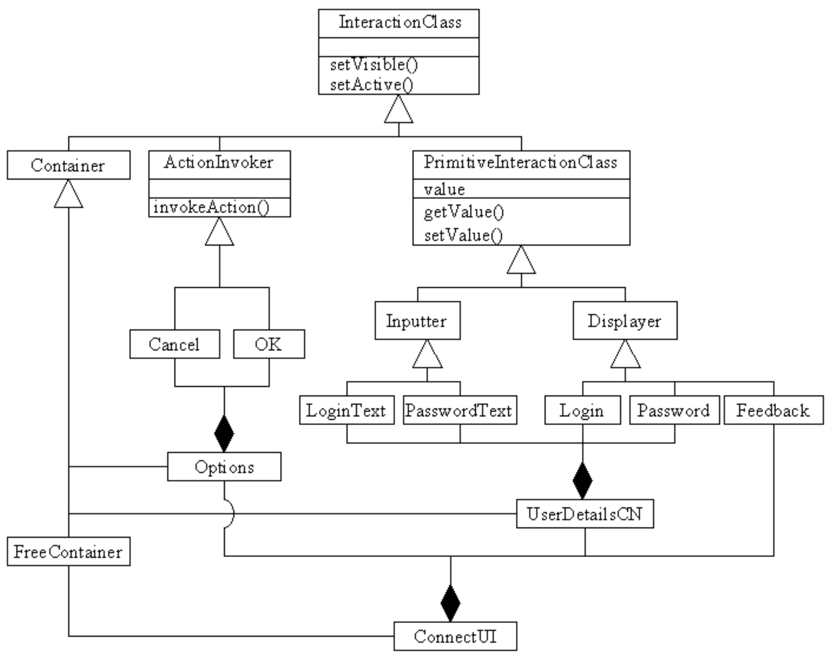


Figure 3.9: The abstract presentation model of the `ConnectUI`.

tends to be even more complex than in UI design environments because of an additional difficulty:

UI Modelling Difficulty 6 *The UML class diagram has a single notation for class, which does not provide a visual distinction between (1) interaction and non-interaction classes, and (2) interaction classes playing different abstract roles in a user interface.*

Unlike in the modelling of the domain, the use of class diagrams for presentation modelling is not a popular choice. Indeed, it is difficult to realise that the class diagram in Figure 3.9 represents the presentation of a user interface, mainly due to modelling difficulties 5 and 6. However, this kind of abstract UI presentation is conceptually equivalent to the abstract presentation in MB-UIDEs [15, 48]. Further, the classes of the APP specify methods similar to those presented by Holub [58].

3.5 Concrete Presentation Modelling

Abstract presentation models do not describe which components compose each ≪*boundary*≫ class. They also do not provide any description of layout. Therefore, *concrete presentation models* are sometimes required during the UI design process.

From a concrete presentation point of view, the abstract presentation model presented in Figure 3.9 is the design pattern specification for the concrete presentation model of `ConnectUI`. In fact, the design pattern approach, as presented in Gamma et al. [42] and incorporated by UML, provides a way to describe how different environments can be accommodated within the diagrams that use elements of the abstract presentation model, e.g., the activity diagram in Figure 3.4. Indeed, concrete presentation models are environment-dependent since they are described in terms of environment classes and components. An environment in our terminology can be classes of an object-oriented programming language, components or both. We will use the Swing components of Java [45] to illustrate how UI classes can be related to *environment* classes. The ≪*cpm*≫ stereotype is used to identify these environment classes.

Figure 3.10 shows the concrete presentation model using the `ConnectUI` abstract presentation model and some Java Swing components. Abstract presentation models have been called *abstract presentation frameworks* (APFs) when used as design patterns. In the `ConnectUI` concrete presentation model, the `ConnectUI` APF is represented using the collaboration symbol of UML. Thus, any name in the `ConnectUI` APF can be used to specify a framework role. Then, the role can be associated with any other element of the same type. For example, an APF class can be associated with a widget and an operation of an APF class can be associated with a widget's operation.

In Figure 3.10, the `ConnectUI` APF is specified with seven different roles: `UserDetailsCN`, `Options`, `FreeContainer`, `PasswordText`, `LoginText`, `ActionInvoker` and `Displayer`. The use of the APF in concrete presentation models provides a clear description of how abstract presentation elements are replaced by concrete presentation elements, respecting the relationships of the abstract presentation model. In fact, the `UserDetailsCN` extending the ≪*cpm*≫`JPanel` is bound to the `UserDetailsCN`, the `Options` extending the ≪*cpm*≫`JPanel` is bound to the `Options`, the ≪*cpm*≫ `JFrame` is bound to the `FreeContainer`, the

≪*cpm*≫ `JPasswordText` is bound to the `PasswordText`, the ≪*cpm*≫ `JTextField` is bound to the `LoginText`, the ≪*cpm*≫ `JTextButton` is bound to the `ActionInvoker`, and the ≪*cpm*≫ `JLabel` is bound to the `Displayer`.

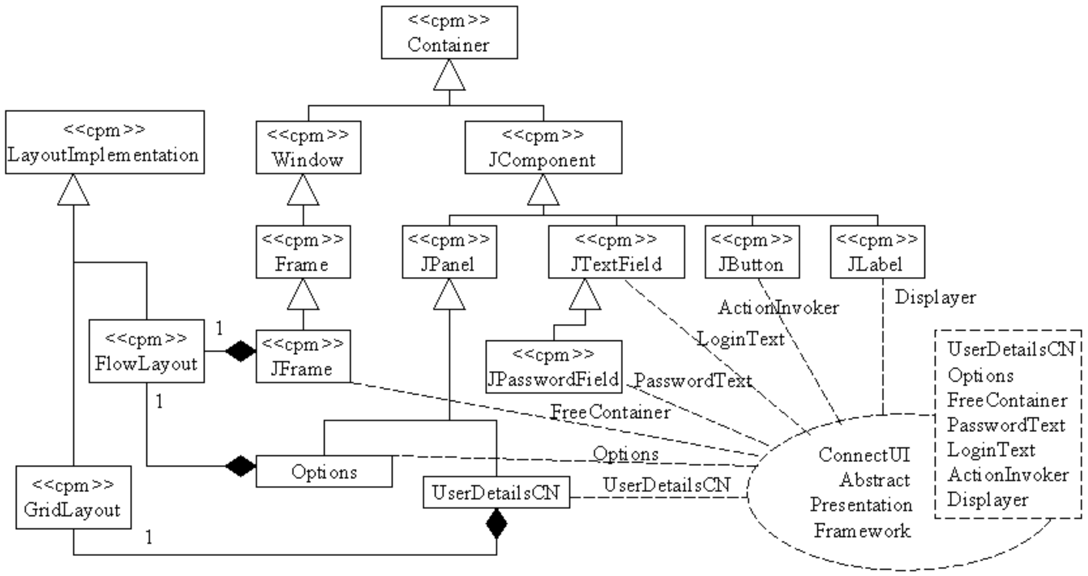


Figure 3.10: The concrete presentation model.

The concrete presentation model, as presented in Figure 3.10, provides what is required to model our case study. Additionally, the concrete presentation model shows how presentation layout can be specified. Every class acting as a `Container` should have an instance of `LayoutImplementation` aggregated to it. Several categories of layout classes can be added to the concrete presentation model by adding them as subclasses of `LayoutImplementation`. Thus, the UI presentation layout is specified by compositions connecting a layout class to each `Container` in the concrete presentation model. The modelling of the UI presentation layout, however, is not entirely explained in Figure 3.10. Indeed, the concrete presentation model relies on the semantics of the environment. In the case of the concrete presentation model of Figure 3.10, Java provides algorithms embedded in methods that work as templates to model layouts for the presentation model of the UI.

Concrete presentation models based on the APFs do not cause large parts of the design to be environment-dependent, providing models with a flexible and well-established relationship between the abstract presentation elements and the component classes. For example, the binding of `Container` classes to ≪*cpm*≫

classes is specified through the `Options` and `UserDetailsCN` classes rather than through the `Container` class itself. The reason for this is that `Options` and `UserDetailsCN` can have different layouts. In fact, the APF role has provided a powerful mechanism for diagrammatically specifying mapping rules between abstract and concrete presentation models.

Concrete presentation models are not discussed any further in this thesis. Additionally, it is assumed that the strategy of using design patterns and frameworks to map abstract presentation elements into concrete presentation elements can be used for producing concrete presentation models from any of the abstract presentation models presented later in this thesis.

3.6 An Integrated View of UI Behaviour and Structure

The necessity of reviewing the behavioural and structural aspects of user interfaces in an integrated way provides an opportunity to exploit the use of sequence diagrams for modelling UIs.

Figure 3.11 shows a sequence diagram for the `ConnectToSystem` use case. According to the use case diagram in Figure 3.1, `ConnectToSystem` is associated with the ≪*actor*≫ `LibraryUser`. Thus, a `LibraryUser` actor initiates this interaction, sending a message to an instance `sys` of `LibrarySystem`, which acts as the whole Library System.

Practically speaking, the `request connection` message can be, for example, a double click on the Library System's icon in a Windows environment. The `sys` object creates the `ui1` object of class `ConnectUI`. The creation of an object is modelled by one object sending a message ≪*create*≫ to the new object. The objects `lt` of class `LoginText`, `pt` of class `PasswordText` and `ok` of class `OK` are instantiated before the execution of the `setVisible()` method by the `sys` object. In fact, the classes of these objects compose the `ConnectUI` abstract presentation model, as described in Figure 3.9. The `uq` object of class `UserQuery` is directly instantiated by the `sys` object.

Interacting with the UI, the user updates the state of the `lt` object which sends the `setLogin(getData())` message to the `uq` object. Further, the user updates the state of the `pt` object which sends the `setPassword(getData())` message to the `uq` object. After providing his/her login and password by interacting with

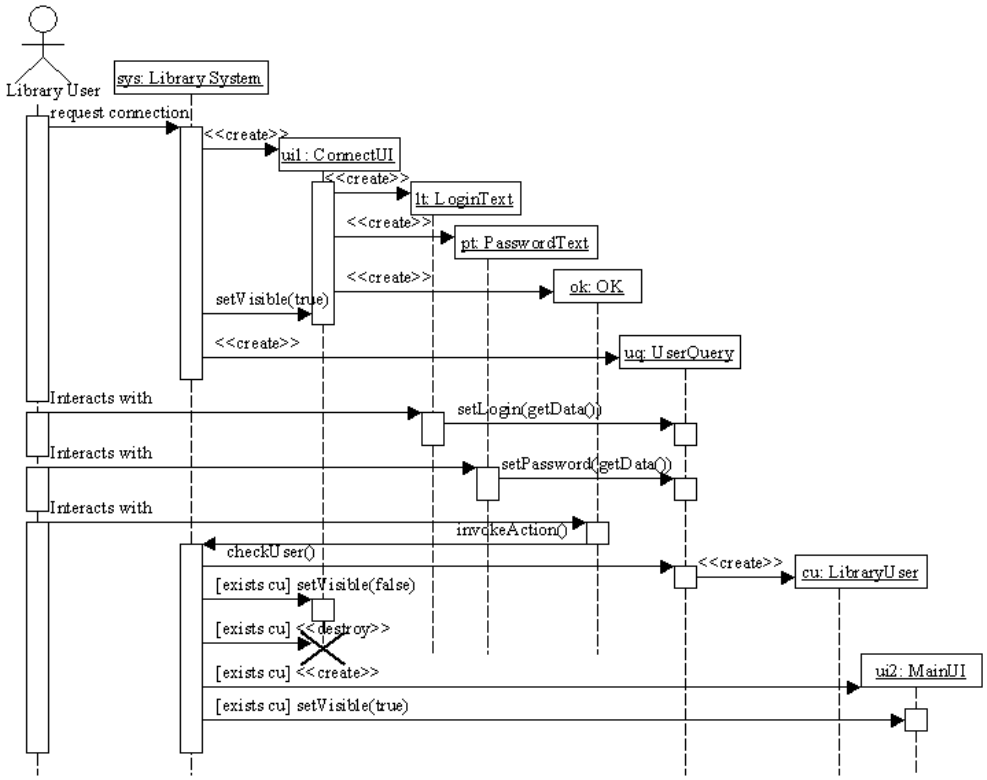


Figure 3.11: A sequence diagram for the `ConnectToSystem` use case.

the `lt` and `pt` objects, the user triggers the `ok` object which notifies this fact to the `sys` object by sending an `invokeAction()` message to it. Thus, the `sys` object sends a `checkUser()` message to the `uq` object. As a result of receiving the `checkUser()` message, the `cu` object of `LibraryUser` is instantiated, if there is a library user in the system's database with the provided login and password. In this case, at the end of the execution of the `checkUser()` method, the `sys` object makes invisible and destroys the `ui1` object, and creates the `ui2` object of class `MainUI` .

The sequence of actions in Figure 3.11 is a possible sequence of actions in Figures 3.3 and 3.4. In fact, sequence diagrams are snapshots of a possible interaction scenario, and consequently their specification is a subset of the specification provided by activity diagrams. For example, the sequence diagram presented is restricted to the scenario where the user successfully logs into the system providing the login before the password. This interaction scenario, however, is one of the many possible scenarios modelled by the `Connect` activity in Figure 3.4.

Moreover, the sequence diagram does not describe when the `lt`, `pt` and `ok` objects are ready to interact with the users as in Figure 3.4. Indeed, the sequence diagram just makes explicit that its sequence of actions is a possible one in the Library System.

UI Modelling Difficulty 7 *Sequence diagrams present possible sequences of actions but without specifying the temporal dependencies between the actions.*

Actions in sequence diagrams, as presented above, and abstract presentation models, as presented in Figure 3.9, are connected by objects of sequence diagrams. Actions in activity diagrams, as presented in Section 3.3, and abstract presentation models are connected by object flows of activity diagrams. Thus, it may be difficult to construct integrated behavioural and structural models of UIs, but it is possible.

3.7 Event Modelling

As described in Booch [18], *events* are "things that happen". Indeed, many things happen when we are using an application: keys and buttons are pressed, the mouse is moved, messages are sent to the network, etc. We call these things that happen *events*. In an object-oriented user interface, inputs and outputs are streams of events [46]. Figure 3.12 shows a general event model where *user actions* and *synchronisation events* are sent to an object-oriented user interface as input events. The application, through its user interface, reacts to these input events by generating output events that are presented as *visual feedback*. Visual feedback can be *normal feedback* or *abnormal feedback*. Abnormal visual feedback, such as error messages, is that associated with difficulties encountered during the enactment of a user's activity.

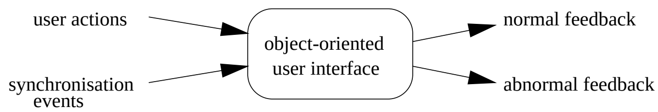


Figure 3.12: The event model.

Abnormal visual feedback can happen with exceptions, and synchronisation events can be generated from system actions. Although involving a significant amount of effort, the modelling of exceptions and synchronisation events using standard UML is a straightforward task. This is illustrated through the modelling of exceptions using UML. Thus, exceptions, as defined by Meyer in [89], are run-time events "that may cause a routine call to fail". Moreover, a routine call fails when it terminates its execution in a state not satisfying the routine's contract. These definitions are complex since they require further definitions such as the *routine's contract* definition, which is itself complex. Despite the formal definition, exceptions here are more akin to those used in object-oriented programming languages such as Java [45] and C++ [132].

In terms of user interfaces, the important aspect of exceptions is that sometimes they are not entirely solved by exception handlers, leading the application to provide visual feedback to users that something is going wrong (or, at least, not going as expected). In fact, once activated, the exception handlers try to solve the problems identified by the exceptions without notifying the users. Unfortunately, exception handlers do not solve every kind of problem. Therefore, the user should be notified of those unsolved exceptions or involved in choosing a solution to the problem.

The problem now is how to model the aspects of the user interface that are related to exception handling.

Structural Aspects of the UI of Exception Handlers

In the application model there are many situations where exceptions and exception handlers can be used. For example, the designer could choose to display an error message somewhere in the `ConnectUI` form due to an exception raised during the execution of a database query.

In UML notation, exceptions are modelled as a stereotyped ≪*send*≫ dependency from a class operation to an exception handler class [18]. Figure 3.13 shows a ≪*send*≫ dependency that links the operation `checkUser()` in the `UserQuery` class with the ≪*exception*≫ `DatabaseFail` class. Moreover, Booch et al. [18] proposes a hierarchy of exception handlers identified by the ≪*exception*≫ stereotype. Usually, uncaught exceptions are sent to higher-level exception handlers in the hierarchy until they are caught by an exception handler or until they reach the top-level handler of the hierarchy. If some exception is not handled by ≪*exception*≫ `DatabaseFail` in Figure 3.13, then it must be handled by ≪*exception*≫ `Exception`.

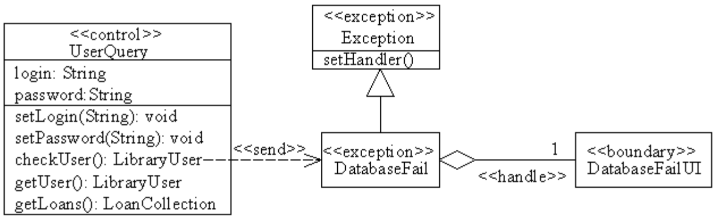


Figure 3.13: The relationship between a UI and an exception handler.

Exceptions can be generated in any class, since classes generally have methods (that are routines), which have contracts that can be broken. Despite the fact that exception handlers can act as ≪*control*≫ classes, they are not modelled exactly as ≪*control*≫ classes. Instead, they can catch exceptions (events) from classes of any category. In this case, ≪*exception*≫ classes are introduced. The operations of these classes can be called from any method of any class, even from methods of other ≪*exception*≫ classes.

One of the roles of ≪*exception*≫ classes is to act as ≪*control*≫ classes to ≪*boundary*≫ classes when exceptions happen. However, there are situations where ≪*exception*≫ classes cannot control a ≪*boundary*≫ class. For instance, if the exception handler requires some decision such as `quit` or `retry` from the user, and the original ≪*boundary*≫ object does not have components to deal with such an interaction, then a new ≪*boundary*≫ object should be created to provide the communication between exception handlers and users.

In terms of the user interface, however, it is important to know how ≪*boundary*≫ classes are related to this hierarchy of exceptions. Objects of ≪*exception*≫ classes can act as objects of ≪*control*≫ classes. Therefore, ≪*boundary*≫ classes can be aggregated to ≪*exception*≫ classes. In Figure 3.13, the ≪*exception*≫ `DatabaseFail` acts as a ≪*control*≫ class, handling the ≪*boundary*≫ `DatabaseFailUI` class. The ≪*handles*≫ stereotype is used to identify the relationship between ≪*boundary*≫ classes and their controllers.

Behavioural Aspects of the UI of Exception Handlers

Exceptions also affect the activity diagram of the user interface since they can modify the flow of control from activity to activity during a user interaction. For instance, the `uq.checkUser()` action state in Figure 3.4 can raise a database exception [23] since a query is performed there. Indeed, the `cu` is a persistent object.

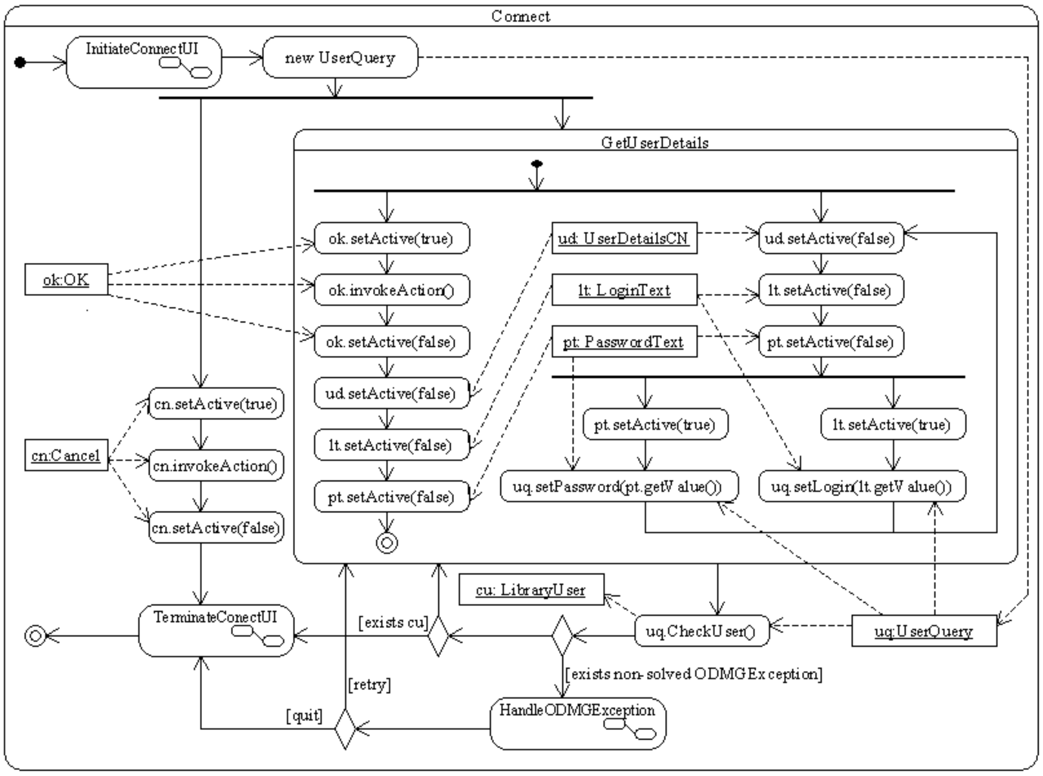


Figure 3.14: Exceptions in activity diagrams.

The modelling of possible modifications to the flow of control of the activity diagram is a straightforward task since UML's activity diagrams provide a branching notation. The outgoing transitions can be re-routed to different activities, depending on boolean guard expressions. Figure 3.14 shows the activity diagram of Figure 3.4 extended to model exception handling. The branch after the action state `uq.checkUser()` re-routes the flow of control to the transition with an `[exists non-solvedODMGExceptions]` guard when an ODMGException is not solved by its handler. Otherwise, the flow of control follows the usual route identified by the unguarded transition.

3.8 Summary

This chapter discussed user interface modelling using a Library System case study. The application system was entirely modelled using the Unified Modeling Language, which has proved to be capable of modelling user interfaces. In fact, UML has a rich set of constructs that is comprehensive enough to model behavioural and structural aspects of form-based user interfaces. However, such UI modelling may not be as straightforward a process as expected and desired, since some modelling difficulties were identified from the case study.

Concerning the behavioral aspects of user interfaces, the case study has demonstrated the difficulty of:

- modelling tasks using use cases, as suggested by UI Modelling Difficulty 1;
- identifying application entry-points, as suggested by UI Modelling Difficulty 2;
- specifying actions that instances of interaction classes can perform when collaborating with other interaction classes and with domain classes, as suggested by UI Modelling Difficulty 3;
- specifying some categories of interactive behaviours, e.g., optional and order independent behaviours, using activity diagrams, as suggested by UI Modelling Difficulty 4;
- specifying temporal dependencies using sequence diagrams, as suggested by UI Modelling Difficulty 7.

Concerning the structural aspects of user interfaces, the case study has demonstrated the difficulty of:

- identifying containment among interaction classes, as suggested by UI Modelling Difficulty 5;
- identifying abstract roles that interaction classes can play in user interfaces, e.g., displaying information to users, receiving information from users, triggering actions, etc., as suggested by UI Modelling Difficulty 6.

Additionally, the case study provides an illustrative example of the use of many UML constructs, in terms of diagrams, for modelling the user interface. The summary of the UML diagrams used is presented in Table 3.1, and the constructs are those used in the diagrams presented throughout the chapter.

There are also some lessons that can be learned from the modelling of the Library System:

User Interface Aspects	UML Resource
Requirements Model	use case diagram
Domain Model	class diagram
Task Model	use case diagram + activity diagram sequence (interaction) diagram
Abstract Presentation Model	class diagram
Concrete Presentation Model	class diagram with design patterns

Table 3.1: Summary of the UML diagrams used to model many aspects of UIs.

- The design of an user interface is a complex process since it requires complete comprehension of the elements that compose the user interface. Indeed, UIs in general have many elements that are not obviously required from the beginning of the design.
- The elements of the user interface have many dependencies among them. Therefore, the design process should consider UI modelling as integral.

There is room for further discussion of how to model user interfaces using UML. Indeed, there will be other ways of representing user interfaces using UML [97]. The study in this chapter presents some alternatives which seem to comprehend the more natural combinations of UML constructs and diagrams to model user interfaces, as identified in [58, 64].

In the next chapter an approach is presented based on the use of the diagrams identified in Table 3.1 to improve support provided by UML for modelling UIs. The expected improvement is to address the specification of UI aspects that are difficult to model using the standard UML, as suggested by the UI modelling difficulties presented in this chapter.

Chapter 4

UML*i* Notation and Metamodel

The Library System case study has demonstrated that aspects of UIs often specified in MB-UIDEs, as described in Chapter 2, can be modelled using standard UML constructs. The case study has also demonstrated that there are inconveniences in the use of UML for modelling UIs, as summarised by the UI modelling difficulties in Chapter 3. UML*i* aims to show that the set of diagrams in Figure 4.1 can be used to build UI models addressing the identified UI modelling difficulties. UML*i*, however, does not aim specifically to develop new user interface modelling constructs, but to adapt or reuse models and techniques proposed for use in MB-UIDEs in the context of UML (Principle 1 in Section 1.5.1). Thus, observations resulting from the survey of MB-UIDE proposals in Chapter 2 can be used to explain how the UML*i* UI models are used together to provide a description of user interfaces. For instance, the diagram in Figure 4.1 shows UML*i* diagrams representing aspects of user interfaces. There, the arrows represent dependencies between properties of UML*i* diagrams. Thus, the arrow (c) shows a dependence between properties of class diagrams and properties of user interface diagrams. Moreover, the arrow (a) shows an inter-dependence between properties of use case diagrams, activity diagrams and class diagrams. Dependencies similar to those represented by the arrows in Figure 4.1 can be observed in the MB-UIDEs proposals in Chapter 2. For example, the arrows (a) and (b) in Figure 4.1 can represent the dependencies between models created by state objects in Teallach [47]. In fact, state objects in Teallach are either domain objects or interaction objects that belong to a specific task. Therefore, the specification of which objects belong to each task creates the dependencies (a) and (b).

The comparison of the UML*i* models in Figure 4.1 with the selection of UML

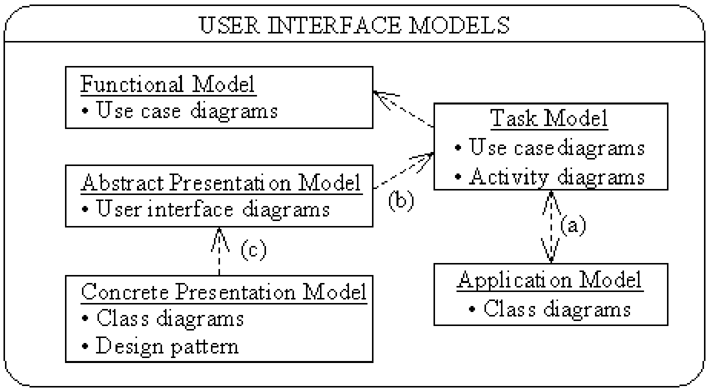


Figure 4.1: UML*i* user interface models. There, the enclosing rounded box represents the complete specification of a user interface. The internal boxes represent sets of UI properties distinct from the other boxes. The arrows connecting the boxes represent dependencies between properties of distinct boxes. For example, properties of the task model depend on properties of the functional model.

diagrams in Table 3.1 shows that the *user interface diagram* replaces the class diagram for abstract presentation modelling, and that sequence diagrams are not considered for task modelling. A description of the user interface diagram notation explaining how it addresses some of the identified UI modelling difficulties is presented in Section 4.1. Concerning sequence diagrams, they are not excluded from UML*i*, since UML*i* is a conservative extension of UML. However, as presented in Section 4.2, a combination of use case diagrams and activity diagrams is used for task modelling in UML*i*. Section 4.2 also presents the UML*i* constructs for activity diagrams, which have been introduced to address the UI modelling difficulties not covered by user interface diagrams. A description of the UML metamodel features required for describing the UML*i* metamodel is presented in Section 4.3. The UML*i* metamodel is introduced in Section 4.4. A proposal for a UML*i* method is presented in Section 4.5. A summary of how UML*i* addresses the UI modelling difficulties from Chapter 3 is presented in Section 4.6.

4.1 User Interface Diagram Modelling

This section introduces the UML*i* *user interface diagram*, a specialised class diagram used for the conceptual modelling of user interface presentations. The UI diagram provides a visual representation for containment between interaction

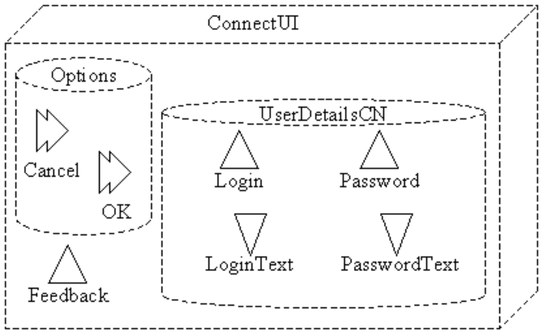


Figure 4.2: The `ConnectUI` presentation modelled using the UML*i* user interface diagram.

classes, addressing UI Modelling Difficulty 5. The UI diagram provides distinct visual representations for abstract roles that interaction classes can play in UI presentations, addressing UI Modelling Difficulty 6.

4.1.1 User Interface Diagram Notation

Recalling the Library System case study, the `ConnectUI` abstract presentation initially modelled as a class diagram, as in Figure 3.9, can be modelled as a user interface diagram, as in Figure 4.2. The two `ConnectUI` abstract presentation models are equivalent in terms of UI presentation specification. Moreover, it can be observed that, for example, the `ConnectUI`, `UserDetailsCN` and `LoginText` classes have distinct graphical representations in Figure 4.2, although they have the same graphical representation in Figure 3.9. Indeed, they have the same representation in Figure 3.9 since they are classes. However, they are also interaction classes, a category of classes responsible for an average of 50% of the deliverable code of interactive systems [95]. Therefore, considering the relevance of the interaction classes and the difficulty of visualising their abstract roles in class diagrams (UI Modelling Difficulty 6), UML*i* proposes the following six constructs for representing interaction classes.

- `FreeContainers`, , are rendered as dashed cubes. They are top-level interaction classes that cannot be contained by any other interaction class, e.g. top-level windows. They are also called *presentation units*, since the interaction classes in a `FreeContainer` are always presented at the same time. An instance of an interaction class 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 classes that are not `FreeContainers`. `Containers` provide a grouping mechanism for the designing of UI presentations, addressing UI Modelling Difficulty 5. Thus, the `Login` subclass of `Displayer` contained by the `UserDetailsCN` subclass of `Container`, as indicated by the composition between them in Figure 3.9, is represented by the placement of the `Login` element inside the `UserDetailsCN Container` in Figure 4.2.

- `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 classes 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, directly or indirectly, 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.

The approach of specifying concrete presentation models from user interface models can be based on the use of UML frameworks to create patterns from user interface diagrams in the same way that UML frameworks are used to create patterns from class diagrams representing abstract presentation models, as presented in Section 3.5. In fact, the user interface diagram is a specialised class diagram preserving the same facilities available for class diagrams.

4.1.2 Using the User Interface Diagram

The elicitation of objects can take place early during requirements analysis using scenarios [120]. In UML*i*, particularly, use cases can be used for the elicitation of interaction objects. Indeed, scenarios can be used for the elicitation of actions by scanning scenario descriptions looking for verbs [136]. Thus, actions may be classified as `Inputters`, `Displayers`, `Editors` or `ActionInvokers`. For example, Figure 4.3 shows a scenario for the `SearchBook` use case in Figure 3.1. Three interaction classes can be identified in the scenario: ∇*providing* that receives a book's title, author and year information; ∇*specify* that specifies some query details; and $\triangle$*displays* that presents the results of the query. Therefore, UML*i* can start the elicitation of interaction classes, using this association of actions with interaction classes, during requirements analysis. These action associations are often possible since the interaction classes of UML*i* are abstract ones. Therefore, user interface diagrams can initially be composed of interaction classes elicited from scenarios, as in Figure 4.4.

> *John is looking for a book. He can check if such book is in the library catalogue ∇providing its title, authors, year, or a combination of this information. Additionally, John can ∇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 $\triangle$displays the details of the matching books, if any.*

Figure 4.3: A scenario for the `SearchBook` use case.

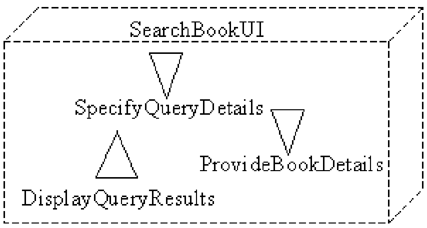


Figure 4.4: A preliminary version of the `SearchUI` presentation modelled using the UML*i* user interface diagram.

Figure 4.5 presents a refined version of the `SearchUI` presentation initially modelled as in Figure 4.4. Indeed, the elicited interaction classes in Figure 4.4 were decomposed into other interaction classes towards a more atomic description of the information provided by users. For example, `ProvideBookDetails` was decomposed into `BookAuthor_I`, `BookTitle_I` and `BookYear_I`. The grouping of interaction classes logically organises the new atomic interaction classes resulting from the decomposition of interaction classes. For example, `Database` and `PreviousQuery` derived from `SpecifyQueryDetails` were grouped into `QueryDomain`. The refinement of UI presentations can be performed in an incremental way to support and be supported by the activity diagrams. Moreover, new interaction classes, especially `ActionInvokers`, are modelled in order to support the system control flow. For instance, `OK` and `Cancel` in Figure 4.5 were modelled to allow users to provide control flow information to the Library System.

Recalling the benefits of using the user interface diagram notation, it would be a complex task to realise that the `Year` class is contained by the `BookDetailsSB` class, that is contained in the `QueryForm` class, that is contained in the `SearchBookUI` class, when using class diagrams to model the `SearchUI` presentation.

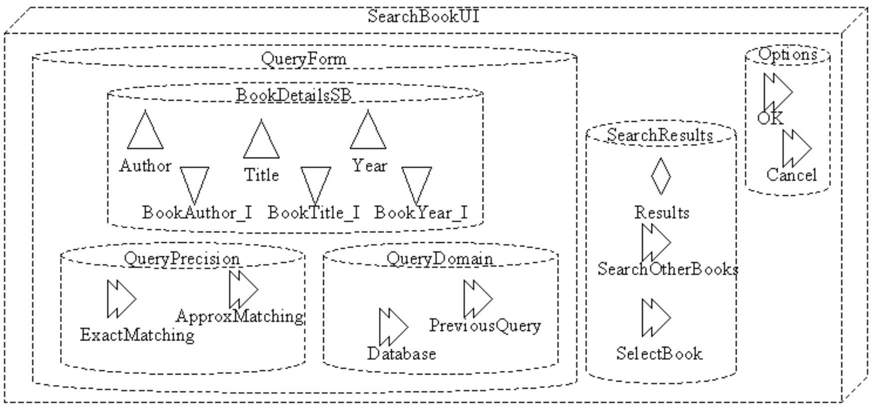


Figure 4.5: A refined version of the `SearchUI` presentation modelled using the UMLi user interface diagram.

4.2 Activity Diagram Modelling

Table 3.1 shows activity diagrams and sequence diagrams as alternative notations for task modelling using UML. Both diagrams suffer from most of the UI modelling difficulties relating to the specification of behavioural aspects of UIs identified in Chapter 3. However, the inappropriateness of sequence diagrams for modelling temporal dependencies between actions (UI Modelling Difficulty 7) has been the crucial factor in selecting activity diagrams as the notation for modelling UI behaviour in UMLi.

A more detailed analysis concerning the specification of temporal dependencies between actions in sequence diagrams may be required to confirm the selection of activity diagrams. In the sequence diagram in Figure 3.11, the fact that the `setLogin(getData())` action is performed before the `setPassword(getData())` action does not mean that they cannot be performed in a different order. The activity diagram in Figure 3.4 specifies that these two actions can be performed in any order and many times before the `Connect` activity is confirmed or cancelled (optional behaviour). Thus, these actions can be performed in the way specified in Figure 3.11. However, they may also be performed in an inverse order, they may be partially performed (if the `Connect` activity is confirmed or cancelled after the execution of one of the actions), or they may not be performed at all (if the `Connect` activity is confirmed or cancelled before the execution of the actions). Nevertheless, the same difficulty in modelling an optional behaviour also happens when modelling sequential behaviours since, for example, Figure 3.11 is not specifying that `setLogin(getData())` and `setPassword(getData())` must be performed in this specific order.

A modification to the UML specification stating that actions in sequence diagrams can only be performed in the order that they are specified would be a partial solution for specifying temporal dependencies using sequence diagrams. However, this solution would violate Principle 1 from Section 1.5.1, which requires that UML*i* be a conservative extension of UML.

UML*i* is concerned about the specification of temporal dependencies, since this is one of the UI features modelled by MB-UIDEs in Chapter 2. Thus, in the context of UML*i*, UI Modelling Problem 7 is solved by specifying a combination of use case diagrams and activity diagrams as its notation for task modelling. The UML*i* approach for the other behavioral UI modelling difficulties are discussed in the rest of this section.

4.2.1 From Use Cases to Activities: A Task Modelling Approach

Concerning the difficulty of identifying application entry-points (UI Modelling Difficulty 2), UML*i* introduces the InitialInteraction construct used for identifying entry points for interactive applications in activity diagrams. This construct is rendered as a solid square, ■, and is used like the UML Initial PseudoState [99], except that it cannot be used within states. In an interactive system, a top level activity diagram must contain at least one InitialInteractionState. Figure 4.6 shows a top level activity diagram for the Library System using a InitialInteraction. This is an entry-point of the Library System.

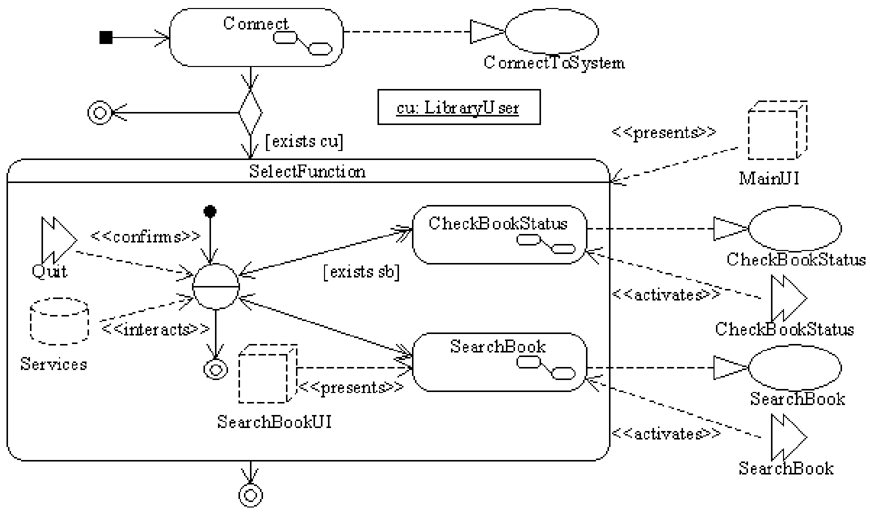


Figure 4.6: Modelling an activity diagram from use cases using UMLi.

Stereotyped ≪*communicates*≫ associations between use cases and actors, such as the association between the `SearchBook` use case and the `LibraryUser` actor in Figure 3.1, indicate that use cases can communicate directly with users when actors are representing users. Furthermore, associations between activities and instances of interaction classes, such as the association between the `SearchBook` activity and the `SearchBookUI` FreeContainer in Figure 4.6, indicate that these activities can communicate directly with users. Therefore, relationships between activities and use cases communicating with users could be created. In fact, use cases interacting with users and activities associated with instance of interaction classes may be specifying the same functionality at different levels of abstraction. In UMLi this relationship is graphically described by a realisation relationship. The diagram in Figure 4.6[1] makes clear which activity realises which use case. For instance, the `SearchBook` activity realises the `SearchBook` use case modelled in Figure 3.1. The implementation of realisations between use cases and activities is the UMLi approach to overcoming UI modelling Difficulty 1.

[1]This diagram is using the UMLi activity diagram notation explained throughout in this chapter.

For example, Figure 4.6 explicitly specifies that the `ConnectToSystem` use case in Figure 3.1 must be successfully performed by way of the `Connect` activity before the `SearchBook` use case can be performed by performing the `SearchBook` activity. In this way, the imprecise, but nevertheless useful, nature of use cases is preserved, thus abiding by Principle 1.

In terms of UI presentation design, interaction classes elicited in scenarios are non-container interaction classes that must be contained by `FreeContainers` (see the APP in Figure 3.8). Further, `FreeContainers` should be associated with activities, in the same way as the `SearchBookUI FreeContainer` is associated with the `SearchBook` activity in Figure 4.6, in order to be used by interactive systems. Therefore, interaction classes elicited from scenarios are initially contained by `FreeContainers` that are related to top-level activities through the use of a ≪*presents*≫ object flow, as described in Section 4.2.3. In that way, UI elements can be imported from scenarios to activity diagrams.

4.2.2 Selection States

Concerning the difficulty of modelling common interactive behaviours (UI Modelling Difficulty 4), UML*i* introduces a simplified notation for order independent, repeatable and optional behaviours. The notation used for modelling an order independent behaviour is presented in Figure 4.7(a). There an OrderIndependentState is rendered as a circle overlying a plus signal, $\oplus$, connected to the activities `A` and `B` by ReturnTransitions, rendered as solid lines with a single arrow at the SelectionState end and a double arrow at the selectable activity end. The double arrow end of ReturnTransitions identifies the selectable activities of the SelectionState. The distinction between the SelectionState and its selectable activities provided by the arrows in ReturnTransitions is required when SelectionStates are also selectable activities. Furthermore, a ReturnTransition is equivalent to a pair of statechart Transitions, one Transition connecting the SelectionState to the selectable activity, and one non-guarded Transition connecting the selectable activity to the SelectionState.

The notations for modelling repeatable and optional behaviours are similar, in terms of structure, to the order independent behaviour. The main difference between the notations for SelectionStates is in the symbols used for their selectors.

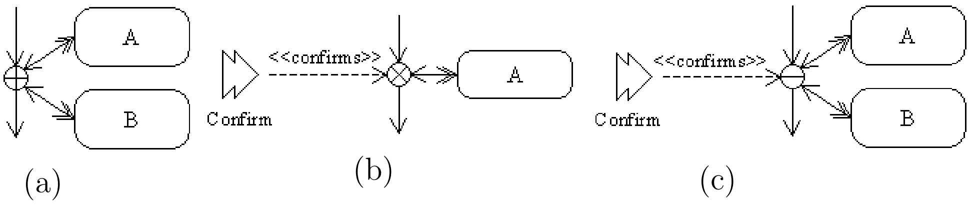


Figure 4.7: The UML*i* modelling of an OrderIndependenState in (a), a RepeatableState in (b), and an OptionalState in (c). These notations can be considered as macro-notations modelling the behaviours presented in Figures 3.6(a), 3.6(b) and 3.6(c), respectively.

The RepeatableState[2] is rendered as a circle overlaying a times signal, $\otimes$. The OptionalState is rendered as a circle overlaying a minus signal, $\ominus$. The RepeatableState requires a ≪*confirms*≫ interaction object flow, as shown in Figure 4.7(b), allowing users to interrupt the execution of `A` proceeding to the following activity. The OptionalState also requires a ≪*confirms*≫ interaction object flow, as shown in Figure 4.7, allowing users to finish the selection of selectable activities. The meaning and use of the ≪*confirms*≫ interaction object flow is discussed in the following section.

4.2.3 Interaction Object Flows

Concerning the necessity of a complete decomposition of activities into action states to achieve a description of object behaviours (UI Modelling Difficulty 3), there are common functionalities related to interaction objects that do not need to be modelled in detail to be understood. This fact is exploited by UML*i* through the provision of five specialised stereotypes for object flows indicating the specification of such common functionalities. Object flows are informally called *interaction object flows* when they are related to interaction classes and have one of the stereotypes presented as follows.

- An ≪*interacts*≫ interaction object flow relates an interaction object to an ActionState or to a PseudoState[3]. If the associated state is an ActionState,

[2]UML*i* considers a RepeatableState as a “selection” state since users might have the possibility of either confirming or cancelling the repeatable state iteration.

[3]PseudoStates are discussed in Section 4.3.2.

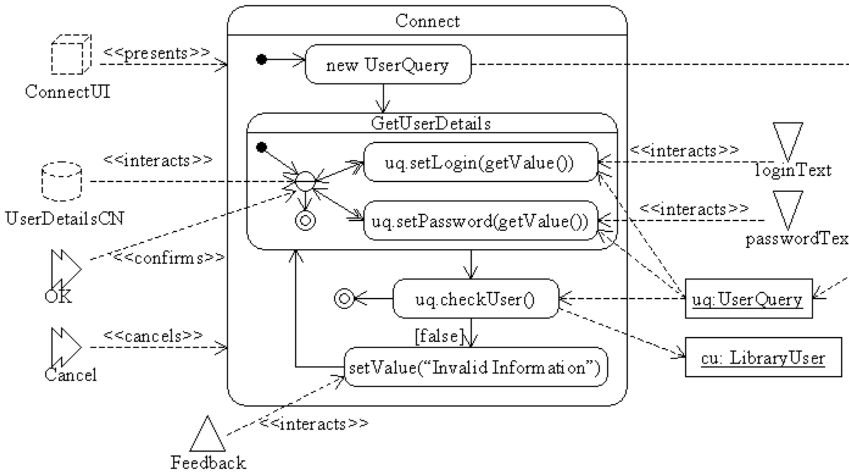


Figure 4.8: The `Connect` activity.

the object flow indicates that the state is responsible for an interaction between a user and the application. Thus, the ActionState can be an interaction where the user is invoking an object operation or visualising the result of an object operation. The ActionStates in the `GetUserDetails` activity in Figure 4.8 are examples of Inputters assigning values to some attributes of the `UserQuery` object from the domain. The △ `Feedback` in Figure 4.8 is an example of a Displayer used for visualising the "`Invalid Information`" message, if required. As can be observed in Figure 4.8, two abstract operations specified in the APP (Figure 3.8) have been used along with these interaction objects. The `setValue()` operation is used by Displayers and Editors for setting values to be presented to the users. The `getValue()` operation is used by Inputters and Editors for passing values obtained from users to domain objects. If the associated state is a PseudoState, the object flow indicates the enactment of the interaction object for interaction. If the interaction object is an instance of Container, such as the object indicated by the `UserDetailsCN` in Figure 4.8, then its contained objects are also activated for interaction.

- A ≪*presents*≫ interaction object flow relates an instance of FreeContainer to an activity. It specifies that the instance of FreeContainer should be visible while the activity is active and that the instance of FreeContainer should be invisible when the control flow leaves the activity. This behaviour of the `ConnectUI` FreeContainer is represented by the `InitiateConnectUI` and TerminateConnectUI activities in Figure 3.4 that are replaced by the ≪*presents*≫ interaction object flow in Figure 4.8. Moreover, invocations of the abstract setVisible() operation of the FreeContainer in Figures 3.5(a) and 3.5(b) are no longer required in UML*i* since they are implicitly specified by the ≪*presents*≫ interaction object flow in Figure 4.8. Therefore, the ≪*presents*≫ interaction object flow specifies that the ConnectUI FreeContainer and its contents are visible while the Connect activity is active.

- A ≪*confirms*≫ interaction object flow relates an instance of ActionInvoker to a SelectionState. It specifies that the SelectionState has finished normally. In Figure 4.8, an event associated with the ▷OK is responsible for finishing the execution of its related SelectionState normally. An OptionalState and a RepeatableState must have one ≪*confirms*≫ interaction object flow directly related to it, or indirectly related to it as in the case that the SelectionState is a selectable activity of another SelectionIState.

- A ≪*cancels*≫ interaction object flow relates an instance of ActionInvoker to any composite activity or SelectionState. It specifies that the activity or SelectionState has not finished normally. The flow of control should be rerouted to a previous state. The ▷ Cancel object in Figure 4.8 is responsible for allowing the user to cancel the Connect activity.

- An ≪*activate*≫ interaction object flow relates an instance of ActionInvoker to an activity. In that way, the associated activity becomes a triggered activity, that, after being activated, waits for the raising of the event associated with the ActionInvoker before starting.

4.2.4 The UML*i* SearchBook Activity Diagram

In the same way that the user interface diagram has simplified the appearance of UI presentations, the facilities of UML*i* have also simplified the appearance of activity diagrams, as presented in Figure 4.8. These UML*i* simplifications enable a discussion about the modelling of an activity diagram for the SearchBook functionality.

For the SearchBookUI in Figure 4.5, for example, the InitiateSearchUI and TerminateSearchUI activities would be visually more complex[4] than the

[4]In this thesis, the visual complexity of an activity diagram is defined as the result of dividing

InitiateConnectUI and TerminateConnectUI in Figures 3.5(a) and 3.5(b). In fact, the SearchUI presentation is composed of 22 interaction classes against the 10 of the ConnectUI, and the cyclomatic complexity of activity diagrams initiating and terminating UI presentations as shown in Figures 3.5(a) and 3.5(b) is 1. Thus, the ≪*presents*≫ SearchUI interaction object flow provides a significant simplification to the modelling of the SearchBook functionality. Nevertheless, the specification of the ≪*presents*≫ SearchUI interaction object flow in Figure 4.9 is redundant there, since it was previously specified in Figure 4.6.

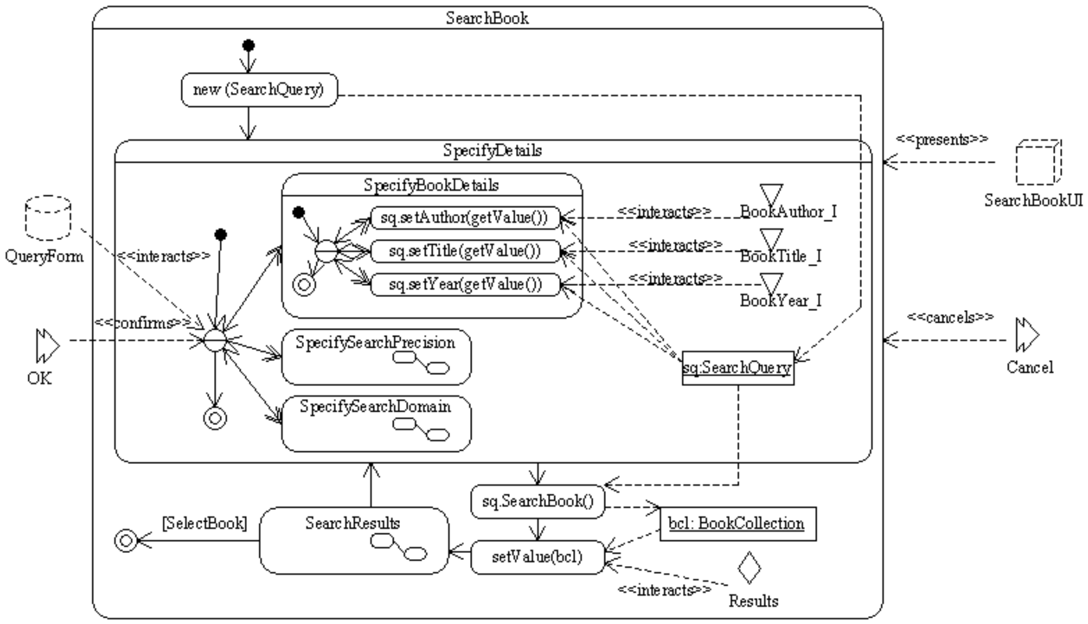


Figure 4.9: The SearchBook activity.

Another point that we would like to emphasise in Figure 4.9 is the composition of SelectionStates. For example, the OptionalState in SpecifyBookDetails is a component of the OptionalState in SpecifyDetails. This means that the selectable states of both OptionalState are activated at once, and that the OptionalState in SpecifyBookDetails relies on the ≪*confirms*≫ OK interaction object flow of the OptionalState in SpecifyDetails. This composition mechanism is important since it allows, for example, an incremental decomposition

the number of visual elements of the diagram by the cyclomatic complexity of the diagram, as discussed in Section 7.2.

of the `SpecifyDetails` activity through the decomposition of the `SpecifyBook-Details`, `SpecifySearchPrecision` and `SpecifySearchDomain` activities, as presented in Figure 4.9. Otherwise, the action states of the subactivities of `Specify-Details` related to its `OptionalState` would need to be modelled in `Specify-Details` itself, as required in UML.

This concludes the introduction to the UML*i* notation. The following sections in this chapter explain how the UML*i* notation can be supported by the UML*i* metamodel, an extended version of the UML metamodel. This explanation aims to clarify the relationship between UML and UML*i*, and to establish the foundations for implementing tool support for UML*i*. This explanation starts by presenting some features of the UML metamodel required to describe the UML*i* extensions in the UML metamodel.

4.3 UML Metamodel Architecture

UML has been revised several times since the elaboration of its first proposal submitted to the OMG in 1997 [74]. A description of the structural aspects of the UML diagrams is an outcome of these revisions. This description is called the *UML metamodel*, as it is partially composed of UML class diagrams and Object Constraint Language (OCL) constraints. The complete specification of the UML metamodel is described in the "UML Semantics" chapter in [99], which is organised by the Packages that compose the UML metamodel. The specification of the OCL is described in the "Object Constraint Language Specification" chapter in [99]. The UML metamodel is important since it plays a key role in the development of many UML tools, e.g., Rational Rose [116] and ARGO/UML [119], facilitating the modelling, handling and sharing of UML models.

Figure 4.10 presents the UML Packages which are partially organised by both the participation of the metaclasses in the UML diagrams and the dependencies among the metaclasses. For each Package, the UML documentation provides three informal and complementary descriptions of the metamodel: the *abstract syntax*, *well-formedness rules* and *modelling element semantics*. The class diagrams, that can properly be called the UML metamodel, are part of the abstract syntax. The abstract syntax also provides a description in prose of each element that composes the UML metamodel. The well-formedness rules are written in OCL. These rules provide additional constraints concerning the metaclasses of the abstract syntax

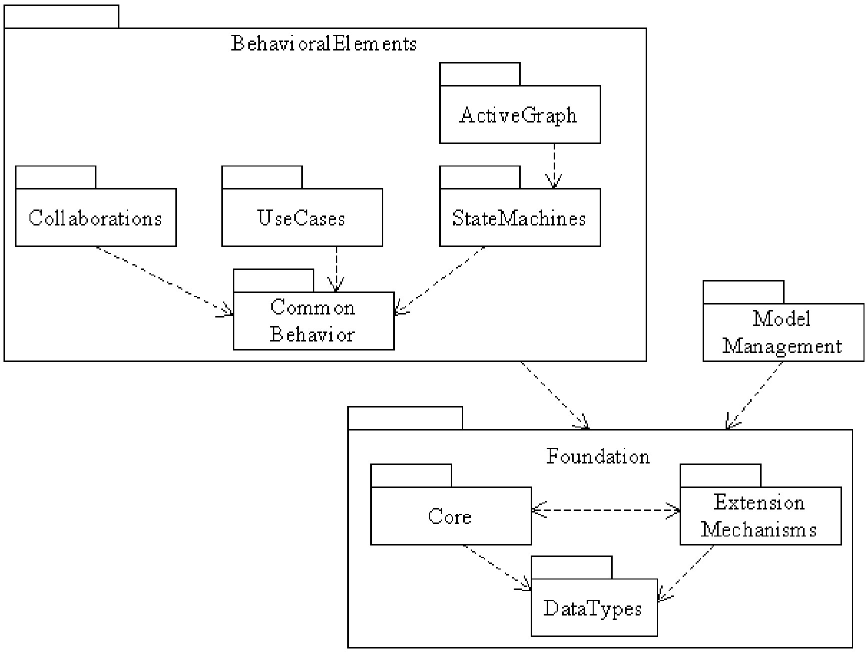


Figure 4.10: Packages of the UML metamodel.

that cannot be expressed using the class diagram constructs alone. These well-formedness rules are also supported by textual descriptions of their meaning. The "semantics" of a modelling element is a description, once again in prose, about the meaning of the Package itself.

Most of the metaclasses of the class diagram are specified in the `Core` Package within the `Foundation` Package, as in Figure 4.10, where the structural constructs of UML are specified. Metaclasses of dynamic (or behavioural) diagrams are specified in Packages within the `BehavioralElements` Package. The metaclasses of the collaboration and sequence diagrams are specified in the `Collaborations` Package. The metaclasses of the use case diagram, state diagram (or statechart) and activity diagrams are specified in the `UseCases`, `StateMachines` and `ActivityGraphs` Packages, respectively.

A comprehensive presentation of the UML metamodel is beyond the scope of this thesis. However, a presentation of part of the `Core` Package in Section 4.3.1 and the `StateMachines` and `ActivityGraphs` Packages in Section 4.3.2 provides a context for explaining how UML*i* constructs are incorporated into the UML specification. In fact, the UML*i* `UserInterfaces` Package depends on the `Core` Package, and the `IntegratedActivities` Package depends on the `StateMachines` and ActivityGraphs Packages.

4.3.1 The Core Package

A UML diagram is a set of elements related to each other according to the UML metamodel. A description of the Core Package can clarify how UML elements are inter-related. Figure 4.11 is a partial representation of the abstract syntax of the Core Package extracted from [99]. There, an element in a UML diagram is an instance of ModelElement which may belong to a Namespace (a Namespace is also a ModelElement). Classifier is an important category of Namespace since it can be composed of many Features required, for example, to enable a ModelElement to represent entities of the domain being modelled. An Attribute is an example of a StructuralFeature. An Operation is an example of a BehavioralFeature. A Class and an Interface (the Interface metaclass is not in Figure 4.11) are specialised Classifiers.

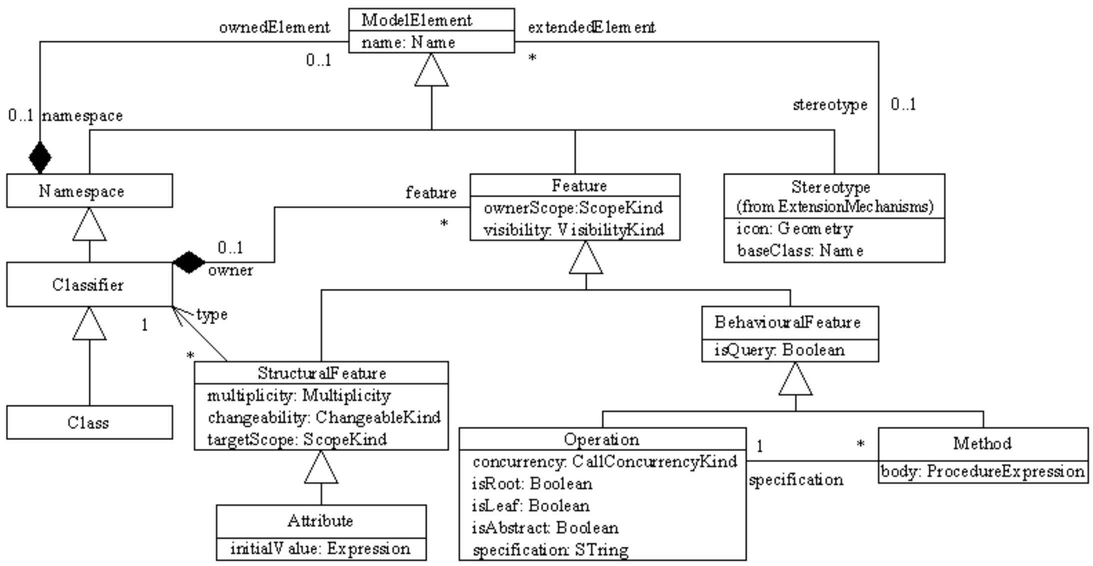


Figure 4.11: Partial representation of the Core Package of the UML metamodel.

For an in-depth description of the Core Package and the other Packages of the UML metamodel, we suggest the reading the UML specification [99] since it is, to the best of our knowledge, the sole document comprehensively presenting the UML metamodel.

Relying on its metamodel, the UML specification [99] introduces the OMG

XML[5] Metadata Interchange (XMI) format which is specified by a document type definition (DTD). In fact, the XMI DTD is the grammar of documents in the XMI format. Therefore, UML-based tools can save, load and exchange UML models using XMI files. For instance, UML models can be saved by writing the XMI text for each instance of ModelElement (see Figure 4.11) to a text file.

A *UML construct* in this thesis is a metaclass, or in other words, a class in a class diagram of the UML metamodel. This is the definition of *UML construct* used to explain the semantics of UMLi in Chapter 5. In this thesis, the names printed using a sans serif font are either names of UML constructs, as specified in [99], or names of UMLi constructs, as specified in Section 4.4.

4.3.2 The `StateMachines` and `ActivityGraphs` Packages

State diagrams (statecharts) and activity diagrams are state-transition diagrams composed of nodes inter-connected by arcs. Figure 4.12 presents a partial and merged representation of the abstract syntax diagrams of the `StateMachines` and `ActivityGraphs` Packages used to support the modelling of state-transition diagrams in UML. There, nodes are modelled using the StateVertex construct and arcs are modelled using the Transition construct. Figure 4.12 also shows that the StateVertex construct can be specialised into a State or a PseudoState construct. A StateMachine can be associated with a StateVertex that is a top State in a statechart. PseudoState constructs are used to model statechart constructs that are neither States nor Transitions. Thus, PseudoStates of statecharts are, for example, Forks, Joins, InitialStates and FinalStates, depending on the value of their `kind` Attribute. The type of `kind` is PseudoStateKind specified in the DataTypes Package.

As explained in the text of the UML documentation, but not in Figure 4.12, a State construct can be used to represent both a statechart's states and an activity diagram's activities. In fact, the relationship between activity diagrams and statechart diagrams is unclear in Figure 4.12. This relationship may be better understood by initially ignoring the `ActivityGraphs` metaclasses in Figure 4.12. In this case, the context of a StateMachine is a single ModelElement. This means that the diagram aggregated to the StateMachine through its top StateVertex can describe the behaviour of a single element. Then, the State construct is used to

[5] Extensible Markup Language.

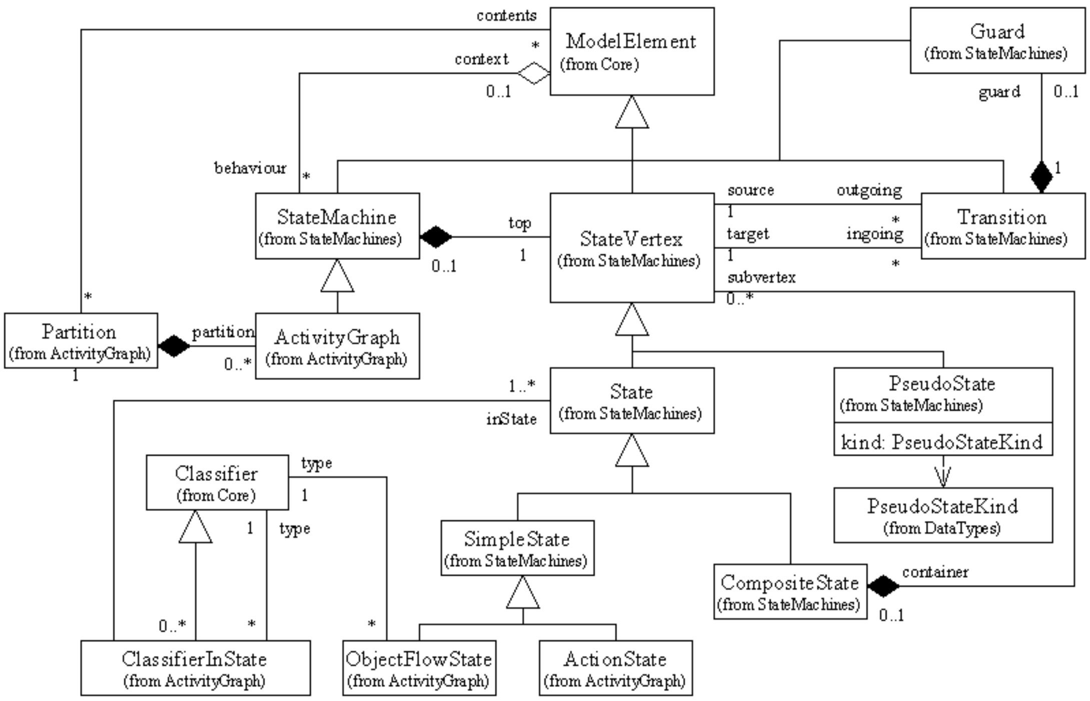


Figure 4.12: Partial representation of the `StateMachines` and `ActivityGraphs` packages of the UML metamodel.

specify states of the elements represented by the ModelElement construct. Further, the Transition construct is used to represent transitions between states of the same instance of ModelElement. Considering the `ActivityGraphs` metaclasses back in the metamodel, as presented in Figure 4.12, it is possible to use a common set of constructs of both `StateMachines` and `ActivityGraphs` Packages to describe the behaviour of any subset of instances of the `ModelElement` construct. In this case, the diagram aggregated to the ActivityGraph subclass of StateMachine, can describe the behaviour of a set of elements represented by the Partition class. Then, the State construct is used to specify states of the set of elements represented by the Partition construct. Further, the Transition construct is used to represent transitions between states of the same instance of `Partition`. It is important to observe that transitions between the same instance of `Partition` include transitions between different `ModelElements`, which are essential to explain, for instance, how functionalities are implemented in terms of collaborations between objects.

Figure 4.12 also helps to explain how object flows are implemented in a UML metamodel, a common difficulty in understanding activity diagrams. There, the

ObjectFlowState and ClassifierInState metaclasses are used to implement object flows. In this case, the ObjectFlowState is rendered as an arrow connecting elements of the ClassifierInState to action states, as emphasised in Figure 4.13. If the arrow of the ObjectFlowState is from an action state to the ClassifierInState it means that an object of the type of ClassifierInState is created and allocated before the execution of the action state. If the arrow is from the ClassifierInState, to an action state, it means that the object that is already instantiated is allocated before the execution of the action state, probably supporting the execution of the action state.

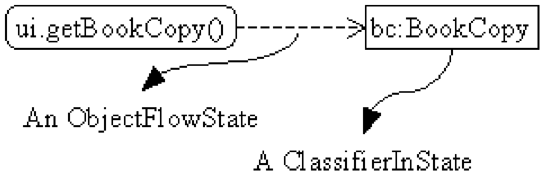


Figure 4.13: The UML constructs of an object flow.

From the description of the UML metamodel above it is possible to specify the UMLi metamodel.

4.4 UML*i* Metamodel Architecture

The UMLi metamodel is composed of few new constructs and few new OCL rules when compared to the number of constructs and OCL rules of the UML metamodel. Indeed, Principle 2 says that UMLi should introduce as few additional constructs as possible into UML. However, despite the fact that there are few extensions, these additions require some more explanation in terms of how they are incorporated into UML. Thus, Figure 4.14 identifies where in the package diagram of the UML metamodel the UMLi additions are incorporated. There, four tags (numbered black disks) identify the UMLi additions.

Tag 1 identifies the `UserInterfaces` Package used to support the modelling of user interface diagrams. The `UserInterfaces` Package is specified in Section 4.4.1. Tag 2 identifies the `IntegratedActivities` Package used to support the modelling of selection states. The `IntegratedActivities` Package is specified in Section 4.4.2. Tag 3 identifies the additions in the `DataTypes` Package used to support the modelling of InitialInteractions. Tag 4 identifies the relaxation of

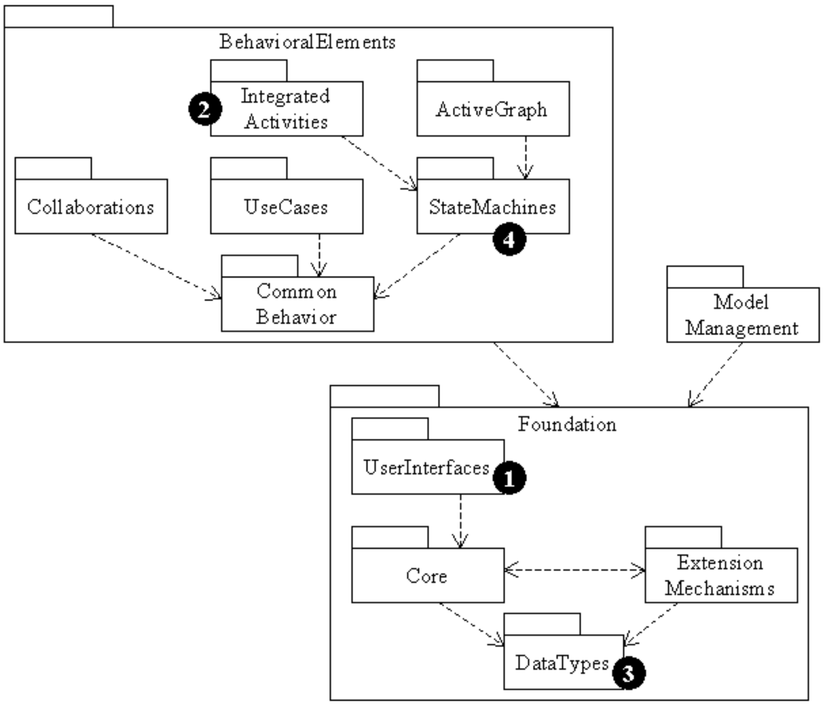


Figure 4.14: Packages of the UMLi metamodel.

some OCL constraints in the `StateMechines` Package to support the modelling of interaction object flows. Additions in both `DataTypes` and `StateMachines` Packages are specified in Section 4.4.3. Finally, a discussion about how the stereotypes of the interaction object flows are supported in the UML metamodel is presented in Section 4.4.4.

4.4.1 The `UserInterfaces` Package

Tag 1 in Figure 4.14 identifies the `UserInterface` Package in the UMLi metamodel. The contents of the Package describing the abstract syntax of user interface diagrams is presented in Figure 4.15. There, the InteractionClass metaclass is a specialisation of the Class metaclass of the Core package. Moreover, the other metaclasses in Figure 4.15 are subclasses of InteractionClass. This means that the user interface diagram can be considered as a specialised class diagram.

The InteractionClass metaclass in Figure 4.15 corresponds to the InteractionClass of the abstract presentation pattern (APP) in Figure 3.8. The main difference between the diagram of the `UserInterface` Package and the APP is that operations of the APP classes are not specified in UMLi metaclasses of the

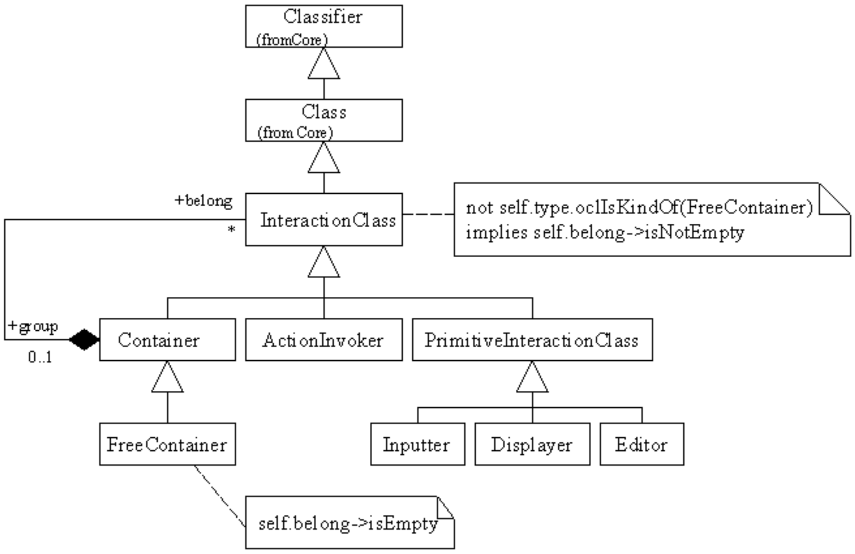


Figure 4.15: UserInterfaces package of the UML*i* metamodel.

Package. The necessity of such operators is partially substituted by the specification of the interaction object flow stereotypes, as discussed in Section 4.2.3. The substitution is completed by the implicit introduction of the the `getValue()` and `setValue()` operations, that do not need to be supported by any additional construct. The invocation of these operations corresponds to the invocation of `obj.getValue()` and `obj.setValue()`, where `obj` is the associated ClassifierInState of type InteractionClass.

4.4.2 The `IntegratedActivities` Package

Tag 2 in Figure 4.14 identifies the `IntegratedActivities` Package in the UML*i* metamodel. The contents of the `IntegratedActivity` Package describing its abstract syntax is presented in Figure 4.16. There, the ReturnTransition and SelectionState constructs are the basic constructs used to implement selection states.

The ReturnTransition Construct

A ReturnTransition construct is a metaclass composed of two Transitions, one from the SelectionState to a selectable activity (that is a StateVertex), and the other from the selectable activity to the SelectionState. The ReturnTransition can accept

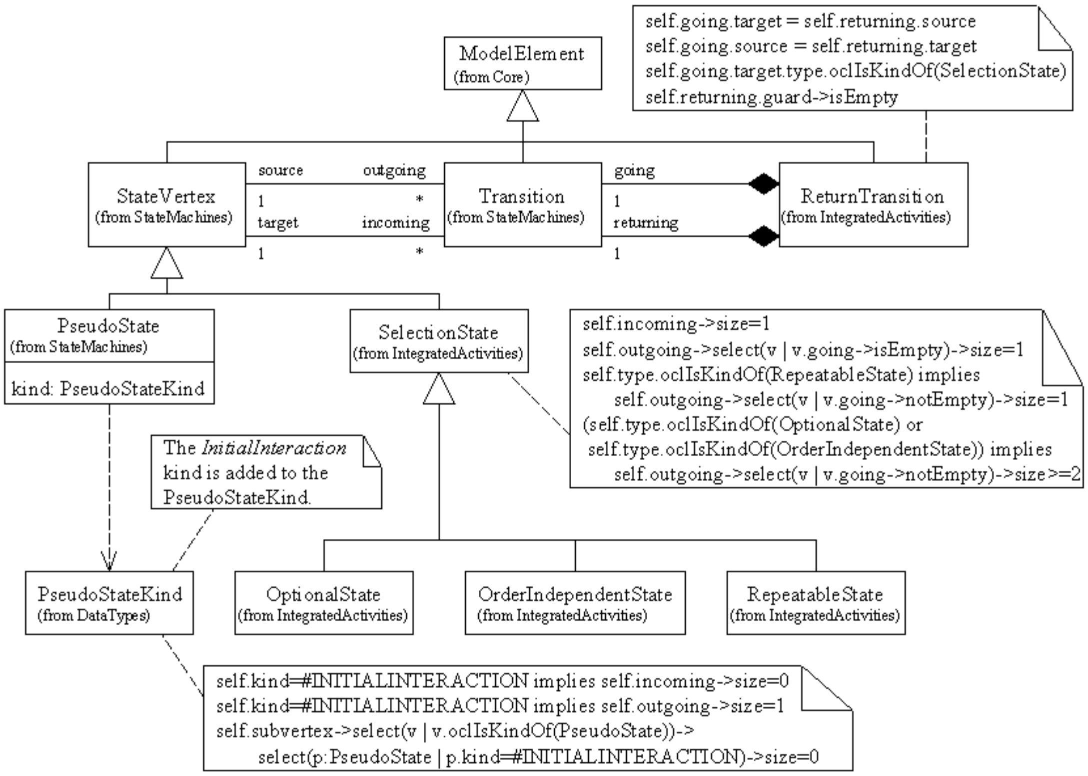


Figure 4.16: IntergratedActivities package of the UMLi metamodel.

one Guard that affects only its outgoing Transition, since its returning Transition is a non-guarded one, as presented in Figure 4.16.

The SelectionState Construct

The SelectionState construct is a subclass of the State metaclass composed of one SelectionState and one or more *selectable activities*. A selectable activity is a StateVertex that can be activated, and consequently reached, when the application workflow is in a SelectionState. The `target` of the default `outgoing` Transition of a selectable activity identifies its SelectionState. Figure 4.16 additionally shows that a SelectionState is specialised into `OptionalState`, `OrderIndependentState` and `RepeatableState`.

4.4.3 Extensions in the DataTypes and StateMachines Packages

Tags 3 and 4 in Figure 4.14 identify the DataTypes Package where the InitialInteraction construct is specified, and the StateMachines Package where some OCL rules enable the construction of interaction object flows.

The InitialInteraction Construct

The InitialInteraction construct is modelled as an additional value for the PseudoStateKind class specified in the DataTypes package. Thus, the addition of the new value `InitialInteraction` for the PseudoStateKind metaclass specifies the InitialInteraction construct (or a PseudoState construct of the kind InitialInteraction), as shown in Figure 4.16. Some additional OCL rules are required in constructs of the StateMachine Package.

Interaction Object Flow Constraints

The UML specification says that a ClassifierInState can be associated with an `ActionState` only. However, there is no OCL rule in the documentation specifying such a constraint. Thus, the following constraint could be an OCL rule of the ClassifierInState construct:

- A ClassifierInState cannot be associated with a State that is not an ActionState.

  ```
  self.inState.forAll(s | s.oclIsKindOf(ActionState))
  ```

Object flows that are interaction object flows are useful if they can be associated with other categories of states, as indicated in Section 4.2.3. Therefore, based on the fact that InteractionClasses are UML*i* constructs, the ClassifierInState rule above can be relaxed as follows:

- A ClassifierInState that is not an InteractionClass cannot be associated with an ActionState.

  ```
  (not self.type.oclIsKindOf(InteractionClass)) implies
     self.inState.forAll(s | s.oclIsKindOf(ActionState))
  ```

- A ClassifierInState that is an InteractionClass can be associated with ActionStates, CompositeStates, SelectionStates and PseudoStates.

```
self.type.oclIsKindOf(InteractionClass) implies
   self.inState.forAll(s | s.oclIsKindOf(ActionState) or
                           s.oclIsKindOf(CompositeState) or
                           s.oclIsKindOf(SelectionState) or
                           s.oclIsKindOf(PseudoState))
```

4.4.4 Interaction Object Flows and Their Stereotypes

The extensions in the UML metamodel presented so far are based on the fact that new graphical notations were needed. Any visual indication that the meaning of a particular construct is being extended, however, is enough to identify interaction object flows. The Stereotype construct is powerful enough to specify the extended meaning of interaction object flows. Therefore, the interaction object flow stereotypes, e.g., ≪*presents*≫, informally introduced in Section 4.2.3, can address the necessity of visually identifying interaction object flows.

The good thing about using Stereotype, when its notation is appropriate, is that it is one of the standard extension mechanisms of UML. This means that the UML metamodel does not need to be modified to accommodate extensions based on this mechanism. In fact, according to Figure 4.11, every ModelElement can have one Stereotype[6] attached to it. Therefore, an interaction object flow is composed of an ObjectFlowState that (i) is associated with a ClassifierInState of an InteractionClass, and (ii) is associated with a Stereotype of one of the five interaction object flow stereotypes introduced in Section 4.2.3.

4.5 A Proposal for a UML*i* Method

The proposed 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 [32].

[6] The Stereotype metaclass belongs to the `ExtensionMechanisms` Package of the UML metamodel.

Part of the Library System case study is used for exemplifying the use of the UML*i* method.

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 3.1 identified some application functionalities. Scenarios can be used as a textual description of the use case goals. For instance, the scenario presented in Figure 4.3 is a textual description of the `SearchBook` use case in Figure 3.1. 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, `SpecifyBook` is a sub-goal of `BorrowBook` in Figure 3.1.

Step 2 InteractionClass *elicitation. Scenarios of less abstract use cases may be used for* InteractionClass *elicitation.*

Scenarios can be used for the elicitation of InteractionClasses , as described in Section 4.1.2. In this case, elicited InteractionClasses are related to the associated use case. Relating InteractionClasses directly to use cases can prevent the elicitation of the same InteractionClass 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 InteractionClasses 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 InteractionClasses 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 3.1.

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 4.6 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 4.6, 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 4.2.1.

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 4.8 presents a decomposition of the `SearchBook` activity introduced in Figure 4.6. The use of ≪*interacts*≫ object flows relating instances of InteractionClasses 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 instances of InteractionClasses required for describing action states. There are two mechanisms that allow UI designers to refine a conceptual UI presentation model.

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

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

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

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

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.

4.6 Summary

The UML*i* notation is introduced in Sections 4.1 and 4.2. There, the UI modelling difficulties described in Chapter 3 were addressed by the UML*i* extensions summarised as follows.

- UI Modelling Difficulty 1 is addressed by the well-established links between use case diagrams and activity diagrams that explain how user requirements identified during requirements analysis are described in the application design.
- UI Modelling Difficulty 2 is addressed by the InitialInteraction construct that provides a way for modelling application entry-points.
- UI Modelling Difficulty 3 is addressed by the use of both interaction object flows and object flows in UML*i* activity diagrams that provide the relationship between visual components of the user interface and domain objects.
- UI Modelling Difficulty 4 is addressed by the use of selection states in UML*i* activity diagrams that simplify the modelling of interactive system behaviour.

- UI Modelling Difficulty 5 is addressed by the notation for Containers that facilitates the grouping of widgets. Further, the UMLi user interface diagram introduced for modelling abstract user interface presentations simplifies the modelling of the use of visual components (widgets).

- UI Modelling Difficulty 6 is addressed by the notation for InteractionClasses that facilitates the visual identification of the abstract roles of widgets in user interfaces.

A brief description of the UML metamodel was presented in Section 4.3. Based on the UML metamodel, the UMLi syntax described in terms of the UMLi metamodel is introduced in Section 4.4.

The UMLi specification introduced so far is equivalent in its level of detail to the UML specification in [99]. The specification of the UMLi notation provides sufficient detail to allow developers to design interactive systems using pen and paper. The specification of the UMLi metamodel may facilitate the identification of disallowed relationships between constructs in UMLi diagrams when implemented in UML-based tools.

The proposal for a semantics for UMLi is presented in the next chapter to complete the specification of UMLi introduced in this chapter.

Chapter 5

UMLi Semantics

The UML metamodel can provide a framework for semantic description. Thus, the metamodel is used by UML to informally provide a semantics for UML describing the meaning of the UML metamodel constructs using English language [99]. In this chapter, the metamodel is used by UMLi to formally provide a semantics for UML and UMLi.

The lack of a formal semantics is an identified difficulty related to the use of UML [19, 36, 37, 88]. Indeed, this lack of a formal semantics allows ambiguous interpretations of UML models, and consequently, of UMLi models. This may lead to disputes over the interpretation of the models, and to the implementation of systems that do not fulfill the intentions of the designers. A formal semantics for UMLi could help to solve this problem of contradictory interpretations of UML and UMLi models. Further, such a semantics could be useful for implementing automated verification of models to identify incorrect uses of the notation, as well automated as interpretation of models to generate software code.

This chapter is structured as follows. Section 5.1 presents an overview of current work on providing a semantics for UML. Section 5.2 gives a brief introduction to LOTOS, the formal specification language used to describe a UMLi semantics. Section 5.3 establishes the foundations of the Φ function introduced in this thesis that translates UMLi models into LOTOS specifications, building on the UMLi metamodel. Section 5.4 presents a part of the Φ function for some structural aspects of UMLi models. Section 5.5 presents a part of the Φ function for some behavioural aspects of UMLi models. Conclusions are presented in Section 5.7.

5.1 Approaches For a UML*i* Semantics

Any approach to the development of a UML*i* semantics should consider the still open question of how best to provide a formal description of the semantics of UML. For example, a specification of UML in terms of a mathematical notation or a formal specification language could provide a semantics for UML. The expressiveness gap between UML and a mathematical notation may be bigger than between UML and a formal specification language. Indeed, the semantics of specification languages are often provided by their mathematical specifications. Thus, the approach in this chapter is based on the use of a formal specification language. Such a specification language should be powerful enough to specify the behavioural and structural aspects that can be described by UML*i* models. Most formal specification languages provide facilities for modelling and verifying behavioural and structural aspects of software systems. Some formal specification languages, e.g., Z [130], are appropriate while describing structural aspects of software systems. However, it may be difficult to check some behavioural properties, such as concurrency, since these languages do not describe any computation that explains how their specifications can be executed [50]. A combination of specification languages could be considered, e.g., Z [130], CCS [91] and CSP [57]. However, it would be most desirable to use just one specification language to provide the required formalism.

In this chapter, a LOTOS [16, 63] approach for specifying the semantics of UML is presented. LOTOS is a specification language that has succeeded in the challenging task of describing structural and behavioural aspects of software systems using a single notation. Indeed, LOTOS has incorporated the specification facilities of CCS and CSP, as well the facilities for specifying the abstract data types of ACT-ONE [33]. Moreover, LOTOS is an International Standardization Organization (ISO) standard specification language developed for the formal description of the Open System Interconnection (OSI) architecture that is applicable to distributed, concurrent systems in general. Thus, through the Library System case study it is described how LOTOS can be used to specify a semantics for representative class and activity diagram constructs. Furthermore, object flows described in this chapter and used in activity diagrams provide a connection between structural models, e.g., class diagrams, and behavioural models, e.g., activity diagrams without object flows.

Previous approaches to providing a formal semantics for the UML can be classified in many ways. In this chapter, related work is presented emphasising the existing dichotomy between approaches formalising structural and behavioural aspects of the UML.

There are many approaches to formalising structural aspects of UML. Evans et al. [36] is an example of one of the two approaches of *the precise UML group* [138] of researchers concerned about the lack of a semantics for UML. In this approach, a semantics for UML is expected to be achieved by the formalisation of some class diagram constructs used to build the *UML metamodel*, as described in [99]. Therefore, the semantics of the other constructs of UML can follow from the previously formalised constructs, specifying in this way a semantics for the entire UML. Particularly in Evans et al. [36], Z [130] is used to formalise the class diagram. In Evans and Kent [37], the use of set theory embedded in Object Constraint Language (OCL) [143] constraints is used to provide a semantics for the generalisation and package concepts. The Action Semantics proposal originated by Mellor et al. [88] also aims to achieve a formalisation of the OCL. Further, Richters and Gogolla [117] have proposed a formalisation of the OCL in an integrated way with some constructs of class diagrams. [37, 117] are examples of the second approach of the precise UML group, where a semantics for UML is expected to be achieved by the formalisation of the OCL.

There is much work on formalising behavioural aspects of UML. The first mention in this context should probably be of David Harel's work which has influenced the development of the UML [51]. Considering this, we can say that the descriptions in [52, 53] of the statechart semantics are partial descriptions of the semantics of UML. There are other results on a formalisation of the behavioural aspects of UML. For instance, one of the final aims of having a formal specification is the possibility of verifying if a model is correct. Latella et al. [77] and Lilius and Paltor [79] present robust work that describes how statecharts can be verified. Both of them are based on the use of the SPIN model checker [59]. Moreover, Latella et al. [77] indicates several points where the informal specification of UML is silent in terms of a proper specification.

There is a concern about this dichotomy between the distinct formalisation of the structural and dynamical aspects of UML. Wang et al. [142] describes a formalisation of the dynamic models of the OMT [121]. This work is considered in this proposal since the OMT is one of the three major predecessors of the UML, and its formalisation is based in LOTOS. More recently, Breu et al. [19]

have proposed another use of mathematical models, denoted *system models*, to describe most of the constructs of UML. It looks like a promising approach for a mathematically-grounded semantics for UML. The approach, however, is currently a long way from being complete enough to provide a verification facility for UML. This dichotomy between the formalisation strategies of the dynamic and structural parts of UML models may be a problem for a complete verification of UML models. Keeping the semantics tractable at a certain level, as described in Latella et al. [77], may be an appropriate strategy for the verification of UML models. However, dynamic and structural models are interdependent, and as such they might usefully be verified at once.

Finally, Clark and Moreira [25] have an approach based on LOTOS-E [62], an enhanced version of LOTOS, for a UML semantics. Very promising in terms of describing dynamic and structural aspects of UML models using a single notation, the approach is similar to that adopted for UML*i* in this chapter, e.g., for translating Classes and Attributes. The lack of details about the mapping of a significant number of UML constructs, however, makes it difficult to use the approach in [25] either to translate the Library System into LOTOS-E specifications or as a basis for describing the semantics of UML*i*.

5.2 The LOTOS Specification Language

5.2.1 Basic LOTOS

A system $\mathcal{S}$ can be specified by a LOTOS process that might be composed of other LOTOS processes. A LOTOS process $P[G]$ has a set of observable gates $G = \{g_1, g_2, ..., g_n\}$. LOTOS assumes that an environment associated with the process P exists that is composed of the process P, its subprocesses, and an unspecified *observer* process that is always ready to observe anything the system $\mathcal{S}$ may do. Thus, a LOTOS *action* can be defined as the interaction between a defined process and, at least, the *observer* process. The behaviour of a process is defined by an algebraic expression composed of: actions that may be observed at the gates (*unary* operators); internal actions that cannot be observed at any gate (*nullary* operators); and of other processes that specify their own algebraic expressions (composed operators). All these operators are connected by binary operators, as presented in Table 5.1. These algebraic expressions are called *behaviour expressions*.

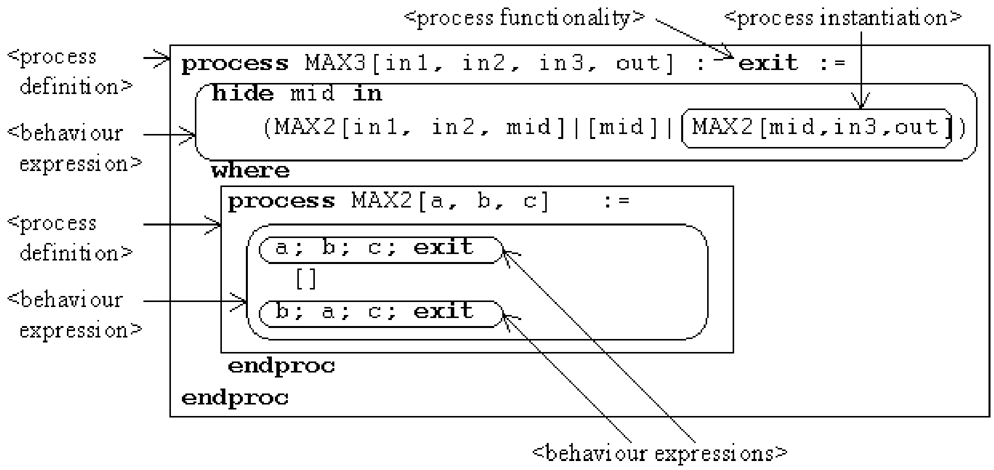


Figure 5.1: Definition of a LOTOS process extracted from [16].

Figure 5.1 shows an example of a LOTOS process definition. There, the `MAX3` process is defined that has `MAX2` as a subprocess. The observable gates of `MAX3` are `in1`, `in2`, `in3` and `out`, and the observable gates of `MAX2` are `a`, `b` and `c`. The behaviour of `MAX3` is defined by the behaviour of the two instances of the `MAX2` process that are synchronised on the `mid` gate, as specified by the interleaving binary operation (|[]|) connecting the two instantiations of `MAX2`. Further, an operation that is expected to terminate has the **exit** functionality, as in the `MAX3` process specification. Otherwise, the process can have a **noexit** functionality. As a LOTOS convention, action names are written in lowercase letters and process names are written in uppercase letters. To facilitate the reading of this chapter, words in bold fonts are reserved words of LOTOS.

5.2.2 Full LOTOS

The LOTOS presented so far is *basic LOTOS*, since only behavioural aspects can be specified. The specification of observable actions, however, can be refined with the use of abstract data structures and values. Thus, the abstract data type specification language ACT-ONE [33] has been integrated with LOTOS, specifying *full LOTOS*. New interprocess communications can be achieved in *full LOTOS*.

Category	Operator	Notation	Description
Binary Operators	Action prefix	;	The expression $a; C$ means that the process behaves like C after the execution of a.
	Enabling	>>	The expression $C >> D$ means that the process D is enabled if, and only if, C terminates successfully.
	Choice	[]	The expression $a; C[]b; D$ means that the process can start to behave like either C or D depending on the next actions provided by the interaction of the current process with its environment. If the environment offers an action a then the process starts to behave like C. If the environment offers an action b, then the process starts to behave like D.
	Interleaving	|||	The expression $C\|\|\|D$ means that the actions of the processes C and D do not need to synchronise (completely independent).
	Interleaving with synchronisation gates	|[]|	if a is an observable action in both C and D processes, then the expression $C\|[a]\|D$ means that C and D must synchronise in order to perform a.
	Hiding	**hide** ... **in**	The expression **hide** x **in** C means that the action x originally specified as observable by any other process interacting with the current process at the gate x now can only be observable from the process C.
	Disabling	[>	The expression $C[> D$ means that the process C can be interrupted any time before its successful termination in the case that the environment provides an unspecified interrupting action called a *disabling operator*. In this case, the process starts to behave like D.
Unary Operators	stop	**stop**	This operator, which offers no event to the environment, means that the process becomes inactive.
	exit	**exit**	This operator offers an special event to the environment notifying the successful termination of the process. After raising this special event, the process also becomes inactive as a result of the **stop** operator.

Table 5.1: Unary and binary operators of LOTOS. It is assumed that a and b are LOTOS observable actions and that C and D are LOTOS processes in the descriptions.

- Value passing: Suppose that the processes C and D are synchronised on the gate x, e.g. $C|[x]|D$. Moreover, suppose that the process C is performing $x!TRUE$ and the process D is performing $x?b$:**Bool**. We can say that C is passing the value $TRUE$ to the process D. Moreover, this $TRUE$ value is assigned to the b variable of the process D.

- Signal matching: Once again, suppose $C|[x]|D$. This time, suppose that C is performing $x!z_1$ and D is performing $x!z_2$. This means that C and D will only be synchronised if the values of z_1 and z_2 become the same ($z_1 = z_2$).

The type of the b variable in the *value passing* example is **Bool** (that means, *Boolean*). *Full* LOTOS, or just LOTOS, provides a set of primitive types for modelling simple data structures. For instance, the LOTOS primitive **nat** denotes a type the domain of which must be a natural number. LOTOS also provides the ability to specify more complex data structures composed of primitive types and other complex data structures. For example, the `PERSON` type definition presented as follows describes a type that might be used by objects of a class `PERSON`.

```
type PERSON is String
  sorts Person
  opns  mk_Person:                  -> Person
        mk_Person2: String, String  -> Person
        setname: Person, String     -> Person
        getname: Person             -> String
        setcode: Person, String     -> Person
        getcode: Person             -> String
  eqns  forall name1, name2, code1, code2: String
        ofsort String
          getname(mk_Person2(name1,code1)) = name1;
          getcode(mk_Person2(name1,code1)) = code1;
        ofsort Person
          setname(mk_Person2(name2,code2),name1) = mk_Person2(name1,code2);
          setcode(mk_Person2(name2,code2),code1) = mk_Person2(name2,code1);
endtype
```

Figure 5.2: `Person` type specification.

The `PERSON` in Figure 5.2 is a type specification. The **String** after the **is** indicates that `PERSON` incorporates the specification of the **String** type. If required, other type specifications could be incorporated along with the **String** type. Further, the **eqns** operator indicates equations used to specify constraints relating intrinsic operations. Equations can be complex since they can specify complex constraints. However, only a set of simple equations required to provide meaningful type specifications such as those presented in Figure 5.2 are considered in this chapter. Table 5.2 provides a brief explanation of the type operators in Figure 5.2.

Operator	Description
sorts	Specifies *data carriers* of a type.
opns	Specifies the intrinsic operations that can be performed over variables of a type.
eqns	Begins the specification of *equations* where constraints relating intrinsic operations are specified.
forall	Declared under an **eqns** operator, it specifies free variables used in equations.
ofsort	Declared under an **eqns** operator, it specifies the outermost operation in its following equations.

Table 5.2: Some type operators of LOTOS.

Thus, for a given variable $aPerson$ of the type `Person`, for example, the operation `getname(aPerson)` returns a value of type **String** that is stored in `Person`. From the definition of `getname(aPerson)` it is possible to identify the declarative nature of LOTOS. The specification of the `getname()` operation specifies *what* is wanted rather than *how* the value can be retrieved from `aPerson`. Two special operations are considered in the type specifications presented in this chapter. In the case of the type `Person`, the `mk_Person` operation is a default *construct* that does not require any parameter to create a value of type `Person`. The `mk_Person2` operation is also a construct, but one that requires the constituent values (attributes) of `Person` to create a composite value of type `Person`. Therefore, using `mk_Person2`, it is possible to see in Figure 5.2 that, for example, the `getname()` operations can be used to get an attribute of type **String** from `Person`, and the `setname()` operation can be used store a **String** value as an attribute in `Person`. The **String** type is presented as a reserved word in Figure 5.2 since it is a primitive type of LOTOS.

The LOTOS processes in this chapter are built in an incremental way and using mainly the basic constructs of LOTOS. Thus, it may be expected that readers without prior experience of LOTOS can understand the LOTOS notation from the description of LOTOS presented in this section. Nevertheless, Bolognese and Brinksma [16] is a suggested introductory paper on LOTOS.

A semantics for UML*i* models can be provided by LOTOS specifications generated from these UML*i* models. Starting in this section, we explain how UML models can be translated into LOTOS specifications.

5.3 From UML*i* Models to LOTOS Specifications: Foundations

The strategy of this LOTOS-based proposal for a UML*i* semantics follows the *core meta-modelling* strategy described in [37]. Basically, the idea is to provide an initial semantics for some UML*i* constructs considered essential for modelling most UML*i* diagrams. The semantics specified for these constructs can then be used as a framework for specifying a semantics for the other constructs. For example, a formalisation for Class might be required in order to formalise Package. This chapter is organised to present the set of constructs specific to UML*i* with respect to UML. However, the original UML constructs required to support the definition of the UML*i* specific constructs and the UML*i* to LOTOS mapping example in this chapter are also presented. Appendix A presents some core UML constructs not introduced in this chapter. The following definitions are required to explain how a semantics for such UML*i* constructs can be specified.

LOTOS has a context-free grammar. According to [1], a context-free grammar has a set of *terminal symbols*; a set of *non-terminal symbols*; a set of *productions* where each production is composed of a non-terminal symbol, an arrow, and a sequence of terminals and/or non-terminals; and a designation of one of the non-terminals as the *start symbol*. Thus, a non-terminal symbol of LOTOS is a LOTOS construct. Assuming that $\mathcal{U}$ is a UML*i* construct and that $\mathcal{L}$ is a LOTOS construct, a semantics for UML*i* can be provided by the LOTOS semantics contained in the specification generated through the use of $\Phi(\mathcal{U}) = \mathcal{L}$. The Φ function is specified through the definition of *UMLi construct definitions (UCDs)*, which are definitions of UML*i* constructs in terms of sets of at least one LOTOS construct.

From this UCD we can see that our approach conforms with the official UML approach for the UML part of UML*i*. In fact, we are respecting the UML specification in the sense that we are neither *modifying* nor *removing* any element of UML. Moreover, we are using the UML specification as a foundation for the proposed semantics for UML*i*.

5.3.1 The Activity Specification

From this point in the chapter we start to introduce a set of 45 UCDs consisting of the UML*i* specific constructs and the original UML constructs required to explain the translation of the Library System into LOTOS specifications. Thus, the Library System, or any system $\mathcal{S}$, can be specified from a top-level activity diagram. In the case of the Library System, for example, it can be specified from its top-level activity diagram as shown in Figure 3.3. The Activities in this top-level activity diagram can be recursively decomposed into less abstract Activities and ActionStates connected by Transitions and PseudoStates. The decomposition of the top-level activity diagram is considered complete when the Activities are entirely described in terms of ActionStates, which are the "leaves" of the tree of Activities. Thus, the UML*i* Activity construct can be specified as follows.

UCD 1 *An* Activity *that has subactivities* $Sub_1..Sub_X$ *is defined by a LOTOS process definition specified as follows.*

```
process ACTIVITY_ACT[..., abort] (...) : exit :=
   <final_activity_behaviour>
  where
   process SUB1_ACT[..., abort] : exit (...) :=
      ...
   endproc
      ...
   process SUBX_ACT[..., abort] : exit (...) :=
      ...
   endproc
endproc
```

The <final_activity_behaviour> *in the ACTIVITY_ACT process above is a text defined by a BNF grammar specified as follows.*

```
<final_activity_behaviour> ::= <activity_behaviour>
                                 ``[> abort; exit''
<activity_behaviour>     ::= <activity> |
                         <activity_behaviour> <operation> <activity>
<activity>               ::= <action_state_imp> | <activity_imp>
<action_state_imp>       ::= <CALL_SEND_ACTIONSTATE> |
                             <CREATE_ACTIONSTATE>
<activity_imp>           ::= <LOTOS PROCESS>
<operation>              ::= ``[]'' | ``|||'' | ``>>''
```

In the grammar above: <*CALL_SEND_ACTIONSTATE*> *is a non-terminal; and* <*LOTOS PROCESS*> *is the specification of a LOTOS process instantiation.*

Hereafter, underlined words in UCDs represent placeholders which vary according to the instance of the UML*i* construct represented by the UCD. For example, for an Activity named `SelectFunction` the `ACTIVITY_ACT` in UCD 1 is replaced to `SELECTFUNCTION_ACT`, as presented later in Figure 5.3.

From UCD 1 we can see that the processes of the subactivities, which can be ActionStates, are defined as subprocesses of the `ACTIVITY_ACT` process. Reuse [67] can be achieved in LOTOS specifications of UMLi models by defining Activities and ActionStates in common higher-level Activities of the Activities that share the same functionality. Still in UCD 1, `ACTIVITY_ACT` has a standard `abort` gate which is responsible for finishing the process. The `abort` gate allows any process synchronised to it to finish the `ACTIVITY_ACT` process at any time. Thus, such a gate may be useful for handling abnormal situations such as a premature destruction of an Object or an error message from the operating system. Further discussion about the `abort` gate is provided in Section 5.5.8.

To conform with the LOTOS specification, an additional UCD should be specified for top-level activities.

UCD 2 *An* InitialInteraction *identifies top-level* Activities *that have their* **process** *and* **endproc** *terminators and the first appearance of the := terminator in UCD 1 replaced by the* **specification***,* **endspec** *and* **behaviour** *terminators respectively. The* InitialInteraction *itself is not translated into LOTOS specifications.*

ActionStates and Classes should also be translated into LOTOS specifications in order to describe $\mathcal{S}$ as a LOTOS process. In fact, ActionStates specify the Objects where Actions are performed, and Objects are specified by their Classes. Classes can be specified in term of LOTOS processes, as explained in Section 5.4.1. Actions can be specified in terms of LOTOS processes as discussed in Section 5.5.3. However, there are two mapping techniques between $\mathcal{U}$ and $\mathcal{L}$, which we are calling *foundation mappings*, that need to be presented before the introduction of the Class and Object constructs. The first foundation mapping technique presented in Section 5.3.2 explains how the connections provided by Transitions and PseudoStates in activity and statechart diagrams can be translated into binary operators of LOTOS. These binary operators are used to compose the behaviour expressions of processes. The second foundation mapping technique presented in Section 5.3.3 explains how types, implicitly specified by Classes in UMLi, can be specified by LOTOS *primitive types* and *type specifications*.

Concerning the behaviour of $\mathcal{S}$, a translation of interaction diagrams, viz. *sequence* and *collaboration* diagrams, into LOTOS specifications is not described in this chapter. Indeed, despite the fact that interaction diagrams may often be felt to be more useful for developers using UML*i* than activity and statechart diagrams, we can observe that interaction diagrams are partial representations of activity and statechart diagrams [105]. Nevertheless, the semantics provided in this chapter for constructs used in both activity and statechart diagrams can be used for specifying a semantics for constructs used to build interaction diagrams.

5.3.2 The Transition and PseudoState Specifications

Activity and statechart diagrams are composed of instances of StateVertex connected by instances of Transition. Recalling Figure 4.12, PseudoState and State are subclasses of StateVertex. Further, Branch, Fork, Join, InitialState and FinalState are categories of PseudoState. Thus, a specification of Transition, Branch, Fork, Join, InitialState and FinalState in terms of LOTOS operators provides the foundation required to map UML*i* models describing behavioural aspects of software systems into LOTOS specifications.

Let A, B, C and D be Activities and a and b be States. Table 5.3 presents the mapping of Transition, Fork, Join, Branch, InitialState and FinalState into LOTOS behaviour expressions. A State can be defined either as *visited* or *non-visited* with respect to its immediate superstate. Thus, a State that can only be reached once during any execution of its immediate superstate is defined as a *non-visited* State. Otherwise it is defined as a *visited* State, even if it is not reached during a particular execution of its immediate superstate. Therefore, mapping of a visited State into LOTOS expressions requires a recursive invocation of the generated LOTOS process, as described by the UCD 6 in Table 5.3.

The LOTOS specification in Figure 5.3 for the top-level activity diagram in Figure 3.3 is produced using the UCDs already presented. The InitialInteraction identifies the top-level activity diagram mapped as the `LIBRARY_ACT` specification (UCD 2). Then, a navigation through the activity diagram should be performed. The navigation starts at the InitialInteraction that acts as an InitialState for top-level activity diagrams. The InitialInteraction itself is not mapped into the behaviour expression of `LIBRARY_ACT` (UCD 2). Following the Transition leaving the InitialInteraction, which is also not mapped in `LIBRARY_ACT` (UCD 5), the `connect` Activity is reached, which is mapped as the `CONNECT_ACT` process (UCD 1). The Transition leaving the `connect` Activity, mapped as >> (UCD 4), reaches the `SelectFunction` Activity, mapped as `SELECTFUNCTION_ACT` (UCD 1).

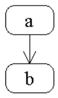
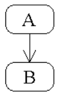
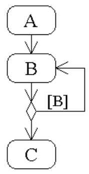
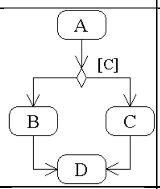
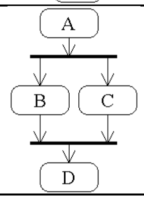
UCD	$\mathcal{U}$	$\Phi(\mathcal{U})$	a generic example	
			UMLi	LOTOS
3	Transition between two non-PseudoStates to a non-visited State in a statechart diagram	action prefix		a;b
4	Transition between two non-PseudoStates to a non-visited State in an activity diagram	enabling		$A >> B$
5	Transition to or from a PseudoState going to or coming from a non-visited State	considered as part of the PseudoState	—	—
6	Transition from a Branch to a visited State	recursive process		A >> REC_ACT **where** **process** REC_ACT[abort]: **exit** := B >> (C [] [B]REC_ACT) **endproc**
7	Branch	choice		$A >> (B[][C]C) >> D$
8	Fork and Join	interleave with parenthesis		$A >> (B\|\|\|C) >> D$
9	InitialState	not mapped	•	—
10	FinalState	not mapped	⊙	—

Table 5.3: UCDs related to Transition, Branch, Fork, Join, InitialState and FinalState constructs.

Finally, following the Transition leaving the SelectFunction Activity, which is not mapped in LIBRARY_ACT (UCD 5), a FinalState is reached, which is also not mapped in LIBRARY_ACT (UCD 10).

To facilitate the identification of the roles that LOTOS processes are playing in the specification of UML*i* constructs, process names are suffixed by "_ACT" if the processes are modelling Activities, by "_AS" if they are modelling ActionStates, and by "_CLS" if they are modelling Classifiers or their subclasses, i.e., Classes and Interfaces.

```
specification LIBRARY_ACT[abort] : exit
  behaviour
    CONNECT_ACT[abort] >> SELECTFUNCTION_ACT[abort]
    [> abort; exit
  where
    process CONNECT_ACT[abort] : exit :=
      (* CONNECT_ACT specification *)
    endproc
    process SELECTFUNCTION_ACT[abort] := exit :=
      (* SELECTFUNCTION_ACT specification *)
    endproc
endspec
```

Figure 5.3: The LOTOS specification of the Library Activity.

5.3.3 Type Mappings

Both UML*i* and LOTOS provide a set of *primitive types* and allow the specification of *complex types* from these primitive types. Primitive types in UML*i* are specified by the constructs in the UML DataType package. Primitive types in LOTOS are provided along with the specification of LOTOS. Complex types in UML*i* are specified by the specification of Classes, where their Attributes can be primitive types, defined in terms of elements of DataType, or other Classes. In the same way, complex types in LOTOS can be specified by a type specification, as introduced in Section 5.2. Further, these type specifications can be composed of primitive types or other complex types.

There is almost a complete match between the primitive types of UML*i* and LOTOS. For example, Boolean matches with **Bool**, and Integer matches with **nat**. Few primitive types of UML*i* do not match with any primitive type of LOTOS. In this case, we can use a LOTOS specification type to define these non-matching types. For example, Enumerate can map to a LOTOS type definition as in Figure 5.4. There, an `element` can be retrieved from the enumeration using the `getnext` operation. In fact, `element` is a parameterised **sort** specified by the **formalsort** operator, and `getnext` is a parameterised **opn** specified by the **formalopn** operator.

```
type ENUMERATE is
   formalsorts element
   formalopns getnext: -> element
   sorts Enumerate
   opns hasnext: -> Bool
endtype
```

Figure 5.4: `Enumerate` type specification.

Further, the `Enumerate` type can be used as the Enumerate to specify different enumerations. For instance, the ENUMERATE_BOOKCOPY type definition in Figure 5.5 can be specified from the BOOKCOPY type definition. The **actualizedby** and **using** operators define that BOOKCOPY should provide the parameters specified in Figure 5.4. Then, the **sortnames** and **for** operators specify that `BookCopy` is the value for the `element` sort parameter. The **opnames** and **for** operators specify that the `nextBookCopy` is the value for the `getnext` operation parameter.

```
type ENUMERATE_BOOKCOPY is
   Enumerate actualizedby BOOKCOPY using
      sortnames BookCopy for element
      opnames nextBookCopy for getnext
endtype
```

Figure 5.5: Specification of `EnumerateBookCopy` from the `Enumerate` type.

Doing these type mappings as presented here, a complete translation of the types specified by Classes into LOTOS type specifications can be achieved.

5.4 Semantics for Structural Aspects of UML*i*

5.4.1 The Classifier Specification

An informal definition of Classifier may be appropriate for readers not familiar with the UML*i* metamodel terminology. Class is a common term when modelling and implementing object-oriented software systems. In terms of the UML metamodel, however, it is common to use Classifier rather than Class in certain circumstances. In fact, Classifier is the construct that has StructuralFeatures such as Attributes, and BehavioralFeatures such as Operations. Class is a specialisation of Classifier that specifies that it can be instantiated into an Object. The main reason for the distinction between Class and Classifier is that there are other constructs that are Classifiers other than Class such as the Interface construct.

Considering that Class is the major construct for specifying structural aspects of software systems in UML, and that the Classifier is a generalisation of Class, a UML construct definition for Classifier can provide a semantics for many structural aspects of UML. In this section, the features of the `BookCopy` Class in Figure 3.2 are gradually translated into LOTOS in order to introduce a generic LOTOS specification for Class, Classifier, and their related constructs.

Classifier is frequently used as a type specification in UML since it plays the `type` role several times in the UML metamodel. In fact, a Classifier is an implicit definition of type in the UML context. In LOTOS, types are explicitly declared. The `BookCopy` class in Figure 3.2 has the attributes `status` and `copyCode`. Thus, a `BookCopy` type can be specified as in Figure 5.6. This means that `BookCopy` is the type of the `BookCopy` class.

```
type BOOKCOPY is enum, String
  sorts BookCopy
  opns  mk_BookCopy:                     -> BookCopy
        mk_BookCopy2: enum, String       -> BookCopy
        setstatus: BookCopy, enum        -> BookCopy
        getstatus: BookCopy              -> enum
        setcopycode: BookCopy, String    -> BookCopy
        getcopycode: BookCopy            -> String
  (* eqns specification *)
endtype
```

Figure 5.6: `BookCopy` type specification.

A Classifier can specify behaviour in addition to the type specification, as indicated by the `BOOKCOPY_CLS` process in Figure 5.7. There, the `copyCode` attribute of `BookCopy` is defined as a pair of low-level operations defined in the type definition, viz., `setcopycode` and `getcopycode`, that are used by a pair of observable actions, viz., `cci` and `cco`, to update and retrieve the current value of the attribute. Moreover, the `BOOKCOPY_CLS` process specifies that the attributes of the `BookCopy`'s instance are ready to be updated, e.g., $cci?new\_cc$: **String**, or retrieved, e.g., $cco!getcopycod(bc)$, while the process is active. The same strategy is used to specify the `status` attribute of `BookCopy`.

```
process BOOKCOPY_CLS[ si , so , cci , cco , destroy_bookcopy ]( bc:BookCopy ) :
  exit :=
  ( si ?new_status:enum ;
        BOOKCOPY_CLS[ si , so , cci , cco , destroy_bookcopy ]
                    ( setstatus (bc, new_status ))        []
    so ! getstatus (bc );
        BOOKCOPY_CLS[ si , so , cci , cco , destroy_bookcopy ]
                    (bc )                                 []
    cci ?new_cc:String ;
        BOOKCOPY_CLS[ si , so , cci , cco , destroy_bookcopy ]
                    ( setcopycode (bc, new_cc ))          []
    cco ! getcopycode (bc );
        BOOKCOPY_CLS[ si , so , cci , cco , destroy_bookcopy ]
                    (bc )  )
  [> destroy_bookcopy ; exit
endproc
```

Figure 5.7: Specification of Attributes in the `BookCopy` process.

A generic Classifier type and process can be defined from the `BOOKCOPY` type and `BOOKCOPY_CLS` process examples.

UCD 11 *A* Classifier *is defined by the specification of a LOTOS type definition and a LOTOS process definition. The type definition is specified as follows.*

```
type CLASS is
    sorts Class
    opns  mk_class  -> Class
endtype
```

The process definition is specified as follows.

```
process CLASS_CLS[ destroy_class] ( c: Class) : exit :=
    i ; CLASS_CLS
        (* this internal action i represents
           a class with an underspecified behaviour *)
    [> destroy_class; exit
endproc
```

The `destroy_class` gate in UCD 11 corresponds to the `abort` gate of `ACTIVITY_ACT` in the context of Classifiers as discussed in Section 5.5.8.

Hereafter, the `CLASS` and `CLASS_CLS` names denote the generic type and process definitions, respectively, of a Classifier. Complete specifications of `CLASS` and `CLASS_CLS` are presented in Section A.5. Considering `CLASS` and `CLASS_CLS`, it is possible to define the mappings for Object, Class and Attribute.

UCD 12 *An* Object *is a LOTOS process variable of the CLASS type and is used by an instance of the* `CLASS_CLS` *process.*

UCD 13 *A* Class *is a* Classifier *that can be used by a* CreateAction ActionState *(UCD 23) to specify* Objects *in LOTOS processes.*

UCD 14 *An* Attribute `Attr` *of type* `AttrType` *is specified by a pair of type operations in* `CLASS`*, e.g.,* `setAttr` *and* `getAttr`*, and a pair of observable actions, e.g.,* `attri` *and* `attro`*, in* `CLASS_CLS`*. The* `CLASS` *type with an* `Attr` *attribute is specified as follows:*

```
type CLASS is AttrType
   sorts Class
   opns  mk_Class:                     -> Class
         mk_Class2: AttrType           -> Class
         setAttr: Class, AttrType      -> Class
         getAttr: Class                -> AttrType
   (* eqns specs *)
endtype
```

The `CLASS_CLS` *process with an* `Attr` *attribute is specified as follows:*

```
process CLASS_CLS[attri, attro, destroy_class](c: Class) : exit :=
  (attri?new_attr:AttrType;
     CLASS_CLS[attri, attro, destroy_class]
                        (setAttr(c,new_attr)) []
   attro!getAttr();
     CLASS_CLS[attri, attro, destroy_class](c)  )    []
  ... ) [> destroy_class; exit
endproc
```

The pair consisting of the `BOOKCOPY` type and the `BOOKCOPY_CLS` processes provide an incomplete specification of the BookCopy Class in Figure 3.2. For instance, the Operations of BookCopy are not specified in Figure 5.7.

5.4.2 The InteractionClass Specification

The semantics of the InteractionClass is based on the LOTOS specification of the Abstraction-Display-Controller (ADC) interactors [82, 83]. Figure 5.8 provides a schematic view of the ADC interactor that may be useful to visualise its top-level structural components before discussing other features. There, an ADC process is represented as a big box (light shadowed area). The top of the box is its *abstraction side*, the bottom of the box is its *display side* and the right side

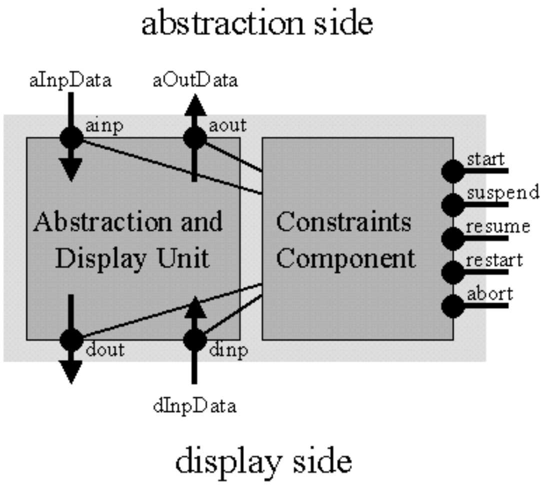


Figure 5.8: An schematic view of the ADC interactor [83].

of box is its *controller side*. The abstraction side is where the communication between the interactor and other objects, which may be domain objects or other interactors, can occur. The display side is where the communication between the interactor and system users can occur. In other words, the interaction between objects of the user interface and other objects of the application can only occur on the abstraction side of interactors. The interaction between users and objects of the user interface can only occur on the display side of interactors. The controller side is responsible for the handling of some specific actions provided by the environment which comprehends domain, interactors and users among other agents.

Figure 5.8 still shows the *Abstraction and Display Unit* (ADU) and *Constraints Component* (CC) processes represented as the two boxes within the ADC process, process gates represented as small black discs, and internal compositions represented as thin lines connecting the CC process to the ADU gates. The ADU and CC processes are represented within the ADC interactor since the interactor is the result of their parallel composition, as presented in Figure 5.9. There, G_{io} is the ordered set of gates $\{$`intClass`, `dinp`, `dout`, `ainp`, `aout`$\}$ and $G = G_{io} \cup G_{cc}$ where G_{cc} is the ordered set of gates $\{$`start`, `suspend`, `restart`, `resume`, `abort`$\}$. The ADU part of the ADC interactor is responsible for handling the operations that can affect the state of the interactor. The CC part of the ADC is responsible for constraining the ADU operations allowing them to be performed according to a set of temporal dependencies between operations.

```
process ADC[G](IC:intClass, A:abs, D:disp): noexit :=
  ADU[G_io](IC,A,D,D) |[G_io]| CC[G]
endproc
```

Figure 5.9: The ADC interactor.

The ADC interactor requires a particular data structure called *Abstraction and Display* (`ad`) type for supporting the specification of the ADU and CC processes. Thus, the `ad` type is presented in the following section in order to introduce the ADU and CC processes.

The Abstraction and Display Type

The `ad` type is an abstract specification of the data structure required to support the ADC interactor functionalities. Figure 5.10 shows the `ad` type specification. There, the `ad` type has six **sorts**. The `intClass`, `abs` and `disp` **sorts** are internally referred in an ADC interactor.

```
type ad is
   sorts intClass, abs, disp, dInpData, aInpData, aOutData
   opns  mk_ad:                               -> intClass, abs, disp
         mk_ad2:   intClass                   -> intClass, abs, disp
         input:    dInptData, disp, abs       -> abs
         echo:     dInpData, disp, abs        -> disp
         render:   disp, aInptData            -> disp
         receive:  abs, aInpData              -> abs
         result:   abs                        -> aOutData
         getContainer: intClass               -> intClass
         getStatus:  intClass                 -> abs, disp
         isInside:  disp, disp                -> Bool
         hasOverlapping: disp, disp           -> Bool
   forall a,b,c:intClass, aa,ba,ca: abs, ad,bd,cd: disp
         getContainer(a) eq c and getStatus(a) eq aa, ad and
         getStatus(c) eq ca, cd
             => isInside(ad,cd) eq true;
         getContainer(a) eq c and getStatus(a) eq aa, ad and
         getStatus(c) eq ca, cd
             => isInside(cd,ad) eq false;
         getContainer(a) eq c and getStatus(a) eq aa, ad and
         getContainer(b) eq c and getStatus(b) eq ba, bd
             => hasOverlapping(ad,bd) eq false;
endtype
```

Figure 5.10: The `ad` type.

- The `intClass` is the unique identification of an interactor. It corresponds to the `Class` **sort** in UCD 11.

- The `disp` is the **sort** carrying the state of the visual side of the ADC interator.
- The `abs` is the **sort** carrying the state of the abstract side of the ADC interactor.

In addition to the internal **sorts** above, the `ad` type in Figure 5.10 specifies three other **sorts** to specify data items that interactors may exchange with users and other objects.

- The `dInpData` is the **sort** carrying raw data received from users when interacting with the display side of the ADC interactor.
- The `aInpData` is the **sort** carrying data received from objects interacting with the abstract side of the ADC interactor. Data item in `aInpData` is in the format used by the object interacting with the abstract side of the ADC interactor.
- `aOutData` is the **sort** carrying data to be sent to objects interacting with the abstract side of an ADC interactor. Data item in `aOutData` is in the format used by the object interacting with the abstract side of the ADC interactor.

The `ad` type specifies eleven operations used to manipulate the **sorts** introduced above.

- The `mk_ad` operation instantiates a new interactor which is not contained by any other interactor. Three variables of type `intClass`, `abs` and `disp`, which characterise the internal state of the newly created interactor, are produced as the result of invoking the operation.
- The `mk_ad2` operation instantiates a new interactor contained by the interactor which provides the `intClass` parameter. Like the `mk_ad`, the `mk_ad2` operation also produces three variables carrying the internal state of the newly created interactor.
- The `input` operation computes the updated data in `abs` by interpreting current data in `dInpData` with respect to current data in `disp` and `abs`.

- The `echo` operation computes the updated data in `disp` by interpreting current data in `dInpData` with respect to current data in `abs` and `disp`.
- The `render` operation computes the updated data in `disp` by interpreting current data in `aInpData`. This computation is the process of rendering the presentation of the interactor.
- The `receive` operation computes the updated data in `abs` by interpreting current data in `aInpData` and performing required transformations to put the data in a suitable format for the interactor.
- The `result` operation computes the updated data in `aOutData` with respect to current data in `abs`, performing any required transformation to deliver the data in a suitable format for objects interacting with the abstract side of the interactor.
- The `getContainer` operation returns the `intClass` of the interactor which contains the current interactor.
- The `getStatus` operation returns the `disp` and `abs` of the `intClass` used as parameter.
- The `isInside` operation returns `true` if the second of its `disp` parameter is within the first `disp` parameter. Otherwise, the operation returns `false`. The notion of being "within" is intensionally abstract to make the operation useful for different kinds of presentations, e.g., two and three dimensional presentations.
- The `hasOverlapping` operation returns `true` if the two `disp` parameters are not overlapping. Otherwise, the operation returns `false`. Similarly with the `isInside` parameter, the notion of "overlapping" is intentionally abstract to make the operation useful for different kinds of presentations.

The `intClass` **sort** and `mk_ad`, `mk_ad2`, `getContainer`, `getStatus`, `isInside` and `hasOverlapping` operations are not specified in [83]. The `intClass` **sort** and `mk_ad` operation are required to make the ADC interactor a Class used to instantiate Objects that are uniquely identified. The `getContainer`, `getStatus`, `isInside` and `hasOverlapping` operations are required to create the notion of containment of interaction objects, abstracted in ADC interactors, that is required to support the specification of the `mk_ad2` operation.

From the `ad` type it is possible to describe the `ADU` and `CU` processes that compose an ADC interactor.

The Abstraction-Display Unit (ADU) Process

The `ADU` process specifies the relationships between `ad` operations and actions resulting from the interaction of objects and users with ADC interactors. The specification of the `ADU` process in Figure 5.11 shows these relationships. For instance, an action on gate `ainp` produces an `x` variable of `aInpData` that is followed by the performance of the `receive` and `render` operations. As can be observed in Figure 5.11, the `ADU` process is recursively instantiated for every interaction performed on its gates.

```
process ADU[Gio](ic:intClass, a:abs, dc,ds:disp) : noexit :=
  ( dinp?x:dInpData;    ADU[Gio](ic, input(x,ds,a), echo(x,ds,a), ds) []
    dout!dc;            ADU[Gio](ic, a, dc, dc)                        []
    ainp?x:aInpData;    ADU[Gio](ic, receive(a,x), render(dc,x), ds)   []
    aout!result(a);     ADU[Gio](ic, a, dc, ds) )
endproc
```

Figure 5.11: The `ADU` process.

Four local variables carry the current state of the interactor every time the `ADU` process is recursively instantiated.

- The `ic` state parameter of type `intClass` is the unique identifier of the current interactor.
- The `a` state parameter of type `abs` is the current state of the abstraction of the interactor.
- The `ds` state parameter of type `disp` is the current state of the display of the interactor.
- The `dc` state parameter of type `disp` is an auxiliary state carrying the latest value computed for the display of the interactor.

A description of the actions recognised by an `ADU` process can explain the relationships between actions, named after the gate where they can occur, and operations. Thus, these are the four actions recognised by an `ADU` process:

- An interaction on gate `dinp` receives data of type `dInpData` that is used to update the `a` state parameter by performing the `input` operation and to compute the `dc` state parameter by performing the `echo` operation.
- An interaction on gate `dout` produces data of type `disp` and update `ds` with the data of `dc`.
- An interaction on gate `ainp` receives data of type `aInpData` that is used to update the `a` state parameter by performing the `receive` operation and to compute the `dc` state parameter by performing the `render` operation.
- An interaction on gate `aout` produces data of type `aOutData` by performing the `result` operation over the `a` state parameter.

The `CC` process is the part of the ADC interactor responsible for specifying temporal constraints between its operations, as described in the following section.

The Constraints Component (CC) Process

From the description of the `ADC` process presented so far, there is no restriction on the way that ADC interactors can react to interactions on their gates. For example, interactors would be ready to interact with users on gate `dinp` once they are instantiated. Operations, however, may need to be constrained in order to become useful. In fact, it may be necessary that an instantiated interactor may not be ready to interact with users on gate `dinp`, for example. Recalling Figure 5.11, the `CC` process is synchronised with the `ADU` process by their common gates G_{io}. Therefore, the `CC` process has control of the `ADU` process since any condition specified in its behaviour expression is also a condition on the execution of the `ADU` process. Figure 5.12 presents the `CC` process where the behavioural pattern of the `ADC` process is specified.

A description of the five gates that the `CC` process has in addition to those in the `ADU` process, viz., `start`, `suspend`, `resume`, `restart` and `abort`, can describe the `CC` process itself. These gates are derived from the GARNET [94] project, where they are used to describe user interface dialogs. They are also called SSRRA gates and can be described as follows:

- An interaction on gate `start` is required to enable the ADC interactor by instantiating the `RUN` process in Figure 5.12. No other action is recognised before this interaction. Once the interaction is performed, the parallel

```
process CC[start, suspend, resume, restart, abort, dinp, dout, ainp, aout] : noexit :=
  start; RUN[suspend, resume, restart, abort, dinp, dout, ainp, aout]

where
  process RUN[suspend, resume, restart, abort, dinp, dout, ainp, aout] : noexit :=
       (     CU[dinp, dout, ainp, aout]
                   |[dinp, dout, ainp, aout]|
             SU_RE[suspend, resume, dinp, dout, ainp, aout])
    [> INTERRUPT[suspend, resume, restart, abort, dinp, dout, ainp, aout]
  endproc

  process CU[dinp, dout, ainp, aout] : noexit :=
    ainp?X:aInpData;     CU[dinp, dout, ainp, aout] []
    aout?X:aOutData;     CU[dinp, dout, ainp, aout] []
    dout?X:disp;         CU[dinp, dout, ainp, aout] []
    dinp?X:dInpData;     CU[dinp, dout, ainp, aout]
  endproc

  process SU_RE[suspend, resume, dinp, dout, ainp, aout] : noexit :=
    ANYORDER[dinp, dout, ainp, aout]
    [> suspend; resume; SU_RE[suspend, resume, dinp, dout, ainp, aout]
  endproc

  process ANYORDER[dinp, dout, ainp, aout]: noexit :=
    ainp?X:aInpData;     ANYORDER[dinp, dout, ainp, aout] []
    aout?X:aOutData;     ANYORDER[dinp, dout, ainp, aout] []
    dout?X:disp;         ANYORDER[dinp, dout, ainp, aout] []
    dinp?X:dInpData;     ANYORDER[dinp, dout, ainp, aout]
  endproc

  process INTERRUPT[suspend, resume, restart, abort, dinp, dout, ainp, aout] :
                                                                        noexit :=
    restart; RUN[suspend, resume, restart, abort, dinp, dout, ainp, aout] []
    abort; stop
  endproc

endproc
```

Figure 5.12: The `CC` process.

composition of the `CU` and `SU_RE` process is instantiated to allow either the normal execution of the `ADU` operations due to the `ANYORDER` process within the `SU_RE` process or the suspension of the ADU operations.

- An interaction on gate `suspend` halts the `ADU` process that is synchronised with the `SU_RE` process by waiting for an interaction on gate `resume`.
- An interaction on gate `resume` re-instantiates the `SU_RE` process, and consequently the `ANYORDER` process, re-enabling the `ADU` functionalities.
- An interaction on gate `restart`, that can happen anytime after the `RUN` process is instantiated, is used to reset the interactor to its initial state.
- An interaction on gate `abort` issues a **stop** operation that finishes the entire

ADC interactor.

The InteractionClass as an ADC Interactor

The InteractionClass definition is derived from the ADC interactor specification as presented in this section. The basic InteractionClass construct is a constrained version of the ADC interactor. The **hide** operation of LOTOS is used to constrain some ADC functionalities by restricting its interaction with other objects on some gates. For instance, hiding the `ainp` gate would prevent an ADC interactor from interacting with other objects on this gate, which eventually prevents the performance of the `receive` and `render` operations.

UCD 15 *An* InteractionClass *is an ADC interactor that, by default, has the* `ainp` *and* `aout` *gates hidden, specified as follows:*

```
process ADC[G - {ainp,aout}](IC:intClass, A:abs, D:disp): noexit :=
  hide ainp, aout in
     (ADU[G_io](IC, A, D, D)  |[G_io]|  CC[G])
endproc
```

There, the `intClass`*,* `abs` *and* `disp` **sorts** *are specified in the* `ad` *type in Figure 5.10. The* `ADU` *process is specified in Figure 5.11. The* `CC` *process is specified in Figure 5.12.*

Thus, from the ADC specification in this chapter it can be observed that an InteractionClass is a specialised Class as defined in UCD 13, and consequently, a specialised Classifier as defined in UCD 11. Indeed, in terms of type specification, the `ad` type of the ADC interactor specifies an `intclass` **sort** which corresponds to the `Class` **sort**. Moreover, the `mk_ad` operation of `ad` type corresponds to the `mk_class` operation of Class. In terms of process specification, an InteractionClass specifies an `abort` gate in its G_{cc} set of gates with the same functionality as the `abort` gate in the Class specification.

According to Figure 4.15, there are seven UML*i* constructs derived from the InteractionClass. These constructs are presented in the following sections.

5.4.3 The PrimitiveInteractionClass Specification

Trees of InteractionClasses can be built using the parallel composition of InteractionClasses. In fact, in addition to the parallel composition, ADC processes that specify InteractionClasses can be hierarchically organised since some of them can be contained by others depending on whether they are instantiated by the `mk_ad` or `mk_ad2` operations. PrimitiveInteractionClasses along with ActionInvokers are the leaves of these trees of InteractionClasses. The PrimitiveInteractionClass is specified in this section and the ActionInvoker is specified in the next section.

UCD 16 *A* PrimitiveInteractionClass *is an* InteractionClass *to* Objects *of which are instantiated using the* `mk_ad2` *operation only. The* `intClass` **sort** *of instances of* PrimitiveInteractionClass *cannot be used as the parameter of* `mk_ad2` *operations.*

An instance of a PrimitiveInteractionClass is always a leaf on trees of InteractionClasses since it must be contained by another InteractionClass, and at the same time it cannot contain any other InteractionClass. A PrimitiveInteractionClass must be contained by another InteractionClass since it cannot be instantiated by the `mk_ad` operation. A PrimitiveInteractionClass must not contain any other InteractionClass since its `intclass` **sort** cannot be used as the parameter for any `mk_ad2` operation.

From the definition of InteractionClass (UCD 15) it can be observed that it does not need to have the `ainp` and `aout` gates hidden. In fact, UML*i* constructs derived from PrimitiveInteractionClass are specified revealing one or both of the `ainp` and `aout` gates of the InteractionClass. Thus, the UML*i* constructs derived from PrimitiveInteractionClass are specified as follows.

UCD 17 *A* Displayer *is a* PrimitiveInteractionClass *with the* `ainp` *gate not used.*

Having the `ainp` gate visible, Displayers can propagate inputs from domain objects and other InteractionClasses to users. However, inputs from users are not propagated to any other object by Displayers since their `aout` gates are hidden.

UCD 18 *An* Inputter *is a* PrimitiveInteractionClass *with the* `aout` *gate not used.*

Having the `aout` gate visible, Inputters can propagate inputs from users to domain objects and other InteractionClasses. However, inputs from domain objects or other InteractionClasses are not recognised by Inputters since their `ainp` gates are hidden.

UCD 19 *An* Editor *is a* PrimitiveInteractionClass *with both the* `ainp` *and* `aout` *gates not used.*

An Editor can propagate inputs both from users to domain objects and other InteractionClasses, and from domain objects and other InteractionClasses to users.

5.4.4 The ActionInvoker Specification

An ActionInvoker is a special kind of InteractionClass that does not receive or send any data item through its `dinp` and `ainp` gates.

UCD 20 *An* ActionInvoker *is an* InteractionClass *with the* `aout` *gate visible and the ADU process specified as follows:*

```
process ADU[G_io]( ic:intClass , a:abs , dc, ds:disp ) : noexit :=
  ( dinp?x:dInpData;  ADU[G_io]( ic , a, echo(x, ds , a), ds ) []
    dout!dc;          ADU[G_io]( ic , a, dc , dc )             []
    ainp;             ADU[G_io]( ic , a, dc , ds )             []
    aout;             ADU[G_io]( ic , a, dc , ds ) )
endproc
```

The `CU` *and* `ANYORDER` *subprocesses of* `CC` *if Figure 5.12 are specified according to the ADU specification above.* Objects *of* ActionInvoker *are instantiated using the* `mk_ad2` *operation only. The* `intClass` **sort** *of instances of* ActionInvoker *cannot be used as the parameter of* `mk_ad2` *operations.*

The restrictions concerning ActionInvoker's instantiation and use of the `int-class` **sort** are the same as those of the PrimitiveInteractionClass, as discussed in Section 5.4.3. The ActionInvoker is very similar to an Inputter since it has the `aout` gate visible and the `ainp` gate hidden. However, it cannot propagate information to other objects other than the `aout` signal itself.

ActionInvokers are useful since they allow users to navigate through interactive applications by modifying the application control-flow according to interactions on their `aout` gates that do not require the exchange of any particular data item such as a mouse click. In UML*i*, these ActionInvoker actions are interpreted according to the kind of interaction object flow the ActionInvoker is associated with, as discussed in Section 5.5.

The properties of the ADC interactor presented so far describe aspects of a generic widget. However, in the same way that widgets can be grouped to build user interfaces, ADC interactors can be combined to build UI specifications. In the next section it is discussed how UI specifications can be constructed by composing ADC interactors.

5.4.5 The Container Specification

There are many different ways of composing ADC interactors. These variations of compositions, however, are of one of the two distinct forms named as the *distributed form* (DF) and the *compound form* (CF) in [82]. For the $ADC_A = ADU_A|[G^A_{io}]|CC_A$ and $ADC_B = ADU_B|[G^B_{io}]|CC_B$ interactors, the DF composition is a parallel composition defined as follows:

$$DF = (ADC_A|[G^A_{io}]|CC_A)|[G_{comp}]|ADU_B|[G^B_{io}]CC_B) \tag{5.1}$$

where $G_{comp} \subseteq G^A \cap G^B$.

The CF composition is defined as follows:

$$CF = (ADC_A|[G^{AB}_{io}]|ADC_B)|[G^A_{io} \cup G^B_{io}]|CC_A|[G^{AB}]CC_B) \tag{5.2}$$

where $G^{AB}_{io} \subseteq G^A_{io} \cap G^B_{io}$ and $G^{AB} \subseteq G^A \cap G^B$.

In this section, we are using DF compositions to exemplify the construction of UI specifications using ADC interactors. The strong bisimulation equivalence of the DF and CF compositions is demonstrated in [82]. This means that the DF compositions in this section can be expressed in terms of CF compositions.

As presented in Equation 5.1, DF is a category of compositions since the set of synchronisation gates G_{comp} can vary from a full synchronisation ($G_{comp} = G_A \cup G_B$) to a pure interleaving ($G_{comp} = \emptyset$). Considering this variability of G_{comp}, Container can be defined as follows.

UCD 21 *A* Container *is an* InteractionClass *where the G_{comp} of any composition containing it is* {`start`, `restart`, `abort`}.

The specification of the G_{comp} for compositions containing Containers in UCD 21 means that for any Container the InteractionClasses contained by it and containing it must be started, restarted and aborted simultaneously. From a presentation point of view, the Container is an InteractionClass that visually can contain other InteractionClasses, as identified by the **eqns** in Figure 5.10. From a behavioural point of view, the Container is a grouping mechanism that causes a set of InteractionClasses to be initiated and destroyed in a synchronised way.

From UCD 21 we can also observe that there is no restriction on Objects of Containers being instantiated by the `mk_ad` or `mk_ad2` operation. Thus, the possibility of instantiating Container using the `mk_ad` operation means that a Container is not required to be contained by another InteractionClass. Further, the possibility of instantiating Container using the `mk_ad2` operation means that a Container can be contained by another Container.

UCD 22 *A* FreeContainer *is a* Container *that can be instantiated using the* `mk_ad` *operation only. A parallel composition of* InteractionClasses *cannot contain more than one* FreeContainer.

The restriction that a FreeContainer can only be instantiated by the `mk_ad` operation means that it cannot be contained by any other Container. The restriction that compositions of InteractionClass can have only one FreeContainer means that the FreeContainer is the top-level InteractionClass in any composition of InteractionClasses. From UCDs 21 and 22 it is observed that a FreeContainer is a *presentation unit* as defined in [14]. Indeed, the `start`, `restart` and `abort` of the FreeContainer of a composition of InteractionClasses corresponds to the `start`, `restart` and `abort` of all InteractionClasses immediately and non-immediately contained by the FreeContainer, respectively.

A further discussion about behavioural aspects of UML*i* completes the framework required to generate LOTOS specifications from UML*i* models.

5.5 Semantics for Behavioural Aspects of UML*i*

The UCDs of the behavioural constructs of UML*i* presented in this section along with the structural UCDs presented in the previous section are the mapping rules required to resume the translation of part of the Library System initiated in Section 5.3. Thus, the LOTOS translation of the `Connect` Activity in Figure 4.8 identified as the `CONNECT_ACT` process in Figure 5.3 is presented in full in this section.

5.5.1 The CreateAction ActionState Specification

CreateActions are performed to instantiate Classes. The specification of when actions creating objects can take place is required to identify when Objects are available. Thus, in the `Connect` Activity in Figure 4.8, `new UserQuery` is a CreateAction ActionState that creates the `ud` instance of Class.

UCD 23 *A* CreationAction ActionState *for a* Class *is specified as the* `CREATE_CLASS_AS` *process as follows:*

```
process CREATE_CLASS_AS [ abort ] : exit (Class) :=
    ( i ; exit (mk_class) )
    [> abort ; exit
endproc
```

A CreateAction ActionState *is invoked in its* Activity*'s behaviour expression (UCD 1) by the* <*CREATE_ACTIONSTATE*> *non-terminal specified as follows.*

```
<CREATE_ACTIONSTATE> ::=
    ‘‘ CREATE_CLASS_AS [ abort ] >> accept c: Class in (’’
    <activity_behaviour > ‘‘)’’
```

The definition of the types and processes of the modelled Classifiers are specified in the top-level Activity. Thus, the description of the structural part of software systems can be used throughout the specification of the behaviour of the software systems. For instance, CreateActions can be specified within any Activity.

5.5.2 The ObjectFlowState and ClassifierInState Specifications

UML provides the ObjectFlowState and ClassifierInState constructs to specify object flows, as discussed in Section 4.3.2. Thus, in the `Connect` Activity in Figure 4.8, the `uq` ClassifierInState[1] of type `UserQuery` is produced as a result of the `new UserQuery` CreateAction ActionState. ObjectFlowStates specify the incoming and outgoing of objects with respect to the scope of an ActionState. Furthermore, ObjectFlowStates provide the Object where the associated ActionState takes place, and optional Objects which can be used as parameters to the Action performed in an ActionState.

UCD 24 *A* ClassifierInState *is an* Object*, as specified by UCD 12.*

The UCD 24 is introduced since ClassifierInStates and Objects are distinct constructs in the UML metamodel.

UCD 25 *An incoming* ObjectFlowState *is a synchronisation of a* Signal *or the* Messages *of an* Operation *between an* ActionState *process and a* Classifier *process.*

[1] Figure 4.13 shows the ClassifierInState and ObjectFlowState constructs for building object flows in the UML*i* notation.

The UCDs defining Signal, Message and Operation are introduced in Appendix A.

UCD 26 *An outgoing* ObjectFlowState *is the passing of a* ClassifierInState *to an* ActionState*.*

5.5.3 The CallAction and SendAction ActionState Specifications

In Figure 4.8, the `uq.checkUser()` is a CallAction ActionState where the CHECK_USER operation of a `UserQuery` Object instantiated in `new UserQuery` is invoked. This CHECK_USER operation is defined in the USERQUERY_CLS. No parameter is passed to this operation, i.e., the `getBookCopy` action is not followed by any variable. The result of the operation is the `cu` Object of the `LibraryUser` Class. In fact, this `cu` should be an Object either previously instantiated in the current session or produced as the result of a query submitted to an associated database system. The exactly specification of how the Method of an implementation could be implemented is under-specified during the design.

A SendAction ActionState has a LOTOS specification quite similar to a CallAction ActionState. The main differences between these two categories of ActionStates are those differences between SendActions and CallActions. For instance, if the associated Action is a SendAction, the specification of the ActionState does not includes the *CallInvoker* Message. With respect to their ActionStates, ObjectFlowStates and `ClassifierInStates` are defined in the same way.

UCD 27 *An* ActionState *performing a* CallAction *or a* SendAction *that has incoming object flows* OF_IN$_1$ *...* OF_IN$_M$ *and outgoing object flows* OF_OUT$_1$ *...* OF_OUT$_N$ *of types* OF_OUT$_1$_TYPE *...* OF_OUT$_N$_TYPE *is defined by a LOTOS process definition specified as follows:*

```
process ACTIONSTATE_AS [action, action_res] : exit (type_result) :=
    action; action_res? result: type_result;
    exit (result)
endproc
```

The ActionState *is invoked in its* Activity*'s behaviour expression (UCD 1) by the <CALL_SEND_ACTIONSTATE> non-terminal specified as follows.*

```
<CALL_SEND_ACTIONSTATE> ::=
              <object_flow_in > <synch>
              `` ACTIONSTATE_AS[action, action_res] accept ''
              <object_flow_out_list > `` in (''
              <activity_behaviour > ``)''
```

```
<object_flow_in> ::=
            <object_flow_in_name>
            ``['' < object_flow_gates > ``]''
<synch> ::= `` |[action, action_res]|''
<object_flow_in_name> ::= `` OF_IN_1'' - `` OF_IN_M''
<object_flow_out_list> ::= <> |
            <object_flow_out_list><object_flow_out>
<object_flow_out> ::=  <object_flow_out_name> ``:''
            <object_flow_out_type>
<object_flow_out_name> ::=
            `` OF_OUT_1'' - `` OF_OUT_N''
<object_flow_out_type> ::=
            `` OF_OUT_1_TYPE'' - `` OF_OUT_N_TYPE''
```

From the definitions of ClassifierInState, ObjectFlowState, CallAction and SendAction it is possible to specify the UML*i* stereotypes introduced in Section 4.2.3 that characterise the interaction object flows. The UML*i* stereotypes are specified in the following sections along with the three categories of SelectionState presented in Figure 4.16.

5.5.4 The ≪*presents*≫ Stereotype Specification

The ≪*presents*≫ stereotype identifies interaction object flows which create the context where InteractionClasses can be used in Activities, ActionStates and SelectionStates. This is the most complex category of interaction object flows and its UCD is based on the definition of the `INITIATE_UI_ACT` process.

The `INITIATE_UI_ACT` process is a sequence of CreateActions of InteractionClasses defined within a FreeContainer in a user interface diagram, including a CreateAction for the FreeContainer itself. The sequence starts with the CreateAction that instantiates the FreeContainer followed by the instantiation of the InteractionClasses immediately contained by the FreeContainer. CreateActions are also performed for each InteractionClass defined within Containers already instantiated until a CreateAction is performed for each InteractionClass defined within the FreeContainer. Figure 5.13 presents the `INITIATE_UI_ACT` for the `Connect` user interface in Figure 4.2 named `INITIATE_CONNECTUI_ACT`. There can be observed that the FreeContainer is instantiated by the `mk_ad` operation

```
process INITIATE_CONNECTUI_ACT[ abort ] :
      exit ( incClass , abs , disp , . . . , intClass , abs , disp ) :=
  CREATE_CONNECTUI_AS[ abort ] >>
  accept cui_ic:intClass , cui_abs:abs , cui_disp:disp in
  ( CREATE_OPTIONS_AS[ abort ]( cui_ic , cui_abs , cui_disp ) >>
    accept op_ic:intClass , op_abs:abs , op_disp:disp in
    ( CREATE_DETAILS_AS[ abort ]( cui_ic , cui_abs , cui_disp ) >>
      accept dt_ic:intClass , dt_abs:abs , dt_disp:disp in
        . . .
      ( CREATE_CANCEL_AS[ abort ]( op_ic , op_abs , op_disp ) >>
        accept cn_ic:intClass , cn_abs:abs , cn_disp:disp in
          . . .
          exit ( cui_ic , cui_abs , cui_disp ,
                 op_ic , op_abs , op_disp ,
                 dt_ic , dt_abs , dt_disp ,
                 . . .
                 cn_ic , cn_abs , cn_disp ,
                 . . . ) ) ) )
where
  process CREATE_CONNECTUI_AS[ abort ] : exit ( intClass , abs , disp ) :=
    ( i ; exit ( mk_ad ( ) ) )
    [> abort ; exit
  endproc
  process CREATE_OPTIONS_AS[ abort ]( cui_ic:intClass , cui_abs:abs , cui_disp:disp ) :
                                        exit ( intClass , abs , disp ) :=
    ( i ; exit ( mk_ad2( cui_ic , cui_abs , cui_disp ) ) )
    [> abort ; exit
  endproc
  process CREATE_DETAILS_AS[ abort ]( cui_ic:intClass , cui_abs:abs , cui_disp:disp ) :
                                        exit ( intClass , abs , disp ) :=
    ( i ; exit ( mk_ad2( cui_ic , cui_abs , cui_disp ) ) )
    [> abort ; exit
  endproc
  . . .
  process CREATE_CANCEL_AS[ abort ]( cn_ic:intClass , cn_abs:abs , cn_disp:disp ) :
                                        exit ( intClass , abs , disp ) :=
    ( i ; exit ( mk_ad2( cn_ic , cn_abs , cn_disp ) ) )
    [> abort ; exit
  endproc
  . . .
endproc
```

Figure 5.13: The `INITIATE_CONNECTUI_ACT` process.

in `CREATE_CONNECTUI_AS` as specified in UCD 22, and that the other InteractionClasses are instantiated by the `mk_ad2` operation in the other CreateAction ActionStates. For instance, it can be observed that the `options` Container is specified as contained by the FreeContainer and that the `cancel` ActionInvoker is specified as contained by the `options` Container.

From the definition of the `INITIATE_UI_ACT` process it is possible to define the ≪*presents*≫ stereotype as follows.

UCD 28 *The* ≪presents≫ *stereotype is the performing of an* `INITIATE_UI_ACT` *process for the associated* FreeContainer *in the definition of the associated* Activity *or* ActionState*,* `ASSOC_ACT`*, as follows.*

```
INITIATE_UI_ACT[abort] >>
  accept ic1:intClass, a1:abs, d1:disp,
         ic2:intClass, a2:abs, d2:disp,
         ...,
         icn:intClass, an:abs, dn:disp  in
  ( ASSOC_ACT[G_io^1 ∪ G_io^2 ∪ ··· ∪ G_io^n,
              start, sus1, sus2, ..., susn,
              restart, resu1, resu2, ..., resun, abortui]
      |[start, restart, abortui]|
    FREECONTAINER_CLS[G_io^1, start, sus1, restart, resu1, abortui]
                     (ic1, a1, d1)
      |[start, restart, abortui]|
    INTCLASS2_CLS[G_io^2, start, sus2, restart, resu2, abortui]
                 (ic2, a2, d2)
      |[start, restart, abortui]|
    ...
    INTCLASSn_CLS[G_io^n, start, susn, restart, resun, abortui)
 >> ...
```

where the behaviour expression of the `ASSOC_ACT` *process is defined as follows:*

```
start; sus1; sus2; ...; susn;
(* ASSOC_ACT specification *);
abortui
```

The `ASSOC_ACT` process in UCD 28 is an Activity as defined in UCD 1 with every gate of the InteractionClasses that compose the `Connect` user interface visible. Thus, the functionalities provided by the user interface are made available within the behavioural expression of the associated activity. The other categories of interaction object flows presented in this section are defined within the context of a FreeContainer associated to an Activity by an ≪*presents*≫ interaction object flow. In this case, the other categories of interaction object flows take advantage of this visibility of the gates of InteractionClasses.

Considering UCD 28, the `LIBRARY_ACT` process in Figure 5.3 can be refined as presented in Figure 5.14 to incorporate the ≪*presents*≫ interaction object flow of the `ConnectUI` FreeContainer associated to the `Connect` Activity. It can be observed there how user interface functionalities are made available to the `CONNECT_ACT`.

```
specification LIBRARY_ACT[abort] : exit
  behaviour
    INITIATE_CONNECTUI_ACT[abort] >>
    accept cui_ic:intClass, cui_abs:abs, cui_disp:disp,
           op_ic:intClass, op_abs:abs, op_disp:disp, ... in
    ( CONNECT_ACT[G_io^cui ∪ G_io^dt ∪ ···, start, sus_cui, sus_op, ...,
                restart, res_cui, res_op, ..., abortui]    |[start, restart, abortui]|
      CONNECTUI_CLS[G_io^cui, start, sus_cui, restart, res_cui, abortui]
                (cui_ic, cui_abs, cui_disp)               |[start, restart, abortui]|
      OPTIONS_CLS[G_io^op, start, sus_op, restart, res_op, abortui]
                (op_ic, op_abs, op_disp)                  |[start, restart, abortui]|
    ...) >> SELECTFUNCTION_ACT[abort]
    [> abort; exit
  where
    process CONNECT_ACT[G_io^cui ∪ G_io^dt ∪ ···, start, sus_cui, sus_op, ..., restart,
                        res_cui, res_op, ..., abortui, abort] : exit :=
      start; sus_cui; sus_op; ...;
      (* CONNECT_ACT specification *); abortui
      [> abort; exit
    endproc
    process SELECTFUNCTION_ACT[abort] := exit :=
      (* SELECTFUNCTION_ACT specification *)
    endproc
    process INITIATE_CONNECTUI_ACT[abort] :
      exit(incClass, abs, disp, ..., intClass, abs, disp) :=
      (* INITIATE_CONNECTUI_ACT[abort]
    endproc
    process CONNECTUI_CLS[G_io^cui, start, sus_cui, restart, res_cui, abortui] := exit :=
      (* FreeContainer specification *)
    endproc
    process OPTIONS_CLS[G_io^op, start, sus_op, restart, res_op, abortui] := exit :=
      (* Container specification *)
    endproc
    ...
endspec
```

Figure 5.14: The `INITIATE_CONNECT_ACT` in the `LIBRARY_ACT` process.

5.5.5 The ≪*cancels*≫ Stereotype Specification

The ≪*cancels*≫ stereotype is an interaction object flow stereotype included in many UI designs. In order to introduce ≪*cancels*≫, let `GENERIC_ACT` be a generic Activity, `ACTIVITY_ACT` be an Activity defined within `GENERIC_ACT`, and `cancel` be a ClassifierInStates of type ActionInvoker associated with `GENERIC_ACT` by a incoming ObjectFlowState. Thus, a ≪*cancels*≫ stereotype in the incoming ObjectFlowState is defined as follows.

UCD 29 *The* ≪cancels≫ *stereotype in an object flow associated with an* Activity *represented by the* `ACTIVITY_ACT` *process identifies the* `CANCELLABLE_ACT` *process presented below.*

```
process GENERIC_ACT[..., abort] : exit :=
 ... >> CANCELLABLE_ACT[..., resumecn, suspendcn, aoutcn] >>...
  process
```

```
  CANCELLABLE_ACT [ ..., resumecn, suspendcn, aoutcn ] : exit :=
    resumecn >>
    (( ACTIVITY_ACT [ ..., aoutcn ] >> exit ) |||
     aoutcn; suspendcn;
      (* suspend associated InteractionClasses *); exit )
  where
    process ACTIVITY_ACT [ aoutcn ]
        ...
    endproc
  endproc
endproc
```

The stereotype also identifies the assignment of the aout*,* resume *and* suspend *gates of the associated* ClassifierInState *of type* ActionInvoker *to the* aoutcn*,* resumecn *and* restartcn *gates in the* CANCELLABLE_ACT *process.*

UCD 29 shows that the aout gate of the associated ActionInvoker becomes the abort gate of ACTIVITY_ACT. This is the precise meaning of the ≪*cancels*≫ stereotype that enables the ActionInvoker while performing ACTIVITY_ACT. Thus, the CANCELLABLE_ACT can finish either by finishing the ACTIVITY_ACT or by an interaction on the aout gate of the associated ActionInvoker.

According to Figure 4.8, the Connect Activity can be cancelled due to the ≪*cancels*≫ interaction object flow of an instance of the Cancel ActionInvoker. Thus, the CONNECT_ACT process in Figure 5.14 can be refined as presented in Figure 5.15, assuming that $G_{connect}$ is the set of gates of the process as previously specified in Figure 5.14. Therefore, the CANCELLABLE_CONNECT_ACT replaces the original the CONNECT_ACT process, taking the original gates and compositions of CONNECT_ACT with other processes. Furthermore, the existing behaviour expression related to the ≪*presents*≫ interaction object flow is also preserved in CANCELLABLE_CONNECT_ACT. Finally, the original CONNECT_ACT process in Figure 5.15 becomes a subprocess of CANCELLABLE_CONNECT_ACT, corresponding to the ACTIVITY_ACT in UCD 29.

5.5.6 The OptionalState Specification

An OptionalState should have at least two selectable Activities. Thus, let a GENERIC_ACT be the process representing a generic Activity having a range of SUB1_ACT $\cdots$ SUBn_ACT representing Activities defined within the generic Activity. Particularly, let SUB2_AS represent an Activity that is an ActionState. In this

```
specification LIBRARY_ACT[abort] : exit
  behaviour
    ...
    ( CANCELLABLE_CONNECT_ACT[G_connect]
                                    |[start, restart, abortui]|
      CONNECTUI_CLS[G_io^cui, start, sus_cui, restart, res_cui, abortui]
                    (cui_ic, cui_abs, cui_disp)
                                                        |[start, restart, abortui]|
    ...)
    ...
  where
    process CANCELLABLE_CONNECT_ACT[G_connect] : exit :=
      start; sus_cui; sus_op; ...;
      res_cn >>
      ((CONNECT_ACT[G_connect] >> exit) ||| aout_cn; abortui; exit)
      [> abort; abortui; exit
    where
       process CONNECT_ACT[G_connect] : exit :=
          (* the regular behav. expression of CONNECT_ACT *)
       endproc
    endproc
    ...
endspec
```

Figure 5.15: Making the CONNECT_ACT cancellable.

context, an OptionalState having the range SUB1_ACT $\cdots$ SUBn_ACT of Activities as selectable Activities is defined as follows.

UCD 30 *An* OptionalState *is defined as the* OPTIONAL_ACT *process specified as follows.*

```
process GENERIC_ACT[..., abort] : exit :=
  ... >> OPTIONAL_ACT[..., aoutconfirm, abort_opt] >> ...
where
    process OPTIONAL_ACT[..., aoutconfirm, abort_opt] : exit :=
      resumesub1; ...; resumesubn; resumeconfirm >>
      (aoutsub1; suspendsub1; ...; suspendsubn; SUB1_ACT []
       (* SUB2_AS action state behaviour expression *)     []
       ...                                                 []
       aoutsubn; suspendsub1; ...; suspendsubn; SUBn_ACT []
       aoutconfirm; suspendconfirm;
          suspendsub1; ...; suspendsubn; exit)
    where
      process SUB1_ACT[..., abort_opt]
          (* SUB1_ACT specification *)
          >> OPTIONAL_ACT[]
      endproc
      process SUB3_ACT[..., abort_opt]
          (* SUB3_ACT specification *)
```

```
            >> OPTIONAL_ACT[]
        endproc
        ...
        process SUBn_ACT[..., abort_opt]
            (* SUBn_ACT specification *)
            >> OPTIONAL_ACT[]
        endproc
    endproc
endproc
```

The `resumesub1`*,* `suspendsub1`*,* `aoutsub1`*, ⋯,* `resumesubn`*,* `suspendsubn` *and* `aoutsubn` *are gates of either* ActionInvokers *associated to subactivities by* ≪activates≫ ObjectFlowStates *(UCD 45) or* PrimitiveInteractionClasses *associated to subactivities that are* ActionStates *by* ≪interacts≫ ObjectFlowStates *(UCD 33). The* `resumeconfirm`*,* `suspendconfirm` *and* `aoutconfirm` *are gates of an* ActionInvoker *associated to the* OptionalState *by a* ≪confirms≫ ObjectFlowState *(UCD 31).*

UCD 30 shows that the OptionalState is a special case of the Choice PseudoState (UCD 7). The behaviour expression of the OptionalState, in contrast with the Choice, depends on the existence of ActionInvokers and PrimitiveActionStates to provide the gates used in its composition. Further, UCD 30 shows that only one subactivity can be performed at a time, although every subactivity is available for selection when none of them is being performed. Thus, the associated objects of InteractionClass are activated in parallel, as presented in the UML version of the OptionalState in Figure 3.6(c). However, the subactivities cannot be performed in parallel as may be suggested in Figure 3.6(c).

Concerning the selection of subactivities in UCD 30, it can be verified that subactivities that are ActionStates are translated into LOTOS specifications in a different way from Activities. In fact, UCD 30 relies on the semantics of the ≪*interacts*≫ stereotype (UCD 33) to specify the behaviour expression for subactivities that are ActionStates.

5.5.7 The ≪*Confirms*≫ Stereotype Specification

The set of gates `resumeconfirm`, `suspendconfirm` and `aoutconfirm` are responsible for normal finishing of the OptionalState in UCD 30. These gates are provided by the mandatory ≪*confirms*≫ ObjectFlowState in the OptionalState specified as follows.

UCD 31 *The ≪confirms≫ stereotype in an* ObjectFlowState *specifies the assignment of the* `resume`*,* `suspend` *and* `aout` *gates of an associated* ClassifierInState *of type* ActionInvoker *to the* `resumeconfirm`*,* `suspendconfirm` *and* `aoutconfirm` *gates, respectively, in the* `OPTIONAL_ACT` *process of an* OptionalState *(UCD 30).*

5.5.8 The DestroyAction ActionState Specification

ActionStates where DestroyActions are performed could be specified in a similar way to SendAction ActionStates (UCD 27). However, DestroyActions are not specified in activity diagrams of UML. This means that no explicit notation exists in activity diagrams to specify the end of the existence of an Object.

We suggest the LOTOS semantics for UML*i* as a feasible formal framework for discussing such DestroyAction semantics. However, a minimal specification is required to avoid the creation of deadlocks when checking the generated LOTOS specification. Therefore, the problem here is the identification of a semantics for the DestroyAction that does not requires any special notation. The approach presented is to force a DestroyAction for any Object created in an Activity when leaving the activity. Thus, a DestroyAction can be specified as follows.

UCD 32 *A* DestroyAction *is the automatic performing of a* `destroy_class` *action of a* Class *when leaving the* Activity *where the* Object *was instantiated.*

This approach may present some initial difficulties specifying Objects that are either persistent or used in many different parts of a system. Concerning persistent Objects, it can be observed that locally, i.e., in an Activity, they can be instantiated, re-instantiated, updated and destroyed as a result of database queries specified whitin Class's Methods [23]. Moreover, this is the same kind of limitation as database transactions impose on the use of persistent Objects that is effective for implementing database systems [23]. Concerning Objects used in many different parts of a system, they can be defined in the lowest common Activity containing the parts of the system where they are required.

Therefore, UCD 32 is a simple and apparently feasible definition for DestroyAction ActionStates since other UML*i* constructs such as Class and Activity provide the framework required for its specification.

5.5.9 The ≪*interacts*≫ Stereotype Specification

The ≪*interacts*≫ stereotype can be used for ClassifierInStates that are PrimitiveInteractionClasses or Containers.

If the interaction Object is an instance of PrimitiveInteractionClass then the ≪*interacts*≫ stereotype identifies an ActionState where the Object can exchange a data item with another Object associated with the associated ActionState. In this case, the ≪*interacts*≫ stereotype also enables the interaction Object when the associated ActionState is performed and it disables the interaction Object when the ActionState finishes. The `getValue()` and `setValue()` standard operation names in UML*i* diagrams are used to indicate the assignment of the `aout` and `ainp` gates to gates of the collaborating object, respectively.

UCD 33 *The* ≪interacts≫ *stereotype in an* ObjectFlowState *associated with a* ClassifierInState *of type* PrimitiveInteractionClass *specifying that the* ClassifierInState *shares its* `ainp` *and* `aout` *gates with the gates of the process representing the* Object *using the* `getValue()` *and* `setValue()` *operations respectively. Further, for the* ClassifierInState `i` *sending/receiving a data item* `x`*, the behaviour expression of the associated* ActionState *is defined as follows.*

$\underline{\mathrm{resumei}}\,; < INTERACTION >; \underline{\mathrm{suspendi}}$

where $< INTERACTION >$ *is* $\underline{ainpi}?\underline{x}{:}\underline{aInpData}$ *and/or* $\underline{aouti}!\underline{x}$

If the interaction Object is an instance of Container then the ≪*presents*≫ stereotypes enables the Object of Container and its contained interaction Objects before performing any other action within the associated Activity, and disables them when the Activity finishes.

UCD 34 *The* ≪interacts≫ *stereotype in an* ObjectFlowState *associated with a* ClassifierInState *of type* Container *specifying that the behaviour expression of the associated* Activity *is defined as follows.*

$\underline{\mathrm{resumeCont}}; \underline{\mathrm{resumeObj1}}; \underline{\mathrm{resumeObj2}}\,; \ldots;\ \underline{\mathrm{resumeObjN}}$
$(*\ regular\ translation\ of\ activity\ 's\ constructs\ *)\,;$
$\underline{\mathrm{suspendCont}}; \underline{\mathrm{suspendObj1}}; \underline{\mathrm{suspendObj2}}\,; \ldots;\ \underline{\mathrm{suspendObjN}}$

where `Cont` *is an instance of* Container *which contains the* Objects `Obj1`*,* `Obj2`*,* $\cdots$*,* `ObjN`*.*

Returning to the running translation of the Library System, the OrderIndependentState constituted the behaviour of the `Connect` Activity. Moreover, the selectable activities of the OrderIndependentState are ActionStates where domain

Objects collaborate with interaction Objects that are associated with the selectable states through the use of ≪*interacts*≫ ObjectFlowStates. Thus, the current translation of the `Connect` Activity in Figure 5.15 can finally be refined as presented in Figure 5.16. There, the `aout_lg` and `aout_pw` are gates shared with the mapping of the `getLogin()` and `getPassword()` Operations of the `uq` Object.

```
process CONNECT_ACT[G_connect, abort_oi] : exit :=
  ( resumesub1; aout_lg ! login; suspendsub1 |||
    resumesubn; aout_pw ! password; suspendsubn )
  [> abort; exit
endproc
```

Figure 5.16: A refinement of the `CONNECT_ACT` process introduced in Figure 5.15.

5.6 Verification of the Library System Specification

There are many tools that can perform verification of LOTOS specifications, making use of the formal properties of LOTOS. Therefore, suppose that the Φ function introduced in this chapter can be defined for the other UMLi constructs not considered in this paper. This means that using Φ it may be possible to verify any UMLi model. Furthermore, problems that may be identified in LOTOS specifications generated by the Φ function may be interpreted as a *semantic problems* in UMLi.

The LOTOS specifications of the UMLi models of the Library System were checked using a LOTOS verification tool. Thus, a LOTOS specification for the class diagram in Figure 3.2, the user interface diagram in Figure 4.2 and the activity diagrams in Figure 4.6 and 4.8 were implemented by applying the UCDs presented in this paper. Indeed, most of the LOTOS examples in this chapter are fragments of this LOTOS specification, which has 1632 lines of code.

CADP [39] was selected to verify the LOTOS specification. CADP is a set of integrated tools with a wide range of functionalities concerning the verification of LOTOS specifications such as simulation and model checking identifying, for instance, deadlock, livelock and unreached states. EUCALYPTUS is the graphical environment of CADP responsible for invoking the CADP tools that analyse the LOTOS specification. Figure 5.17 presents a snapshot of EUCALYPTUS when verifying the LOTOS specifications of the Library System. The background frame in Figure 5.17 is the main user interface of EUCALYPTUS.

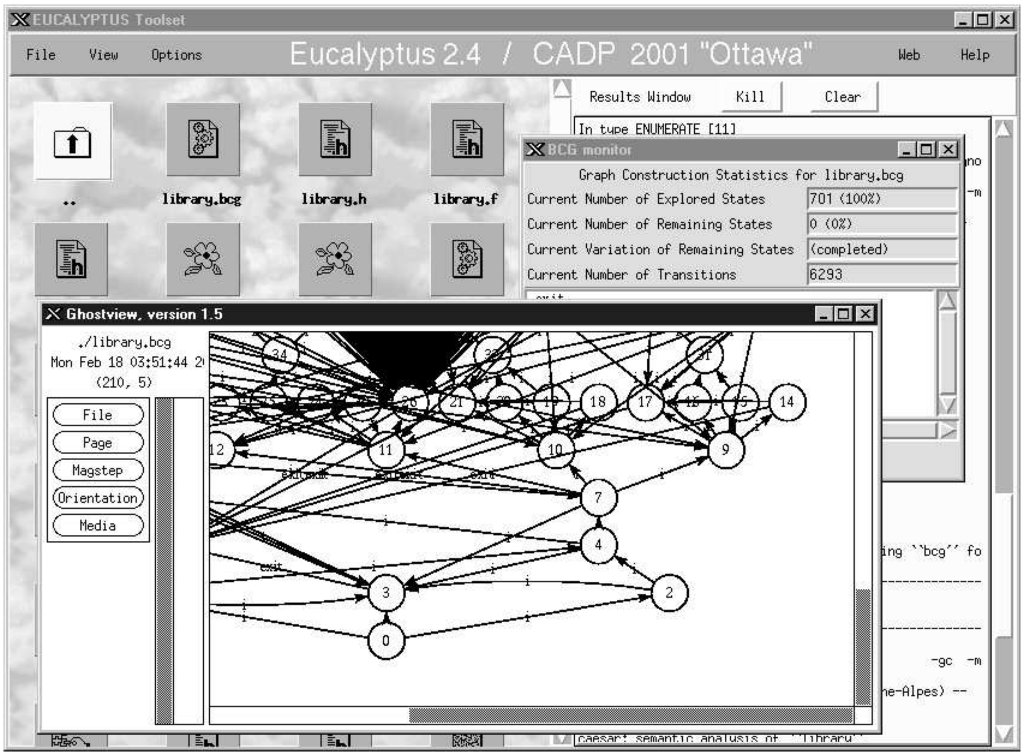


Figure 5.17: Snapshot of CADP when analysing the Library System specification.

The CAESAR and CAESAR.ADT tools in CADP are responsible for the compilation and verification of the LOTOS specification. For instance, using CAESAR it was possible to verify if the LOTOS specifications were syntactic correct. Further, it was possible to generate a *Binary Coded Graph* (BCG), which is a computer representation for a *Labelled Transition System* [92] from the LOTOS specifications. In a BCG, a state is of the entire application rather than of part of the application such as an object. In the case of the Library System specifications, we can see in the `BCG monitor` frame in Figure 5.17. Moreover, from the BCG generated, it was possible to verify if the LOTOS specifications did not have deadlocks, livelocks and unreachable states. Still in Figure 5.17, we can have a partial view of the generated BCG. The arcs in this partial view are the transitions and they are labelled with their triggering actions. As expected, most actions in the BCG are internal actions, e.g., *i*, since Operations are under-specified in the LOTOS specifications generated from UML models.

5.7 Summary

The UMLi specification initiated with the introduction of its notation and meta-model in Chapter 4 has been concluded in this chapter with the introduction of the UMLi semantics given in terms of the mapping of its constructs into LOTOS. In fact, the translation of a UMLi model into LOTOS specifications using the Φ function provided throughout this chapter means that the UMLi model can have exactly one interpretation, provided by the formal semantics of the generated LOTOS specifications. Moreover, the UMLi specification provided in this thesis is more precise than the UML specification in [99] that relies on the use of the English language to describe the meaning of the UML constructs. The Φ function introduced in this chapter improves in the following ways on mappings of object concepts into LOTOS specifications given in the literature:

- The Φ mappings of Actions and Attributes are partially based on [142]. The mappings of Activities and Transitions in [142] which are related to the mappings of Actions are not incorporated into Φ since they are restricted to state diagrams of a single object. Thus, the mapping of Actions in [142] is improved by the Φ function to fit with more generic mappings of Activities and Transitions.

- The Φ mapping of Class is based on the ADC interactor [82]. The Φ mapping of Class, however, is richer than an ADC interactor. For instance, it provides mappings for Operations. Further, the definition of Attributes in [142] is improved in the Φ function to be a pair of LOTOS gates, as specified in the ADC interactor.

- The Φ mapping of InteractionClasses based on the ADC interactor [82] is improved by the incorporation of the notion of containment between InteractionClasses, as proposed in [104].

Other benefits of having a formal semantics as presented in this chapter are presented as follows.

- ConcurTaskTree [101] has a LOTOS-based semantics [102]. ConcurTaskTree, however, does not provide an integrated specification of the domain part and the user interface part of an interactive system. In fact, ConcurTaskTree does support the specification of some domain objects, however it does not support the specification of the entire domain including, for instance, the specification of the associations, generalisations and specialisations of domain objects. This means that corrections of problems identified during the verification of the user interface part of an interactive system using ConcurTaskTree [102] may affect its integration with the domain part of the system, in the same way that modifications resulting from the verification of the domain part may affect the user interface part. This kind of problem does not happen in UML*i* since both the user interface part and the domain part of an interactive application can be specified at once using a single notation.

- Formal languages have been used for a long time to specify user interfaces, e.g., [10]. Formal specifications in general, however, may be difficult to explain to everyone participating in the design of interactive systems, e.g., system users and managers. Moreover, formal specifications may become so detailed that they can become difficult to understand even by people skilled in the use of formal languages. For instance, the understanding of the formal specification of the Library System presented in this chapter can be challenging for people skilled in LOTOS. Thus, the UML*i* semantics presented here provides an approach to formally specify interactive systems using a diagrammatic notation such as UML that embeds the formal specification in its semantics.

- The semantics of the InteractionClass construct of UML*i* is an ADC interaction, which is one of the current approaches to formalising interactive systems [82]. This demonstrates that the proposal for a UML*i* semantics presented in this chapter is more in line with other approaches to formalisation of interactive systems.

- The InitialInteractor (UCD 2) identifies top-level activities that can be used as the starting point to traverse UML*i* model, for example, to perform a model checking as in Section 5.6, or to generate user interface code [107]. Without the InitialInteractor is unclear how UML*i* and UML models can be traversed.

- The DestroyAction (UCD 32) provides an approach to addressing the destruction of Objects that is omitted in the informal description of the UML semantics in [99].

- Many potential problems resulting from the design of UMLi models such as deadlocks, livelocks and unreached states can be identified using LOTOS tools as discussed in Section 5.6.

From the UMLi specification describing its notation, syntax and semantics, it is possible to investigate the practical use of the language for developing UIs. Thus, tool support for UMLi is presented in the next chapter.

Chapter 6

UML*i* Tool Support

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., Rational Rose [116], ArgoUML [119], Together [140]), and MB-UIDEs are themselves often associated with interactive model development tools (e.g., [6, 7, 86, 112, 135]). The focus of this chapter 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, ARGO*i*, the UML*i* environment introduced in this chapter, is an extension of ArgoUML [119].

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

- where user interfaces and their associated applications can be modelled in an integrated way;
- that facilitates the design of activity diagrams that simultaneously depend on 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 chapter has the following structure. ArgoUML [119] is introduced in Section 6.1. Generic aspects of ARGO*i* are presented in Section 6.2. Tool support for modelling UI presentations is described in Section 6.3. Tool support for modelling interactive application control-flow and data-flow using activity diagrams is presented in Sections 6.4 and 6.5. Tool support for modelling the collaboration between interaction and domain objects is presented in Section 6.6. Conclusions are presented in Section 6.7.

6.1 ArgoUML: A UML-Based Development Environment

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

- ArgoUML 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 ArgoUML object model used for handling UML models at runtime conforms with the OMG UML 1.3 specification [99].

The ArgoUML features required to introduce ARGO*i* are presented in this section.

6.1.1 ArgoUML User Interface

ArgoUML is a graphical environment where designers can build UML models by manipulating graphical elements representing UML constructs. These graphical elements are manipulated by actions performed using pointing-devices on the ArgoUML user interface that implements a number of direct-manipulation techniques.

UML tools usually provide additional designer support for handling the inherent complexity of building UML models. For instance, forms are used to refine the specification of model elements, and trees with collapsible/expandable leaves representing the UML elements are used to navigate through UML models.

A snapshot of the ArgoUML user interface in Figure 6.1 provides an insight into the facilities of ArgoUML. There four distinct panels can be observed, each providing a functionality described as follows.

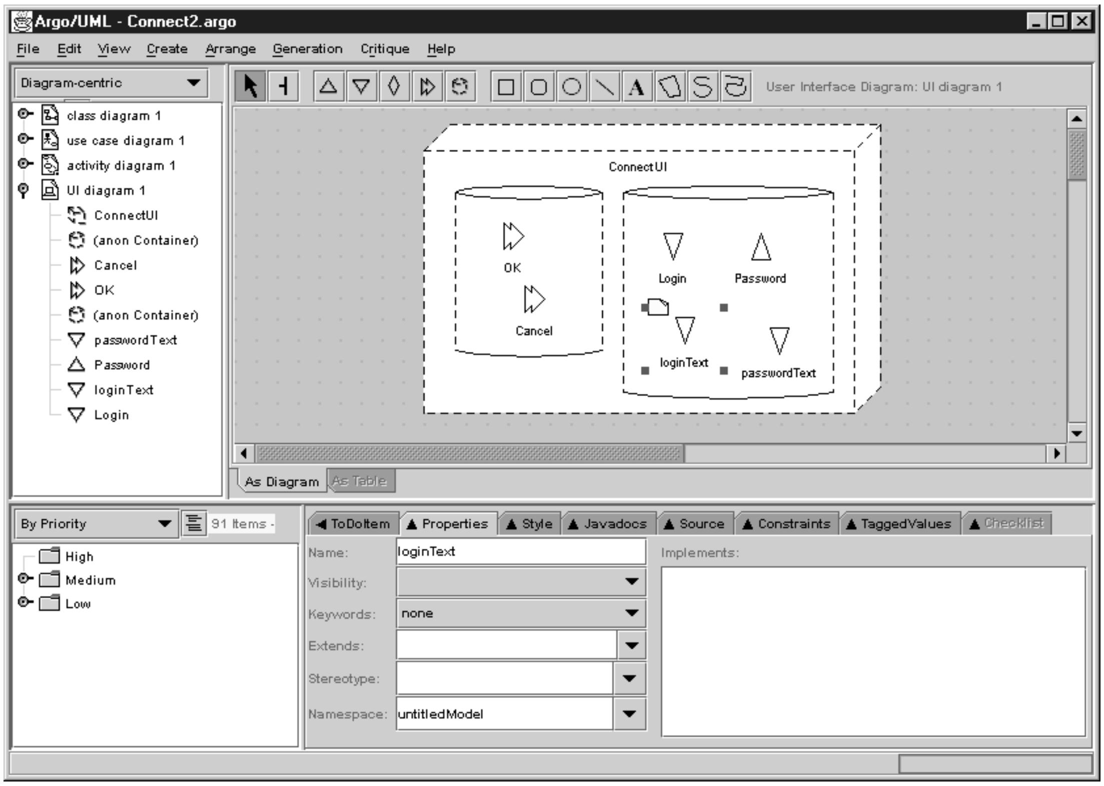


Figure 6.1: A snapshot of the ArgoUML user interface.

- *Editing panel.* This panel is located at the top right of Figure 6.1, 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 *construct creator* or an *operator.* Construct creators are used for adding new components to the working area. Operators are used for modifying constructs already created in the working area. The contents of the selection box, in terms of construct creators and operators, depends on the kind of diagram that is being edited. For instance, the selection box in Figure 6.1 contains the construct 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 6.1. From the collapsible/expandable tree representing the UML elements, designers can navigate through the entire UML model, switching from one diagram or diagram element to another by selecting tree elements using a pointer-device. 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 6.1. In this panel designers can interact with elements of the UML model and ArgoUML 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 6.1. In the case of UML*i*, we are particularly interested in the *properties form*, where designers can provide additional information for UML*i* constructs. 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 6.1 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 6.1. Design critics provided by ArgoUML 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, ArgoUML provides non-compulsory criticism facilities that may provide guidance for building consistent models in an incremental way. The version of ArgoUML that implements the UML*i* extensions, v.0.8.1, does not make use of the ArgoUML criticism facilities.

Still in Figure 6.1, there is a menu on top of the panels with the following options:

- *File.* Handles Argo/UML projects that are sets of UML diagrams stored in XMI format [99] along with complementary information stored in PGML format [2].

- *Edit.* Cutting, copying and pasting facilities for constructs. Further, this option allows the removal of objects from UML models in addition to their removal from diagrams.

- *View.* Provides options for finding and selecting models and constructs within models.

- *Create.* Provides options for including new instances of UML diagrams in the current model.
- *Arrange.* Provides advanced facilities for organising nodes and edges in diagrams. For instance, it provides the *Broom* tool which helps the vertical and horizontal alignment of edges in UML diagrams.
- *Generation.* Provides facilities to generate Java classes from UML classes. The generation of code, however, is restricted to Classes. It does not produce code for Operations, for example.
- *Critique.* Provides the critiquing facilities that distinguish Argo/UML from other UML tools. The critiques are guidelines to good practices in UML modelling and can be verified against the current model. Thus, Argo/UML can identify and present in the *To Do* Panel the violation of these guidelines.
- *Help.* Presents credits for the developers of Argo/UML.

6.1.2 ArgoUML Architecture

A high-level description of the ArgoUML architecture, in addition to the description of the ArgoUML user interface above, is required to explain the implementation of ARGO*i*. The components of ArgoUML are presented in Figure 6.2. There, the packages are composed of Java/Swing classes [45] that may have their own sub-packages. These packages have the following roles in ArgoUML.

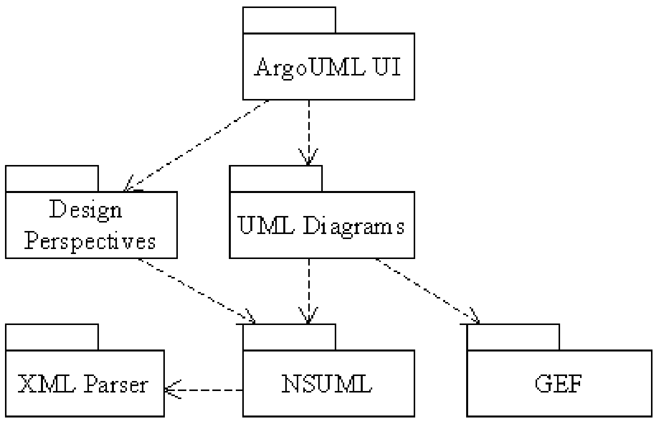


Figure 6.2: A top-level package view of the ArgoUML architecture.

- The `Graph Editing Framework (GEF)` package [139] provides a generic set of graphical constructs for implementing diagrams, nodes and edges.
- The `Novosoft UML (NSUML)` package [96] implements the UML object model in Java.
- The `XML Parser` package is used by classes of the `NSUML` package for loading object models from XMI files.
- The `Design Perspectives` package provides the functionalities required for supporting the collapsible/expandable tree in the Navigation Panel.
- The *UML Diagrams* package is responsible for the integration of the UML object model implemented by the `NSUML` package to the graphical representation of the UML model implemented by extended `GEF` classes.
- The *ArgoUML UI* package is responsible for the integration of the the panels supported by the `Design Perspectives` and `UML Diagram` packages in a single user interface, as presented in Figure 6.1.

The snapshot in Figure 6.1 is actually a version of ArgoUML extended to support UML*i*. Indeed, it presents a UI diagram not specified in UML. The following sections in this chapter discuss the implementation of ARGO*i*.

6.2 ARGO*i*: A UML*i*-Based Development Environment

ARGO*i* has been implemented from ArgoUML version 0.8.1 [119]. Thus, some ARGO*i* classes have been created from scratch while others are modified classes of ArgoUML. A discussion about the technical aspects of the implementation is minimised in the chapter. Nevertheless, there are two reports available in the ARGO*i* web-page (http://img.cs.man.ac.uk/umli/software.html) indicating the differences between ARGO*i* 0.1 and ArgoUML 0.8.1 and between the UML XMI DTD and UML*i* XMI DTD.

Most of the modifications have been made in the `NSUML` and `UML Diagrams` packages. The `UML Diagrams` package has been extended to provide the following facilities for modelling UML*i* diagrams:

- Editing facilities for user interface diagrams, as discussed in Section 6.3;
- Editing facilities in activity diagrams for modelling SelectionStates, InitialInteractions and interaction object flows, as described in Sections 6.4 and 6.5;
- Wizards in activity diagrams for designing control-flow, as discussed in Section 6.4;
- Wizards in activity diagrams for modelling interaction object flows, as discussed in Section 6.5.

The NSUML package has been extended to provide the following facilities for modelling UMLi diagrams:

- Support the UMLi constructs in terms of the UMLi metamodel specified in Section 4.4.
- Support the generation and reading of UMLi XMI DTD conformant files.

6.3 User Interface Diagram Modelling Support

A snapshot of the ARGOi user interface when modelling a UI diagram is presented in Figure 6.1, which shows the modelling of the ConnectUI diagram from Figure 4.2. 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 InteractionClass 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 ArgoUML.

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 6.1.

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.

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

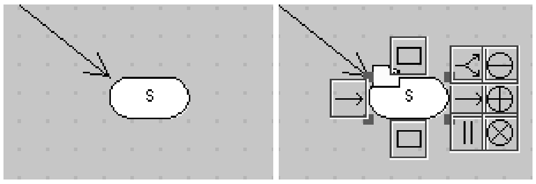
InteractionClass placement (in contrast with containment) is not relevant in UI diagrams, since layout is normally more a concrete presentation concern than an abstract presentation one, as discussed in Chapter 2. Therefore, the UI diagrams need to be refined into concrete presentations, as described in Section 3.5.

6.4 SelectionState 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 [69, 101]. 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 UML*i*, 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 constructs was anticipated by Harel and Gery [51] when statecharts were adopted by UML. The temporal-relation wizard in ARGO*i* 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 constructs required



(a) (b)

Figure 6.3: (a) The unselected S State. (b) The selected S State along with the ARGO*i* temporal-relation wizard on its right.

for modelling the behaviour of temporal relations in a task model.

2. It exploits the potential of SelectionStates and ReturnTransitions 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 [101], MOBI-D [112] and Teallach [7].

The temporal-relation wizard appears every time a node in an activity diagram is selected, such as for the State S in Figure 6.3(b). The wizard is the iconographic menu to the right of S. We can see the same State S in Figure 6.3(a) before it was selected. The other wizards in Figure 6.3(b) are standard activity diagram wizards of ArgoUML. The temporal-relation wizard has six options that perform the following actions:

- *Sequential option* (→): This option builds an Activity connected to the current node by a Transition. This action is represented graphically in Figure 6.4(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 6.4(b).

- *Choice option* (−<): 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

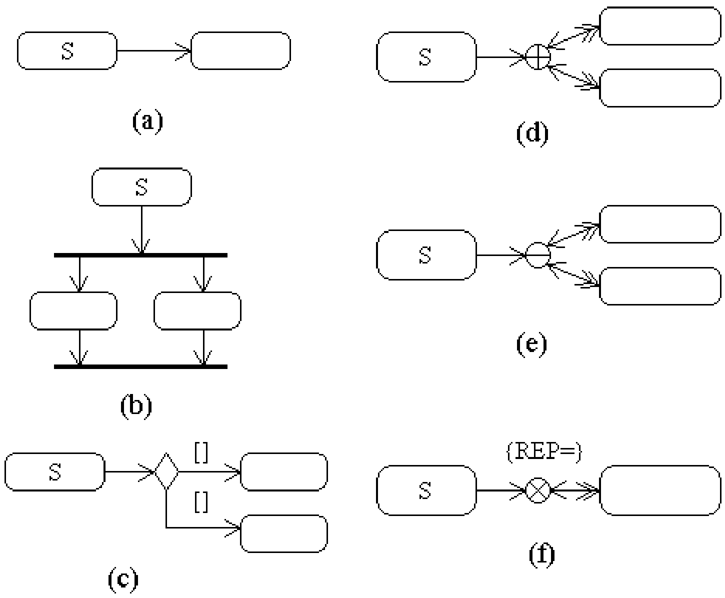


Figure 6.4: The result of each of the six actions that can be performed by the temporal relation wizard. The S State is that from which the wizard is invoked.

guarded to remind designers that these guards may be required. This action is represented graphically in Figure 6.4(c).

- *OrderIndependent option* ($\oplus$): 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 6.4(d).

- *Optional option* ($\ominus$): 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 6.4(e).

- *Repeatable option* ($\otimes$): 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 6.4(f).

Figure 6.5 shows an snapshot of ARGOi during the modelling of the activity

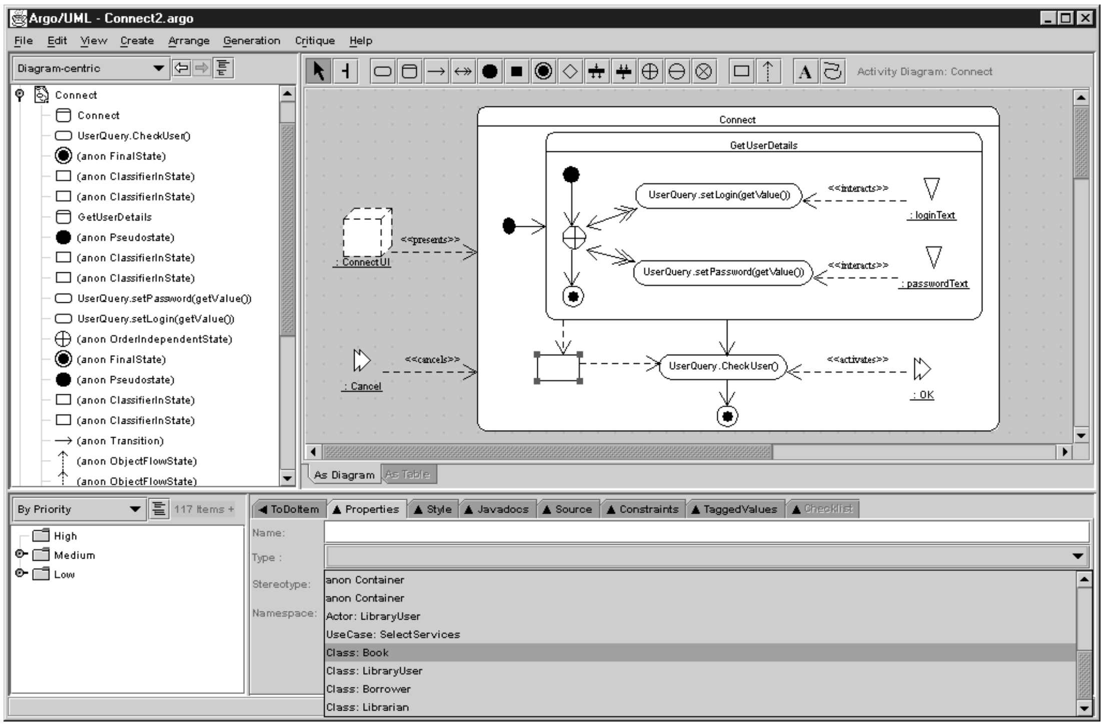


Figure 6.5: The modelling of the activity diagram in Figure 4.8.

diagram in Figure 4.8. The OrderIndependentState in Figure 6.5 was built using the temporal-relation wizard. In fact, the InitialState of `GetUserDetails` corresponds to the `S` State in Figure 6.4(d), and the `UserQuery.setLogin(getValue())` and `UserQuery.setPassword(getValue())` ActionStates are refinements of the two SelectableStates created along with the *OrderIndependentState*.

In terms of control-flow modelling, UMLi still specifies the InitialInteraction. Thus, ARGOi provides an InitialInteraction creator in the selection box when editing an activity diagram.

6.5 Interaction Object Flow Modelling Support

Object flows provide the ability to specify data-flows in activity diagrams. For non-interaction Classes, UMLi specifies that object flows can be connected to CompositeStates in addition to ActionStates. For InteractionClasses, UMLi specifies interaction object flows 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 InteractionClass, 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 an appropriate 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 as discussed in Section 5.5.4.

ARGO*i* provides facilities to cope with the complexity of selecting types and stereotypes for interaction object flows. Examples of these facilities can be provided using the activity diagram in Figure 4.8.

6.5.1 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 6.5 shows how the combo box in the properties form can be used to specify the `type` of the ClassifierInState 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 ClassifierInState. Two aspects of a UML*i* model analysed by ARGO*i* for filtering Classifiers are the following:

- *FreeContainer context.* Figure 4.8 shows the use of the `ConnectUI` FreeContainer as a ≪*presents*≫ interaction object flow of the `Connect` activity. This means that only the following InteractionClasses 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 *PrimitiveInteractionClasses* contained by `ConnectUI` (see Figure 4.2);
  2. The specified FreeContainers. This provides the ability to create new FreeContainer contexts.

- *ActionInvoker roles.* An instance of an ActionInvoker should not play more than one role in a FreeContainer, e.g., ▷ `Cancel` in Figure 4.2 should not be associated with any other sub-activity of the `Connect` activity in Figure 4.8 since it is already responsible for the cancelling behaviour within the `Connect` activity. Thus, ARGOi can notify designers when it identifies an ActionInvoker playing more than one role.

6.5.2 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 ClassifierInState. An aspect of the UMLi model 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 InteractionClass, or ≪*activates*≫, which makes the associated InteractionClass a trigger of the ActionState.

6.6 Domain Modelling Support

Interaction and domain Classes 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 `uq.setLogin(getValue())` ActionState in Figure 4.8. UMLi makes explicit such collaborations since it provides a clear distinction between interaction and non-interaction Classes. Moreover, UMLi makes explicit the problem of creating and preserving the integration between interaction and domain Classes in the designs of interactive applications.

The difficulty of creating this integration can be partially addressed through the checking of UML*i* models. In fact, ActionStates, as defined throughout Section 5.5, are not allowed to be performed over methods of non-instantiated Objects, as can be verified using the LOTOS semantics and model checking techniques discussed in Section 5.6. 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. Thus, the wizard can check and notify the designer about ActionStates affected by such Class removal. Indeed, designers may 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.

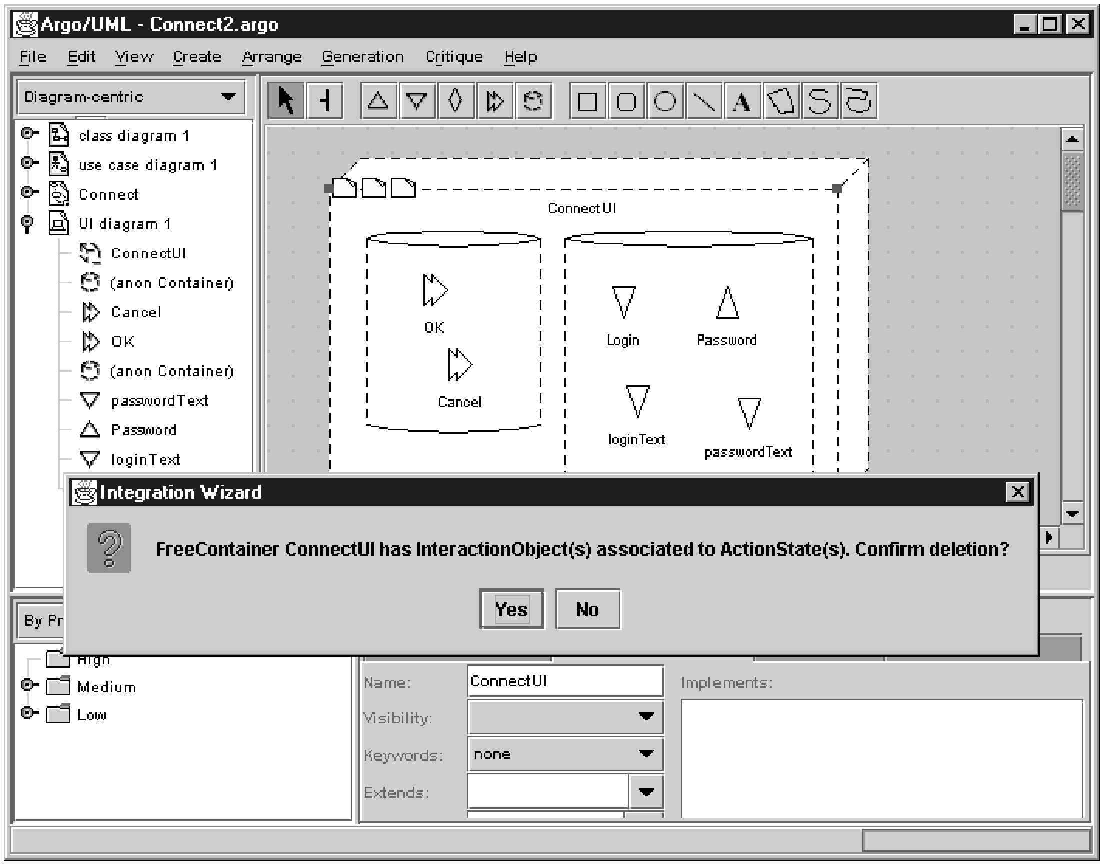


Figure 6.6: An attempt to delete the `ConnectUI` FreeContainer can trigger the integration wizard.

Considering this difficulty of preserving the collaboration of the interaction and domain Classes, the *integration wizard* notifies designers about the effects of Class 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 a ClassifierInState that shares at least one common State with an InteractionClass. For example, the interaction wizard will notify a user deleting the `UserQuery` Class in a class diagram about the risk of losing the relationship that this Class has with the `loginText` Inputer, as specified in Figure 4.8.

- *Deleting* InteractionClasses. The *integration wizard* checks in the UMLi models to see if a PrimitiveInteractionClass to be deleted is a type of a ClassifierInState that shares at least one common State with a Class. For example, the interaction wizard will notify a user deleting the `loginText` Inputer in a user interface diagram about the risk of losing the relationship that this Inputer has with the `UserQuery` Class, as specified in Figure 4.8.

- *Deleting* FreeContainers *and* Containers. The *integration wizard* checks in the UMLi models to see if a Container or a FreeContainer to be deleted contains PrimitiveInteractionClasses or ActionInvokers that share ActionStates with domain Classes, as described in *deleting* InteractionClasses above. The deletion of a Container or FreeContainer in ARGOi implies the recursive deletion of its contained InteractionClasses. Thus, the *integration wizard* also verifies recursively the side-effects of such a deletion in terms of interaction and domain Class integration. For example, Figure 6.6 shows the *integration wizard* when deleting the `ConnectUI` FreeContainer. In this case, the *integration wizard* is activated since the `loginText`, `passwordText` and `OK` InteractionClasses of the `ConnectUI` FreeContainer are associated to ActionStates, as modelled in the activity diagram in Figure 6.5.

6.7 Summary

This chapter shows that the UMLi specification can be effectively implemented in a UML-based design environment. Moreover, this chapter shows that ARGOi, a UMLi-based tool, can provide support for modelling the aspects of a interactive application usually modelled in MB-UIDEs, e.g., the modelling of abstract presentation models using the UI diagram and the modelling of the collaboration between domain and interaction objects using object flows and interaction object flows, by exploiting the UML*i* specification. Moreover, this chapter demonstrates practically that UML*i* is a conservative extension of UML able to provide the UML*i*-specific support for user interface design in models initially built in ArgoUML, a standard UML-based tool.

Compared with other UML tools (e.g., Rational Rose [116], Together [140], ArgoUML [119]), ARGO*i* provides additional tool support for:

- modelling abstract user interface presentations using the UML*i* user interface diagram (Section 6.3);
- modelling common UI behaviours using the *temporal-relation wizard* that exploits the use of the UML*i* SelectionStates (Section 6.4);
- modelling in an explicit and effective way the collaboration between interaction and domain Classes (Section 6.5);
- preserving, when modelled, the integration between interaction and domain Classes using the *integration wizard* (Section 6.6).

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 6.5, 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 [7] and MOBI-D [112].
- 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 [6] uses Together [140] for building its models, and AME [86] 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 support for modelling structural and dynamic aspects of interactive applications (Sections 6.3 and 6.4). Moreover, ARGO*i* provides support for relating structural and dynamic aspects (Sections 6.5 and 6.6). 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.

ARGO*i* has been used to model three comprehensive versions of the Library System. The first version does not specify any aspect of the system's user interfaces. The second version specifies the system's user interfaces using standard UML. The third version specifies the system's user interface using UML*i*. These models are used to evaluate the benefits of UML*i* in terms of design metrics, as described in the following chapter.

Chapter 7

Metric Assessment of Resulting Models

This chapter describes a study that analyses the effects of modelling UIs in UML, and the effects of using UML*i* as an alternative to UML when modelling interactive systems. The study analyses these effects via metrics that quantify dimensions of the complexity of models of the Library System. Two hypotheses are tested through the analysis of this metric study.

Hypothesis 1 *Standard UML models that include UI properties are structurally, behaviourally and visually more complex than standard UML models that do not include UI properties when describing properties of an interactive system.*

Hypothesis 2 *Standard UML models are structurally, behaviourally and visually more complex than UMLi models when describing the same set of properties of an interactive system.*

Hypothesis 1 emphasises the concern about the lack of UI support for UI modelling in UML. For instance, there is no indication of how much relevant specification in terms of complexity may be added by modelling UIs along with the core application. Moreover, considering the current tendency of system designers using UML to address user interface issues somewhat superficially, there is no indication of how much relevant specification in terms of complexity may be missed by not modelling UIs along with the core application functionality.

Relying on Hypothesis 1, Hypothesis 2 claims that the effects of UIs on the overall complexity of a model of an application can be reduced by better UML support for UI modelling.

The metric study is done by comparing design metrics measured from internal attributes of UML and UML*i* models. Initially, three scenarios are considered in the study, two described by sets of UML models and one described by a set of UML*i* models. In the first set of UML models, UI properties of an interactive application are not described. In the second set of UML models, the UI properties are described. In the set of UML*i* models, UI properties are described. To simplify the description of the metric study, these three modelling scenarios are hereafter called UML_noUI, UML_UI and UML*i*_UI models, respectively. Thus, a comparative analysis of metrics measured in the UML_noUI and UML_UI models is used to test Hypothesis 1. A comparative analysis of the metrics measured in the UML_UI and UML*i*_UI models is used to test Hypothesis 2.

The metric study covers the UML_noUI, UML_UI and UML*i*_UI models for two scenarios: when modelling the `ConnectToSystem` service, representing a single interactive service of the Library System; and when modelling the complete Library System. The results for the `ConnectToSystem` service are provided separately as they (i) allow a focused description of the metric study; (ii) ease the explanation of certain details, as they are obtained from a specific context. More generic results can be achieved analysing the nine interactive services that compose the Library System than is the case for the `ConnectToSystem` service on its own. The use of typical modelling techniques and similar reuse strategies for common elements in all models are two aspects of the study where special attention has been paid in order to produce valid results.

This chapter is organised as follows. Section 7.1 presents related work on design metrics. Section 7.2 defines the metrics used in this thesis to quantify the complexity associated with the design of UIs in UML*i* and UML. Section 7.3 describes how the models used in the metric study have been produced. Section 7.4 describes a metric study of the `ConnectToSystem` service of the Library System. Section 7.5 shows the accumulative effects in terms of metrics of the models of the complete Library System. Section 7.6 presents conclusions about the the effects of UI functionalities in UML*i* and UML models.

7.1 On the Assessment of Models using Metrics

Metrics are used in the study in this chapter since they can quantify many dimensions of complexity from internal attributes of designs. There are many proposals for metrics that measure complexity, e.g., [8, 11, 20, 21, 24, 87, 127, 148] in object-oriented designs. The large number of proposals indicates that design metrics is an evolving area in computer science with many open questions. For instance, for many metrics, e.g., coupling and cohesion, there is no objective criterion for choosing which among the alternatives are to be used in any particular circumstance. Also, many proposed metrics are not validated as indicators of desirable characteristics of designs. Moreover, it is unknown exactly how any desirable characteristic to which metrics have been associated correlate or not with any observable quality of designs. Thus, it is unknown which combination of metrics best correlates with quality of designs. Considering these uncertainties, the approach in this chapter is to use metrics that, if possible, have been systematically validated for properties that are well-accepted indicators of quality, e.g., low fault-proneness and low demand for maintenance.

In terms of structural complexity, this study uses the suite of metrics proposed by Chidamber and Kemere (CK metrics) [24] because:

- Basili et al. [8] have validated the CK metrics for class fault-proneness (except for *lack of cohesion on methods* LCOM), and Li and Henry [78] have validated the CK metrics for the number of maintenance modifications.

- Basili et al. [8] have provided a valuable indication of the impact of UI classes on the validation of each CK metric. Indeed, the experimentation described there was composed of eight medium-sized interactive systems built in C++. The user interfaces of those systems were built using the OSF/MOTIF [149] toolkit. Therefore, due to the controlled nature of the experiment, it was possible to describe the impact of CK metrics on the database and user interface classes.

- It is based on Booch's method and notation [17], a predecessor of UML [99].

- It has influenced many other proposals [11, 148].

Briand et al. [21] is another metric study related to structural complexity which raises the question about the participation of class libraries on metrics. The paper concludes that classes that belong to class libraries are important to making coupling-based metrics relevant. Therefore, in terms of UI design, toolkits, which are class libraries of interaction classes (or *widgets*), should be considered when measuring coupling. For instance, the MotifApp implementation of OSF/MOTIF [149] considered in Basili et al. [8] has affected the resulting metrics. However, the results in Briand et al. [21] raise a question about the validity of the CK metrics for fault-proneness if coupling involving toolkits is not counted.

In terms of behavioural complexity, this study uses McCabe's cyclomatic complexity [87] because it has long been a well-established metric whose original definition at code-level has been translated to the design level. It has also influenced other metric studies related to UML [127].

Visual complexity is a dimension of complexity of models of interactive systems apparently affected by the use of UML*i* with respect to UML models, as can be observed, for instance, by comparing the UI diagram in Figure 4.2 and its corresponding class diagram in Figure 3.9. Therefore, this is an aspect of models of interactive systems that could be analysed in the metric study. However, despite this potential benefit of UML*i*, there is no well-established metric for measuring the complexity of visual languages [38]. Therefore, although not validated, two new metrics for visual complexity are proposed in this chapter along with the definition of the other metrics selected for the study, as described in the following section.

7.2 Metric Definitions

This section briefly describes the CK metrics, cyclomatic complexity and visual complexity metrics that are subsequently applied to the UML_noUI, UML_UI and UML*i*_UI models used in the metric study.

Structural Complexity The CK metrics are presented as a suite of 6 structural metrics used to quantify the complexity of classes and their relationships. LCOM, one of the CK metrics of the suite, however, is not considered in the case study since it could not be validated for class fault-proneness [8].

- *Weighted Methods per Class (WMC)*[1] is defined as the number of member functions (operations and attributes) of a class. This metric measures the complexity of an isolated class. The assumption regarding this metric is

[1]The methods were expected to be weighted by an surprisingly undefined "complexity" [24] which is unconsidered in this study.

that the more operations and attributes it possesses, the more complex the class.

- *Depth of Inheritance Tree of a Class (DIT)*[2] is defined as the maximum depth in the inheritance graph at which the class lies. This metric measures the complexity of a derived class, i.e., with respect to what it may inherit from its ancestors. The assumption regarding this metric is that the deeper in an inheritance hierarchy it is located, the more complex the class.

- *Number of Children of a Class (NOC)* is defined as the number of direct descendants of a class. This metric measures the complexity of a superclass, i.e., with respect to the potential difficulty of modifying and consequently testing, a class due to the possibility that these modifications can propagate to its children. The assumption regarding this metric is that the more numerous the children, the more complex the class.

- *Coupling Between Object Classes (CBO)* is defined as the number of classes to which the class is associated, plus the number of classes with which the class shares methods in action states. This metric measures the complexity of modifying and testing a class in relation to how changes to the class may propagate to classes that are not related to it by inheritance. The assumption regarding this metric is that very coupled classes are more fault-prone than less coupled classes.

- *Response For a Class (RFC)* is defined as the number of methods that may be executed in response to a message received by an object of that class. This metric measures the number of activities and action states reached by transitions triggered by operations belonging to the class. The assumption regarding this metric is that the more responses, the more complex the class.

Behavioural Complexity McCabe's cyclomatic complexity metric [87] is defined as the number of decisions (or predicates) in a control flow graph plus 1. For example, the cyclomatic complexity of the activity diagram in Figure 7.1 is 3 since it has 2 Branches representing decisions. Guarded Transitions may also

[2] The original name of DIT is preserved although calling it *Depth of a Class in the Inheritance Tree* is perhaps less ambiguous.

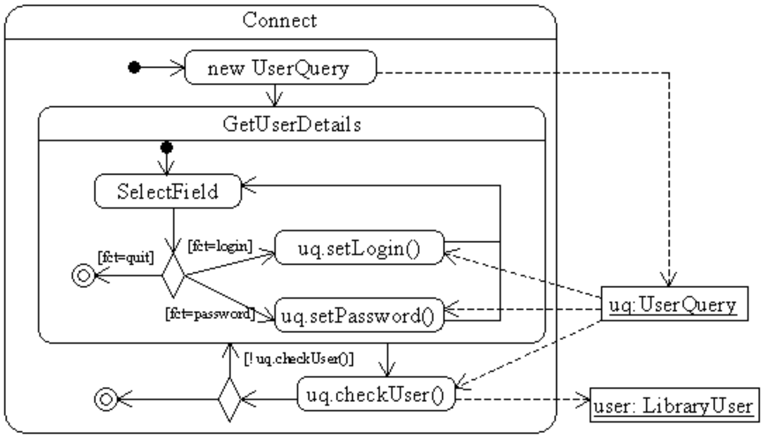


Figure 7.1: The `Connect` activity of `ConnectToSystem` without its UI.

be used to model decisions in activity diagrams. Since decisions are specified in behavioural models, this is a metric for behavioural complexity. Moreover, if the behavioural model is an activity diagram and A is an activity in this activity diagram, then $\nu(A)$ is the cyclomatic complexity of A. The reasoning behind this metric is that $\nu(A)$ corresponds to the number of possible execution paths specified in an activity. The assumption regarding this metric is that the higher the $\nu(A)$, the more complex the activity.

Visual Complexity. Diagrams in ARGO*i* are stored and exchanged using the Precision Graphics Markup Language (PGML) format [2]. Therefore, as PGML is a textual representation for the diagrams in ARGO*i*, the number of lines of code (LOC) of the PGML files is the metric for measuring the size of both the UML and UML*i* diagrams in this study. Size, however, may not be an appropriate metric for visual complexity since designs with long textual representations can be very simple ones. Therefore, the following metrics are used in this dissertation for measuring the visual complexity of the models.

- *Density of coupling between objects in diagrams (DCBOD).* This is defined as the ratio of the level of CBO in the models and the total number of LOC of the PGML files representing the diagrams. The non-validated assumption is that high densities indicate that more relevant structural specification, e.g., structural complexity, can be represented by fewer graphical elements than with low densities.

- *Density of cyclomatic complexity in diagrams (DCCD).* This is defined as the ratio of the cyclomatic complexity in the models and the total number of LOC of the PGML files representing the diagrams. The non-validated assumption is that high densities indicate that more relevant behavioural specification, e.g., behavioural complexity, can be represented by fewer graphical elements than with low densities.

The UMLi model of an interactive systems has the same size or is smaller than the UML model of the same system, as is shown later in Section 7.3.4. The use of size as a metric, however, is not considered in this study since the construction and further uses of UML and UMLi models can be partially supported by tools. In fact, computer-based tools may analyse big, non-complex models faster than small, complex models.

7.3 Models Used in the Metric Study

The models used in the metric study are those of the Library System partially presented throughout the dissertation. In the case of the metric study, they have been comprehensively modelled in ARGOi in order to specify the functionalities identified in the use case diagram in Figure 3.1. The actual UML_noUI, UML_UI and UMLi_UI models are available at `http://img.cs.man.ac.uk/umli/metrics`. Moreover, the models can be viewed and adapted using ARGOi, which is also publicly available from `http://img.cs.man.ac.uk/umli/software.html`.

Measuring the metrics in models is a straightforward task. A concern regarding the use of these design metrics is the production of models that have the same reuse strategy. In fact, different reuse strategies can significantly affect metrics [9]. In the case of the UML_UI and UMLi_UI models, an additional concern regarding the use of these design metrics is the production of models using two different notations to model a common set of properties from the specification of a system. Therefore, a systematic mapping strategy should be defined in order to ensure that the services of the UML_UI and UMLi_UI models are translated in a consistent way and that the same reuse strategy is adopted in all models.

7.3.1 UMLi_UI Models

The UMLi_UI model of the Library System is composed of many diagrams presented in Chapter 4. The class diagram in Figure 3.2 models the domain of the system. The UI diagrams in Figures 4.2 and 4.5 are elements of the set of UI diagrams modelling the structure of the system's UIs. The activity diagrams in Figures 4.6, 4.8 and 4.9 are elements of the set of activity diagrams describing the system's behaviour.

Concerning the `ConnectToSystem` service, it must be established which models are used to represent the service, defining in this way its scope. Thus, the class diagram in Figure 7.2, that is a subset of the class diagram in Figure 3.2, represents the domain required to support the `ConnectToSystem` functionality. It can be observed that the `getUser()` and `getLoans()` operations of the `UserQuery` class in Figure 3.2 are not represented in the `UserQuery` class in Figure 7.2. The entire set of UI diagrams of the service is composed of the UI diagram in Figure 4.2. Finally, the entire set of activity diagrams representing the behaviour of the service is composed of the activity diagram in Figure 4.8.

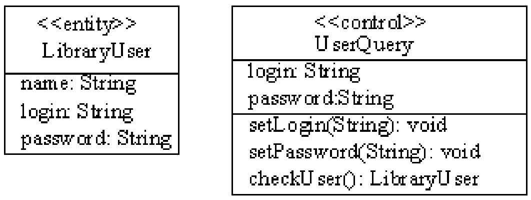


Figure 7.2: The `ConnectToSystem`-specific class diagram.

With respect to the metric study in this chapter, the UMLi_UI version of the Library System, including the `ConnectToSystem` service, is the model that has been designed directly. The UML_UI model has been translated from the UMLi_UI model, and the UML_noUI model has been translated from the UML_UI model. The mapping rules used to perform these translations are presented in the following sections where the translation of the UMLi_UI model of the `ConnectToSystem` service into the UML_UI and UML_noUI models of the service is described. The mapping rules are the same for the other services of the Library System, making unnecessary a discussion about the production of the models of the complete Library System. The mapping rules from the UMLi_UI model to the UML_UI model are presented in Section 7.3.2. The mapping rules from the

UML_UI model to the UML_noUI model are presented in Section 7.3.3.

7.3.2 Mapping UML*i*_UI Models into UML_UI Models

The standard UML diagrams in Figures 3.9 and 3.4 are the models resulting from a systematic translation of the UML*i* diagrams in Figure 4.2 and 4.8, respectively. Therefore, the mapping rules from UML*i*_UI models into UML_UI models can be explained with reference to the Figures 3.9, 3.4, 4.2 and 4.8 as follows.

InteractionClasses to Classes. The InteractionClasses in the user interface diagram in Figure 4.2 are mapped into Classes in the class diagram in Figure 3.9. This is a natural mapping since the UML*i* InteractionClass is a subclass of UML Class as shown in Figure 4.15. As a consequence of this mapping we can observe the following.

- *Placement to Composition.* In UML*i*, an InteractionClass which is not a FreeContainer must be associated with a Container. Thus, the association of an InteractionClass with its Container is specified in UML*i* by the placement of the InteractionClass into the Container in the diagram. In UML, however, this association is specified by a composition. Therefore, a composition is created from each `InteractionClass` in Figure 3.9 that is not a `FreeContainer` to its immediate `Container`. For instance, `Cancel` is placed in `Options` in Figure 4.2, so there is a composition between `Cancel` and `Options` in Figure 3.9.
- *Visible(), Active() and InvokeAction() operations to explicit class methods.* These operations originally embedded in the UML*i* metamodel are explicitly modelled in the UML presentation model in Figure 3.9.

Interaction Object Flow to Object Flow. Interaction object flows in Figure 4.8 were translated into object flows in Figure 3.4. Despite missing the information as to which role each Object can play in a UI, this is a natural mapping since an interaction object flow is an object flow of a ClassifierInState of type InteractionClass.

Interaction object flow stereotypes into fragments of a standard activity diagram. There is no one-to-one mapping between UML*i* constructs and

UML constructs. Thus, the mapping of each stereotype must be individually explained.

- The ≪*Presents*≫ stereotype in Figure 4.8 specifies that the `ConnectUI` FreeContainer is a presentation unit. This means that the widgets directly and indirectly contained by `ConnectUI` must be instantiated (if not previously explicitly instantiated) and must be made visible when the `Connect` Activity is reached as specified in UML by the `InitiateConnectUI` Activity in Figure 3.4 and refined in Figure 3.5(a). Further, these widgets must be made invisible and destroyed when the `Connect` Activity is left as specified in UML by the `TerminateConnectUI` Activity in Figure 3.4 and refined in Figure 3.5(b).

- The ≪*Cancels*≫ stereotype behaviour in Figure 4.8 is modelled in Figure 3.4 by the `Cancel` Object that is made active immediately after the instantiation of `ConnectUI`.

- The ≪*Interacts*≫ stereotype can be associated with a Container, PrimitiveInteractionClass or ActionInvoker. If the stereotype is associated with a `Container`, this means that contained InteractionClasses are made active when the associated Activity or ActionState is reached. If associated to a PrimitiveInteractionClass or ActionInvoker, this means that the Object is made active (if its Container was not previously activated by another ≪*Interacts*≫) and ready to interact with a user through the `getValue()` and `setValue()` operations.

- The ≪*Confirms*≫ stereotype in Figure 4.8 is mapped into three sequential ActionStates associated to the `OK` ActionInvoker in Figure 3.4. The first ActionState activates the associated ActionInvoker, the second waits for the performance of the invokeAction(), and the last deactivates the ActionInvoker. Therefore, the performing of an `invokeAction()` is responsible for *confirming* the end of the selection state in Figure 3.4.

- The ≪*Activates*≫ stereotype, not used in Figure 4.8, is mapped into three sequential ActionStates, as in ≪*Confirms*≫. The important characteristic of ≪*Activates*≫, though, is that these sequential ActionStates are placed before the Activity to be triggered by the associated ActionInvoker.

7.3.3 Mapping UML_UI models into UML_noUI Models

The UML activity diagram in Figure 7.1, which does not encompass any UI specification, is the diagram resulting form a systematic translation of the UML activity diagram in Figure 3.4. Further, the class diagram in Figure 3.9 is not translated at all into the UML_noUI models of the `ConnectToSystem` service since it is a specific part of the UI specification in UML_UI. The mapping rules from UML_UI to UML_noUI can be explained as follows.

No Modelling of User Interface Presentations. The user interface presentation in Figure 3.9 is not considered as part of the UML_noUI models.

Deletion of interaction object flows. The ClassifierInStates that are of any type defined in user interface presentations, e.g., the classes in the class diagram in Figure 3.9, along with their ObjectFlowStates are removed from activity diagrams. For example, the `ok` and `cn` objects of types `OK` and `Cancel` in Figure 3.4 are removed in Figure 7.1.

Deletion of Activities associated only with interaction object flows. The `cn.setActive(true)` and `cn.invokeAction()` ActionStates in Figure 3.4 are example of ActionStates related to `cn` of type `Cancel` specified as an InteractionClass that are removed in Figure 7.1. Moreover, `InitiateConnectUI` and `TerminateConnectUI` Activities are also removed in Figure 7.1 since their ActionStates are related to interaction object flows, as can be observed in Figure 3.5.

Replacement of Forks and Transitions related to *cancel*, *order independent*, *optional* and *repeatable* behaviours by a Branch. The Forks in Figure 3.4 is removed in Figure 7.1. In fact, the Fork leaving `new UserQuery` is related to a *cancel* behaviour and the other two Forks in Figure 3.4 are related an *order independent* behaviour.

Deletion of invocations of methods of InteractionClasses. The `uq.setPassword(pt.getValue())` ActionState in Figure 3.4 is replaced by the `uq.setPassword()` ActionState in Figure 7.1 that does not invoke the `getValue()` of `pt`, an Object of type `PasswordText` defined in a user interface presentation.

7.3.4 Resulting Models of the Metric Study

Six sets of files containing a textual representation of the UML_noUI, UML_UI and UMLi_UI models of the `ConnectToSystem` service and of the Library System were produced using ARGOi. Table 7.1 presents the consolidated size in terms of LOC of these sets. There, it can be observed the same pattern in terms of size for models of `ConnectToSystem` and the complete Library System. Indeed, the size of the textual representations of UML_UI models are much higher than in UML_noUI models for models with the same scope. Furthermore, part of this increase of size in UML_UI models with respect to UML_noUI models is reduced in UMLi_UI models. Thus, the hypotheses of the metric study would be confirmed if size was an appropriate metric for complexity. However, size usually does not say much about how difficult it is to construct a model of an interactive system or how difficult it is to understand such models. By contrast, the metrics in this dissertation are designed to quantify the inherent difficulties of constructing and understanding models, since they measure many dimensions of the complexity associated with the models. Therefore, a metric analysis of the six sets of files in Table 7.1 is performed in the following sections. Nevertheless, the LOC of the PGML files are considered in the following metrics when it is used to compose the DCBOD and DCCD metrics.

Scope	Model	LOC of XMI	LOC of PGML
`ConnectToSystem`	UML_noUI	845	1,317
	UML_UI	5,308	9,801
	UMLi_UI	1,329	2,286
Library System	UML_noUI	9,608	14,959
	UML_UI	58,029	125,658
	UMLi_UI	16,836	31,325

Table 7.1: Size of the models of the `ConnectToSystem` service.

7.4 A Metric Assessment of the `ConnectToSystem` Service Models

The analyses in this section provide an insight into the impact of the many dimensions of complexity in models of a typical interactive application. Therefore, results are superficially analysed in this section, while in-depth analyses of each metric over the models of the complete Library System are held over to Section 7.5.

Table 7.2 presents the measurements of the CK metrics of the models of the `ConnectToSystem` service from where the following can be observed:

Model	# Classes	Data	WMC	DIT	NOC	CBO	RFC
UML_noUI	2	Mean	4.00	0.00	0.00	1.00	1.50
		Sum	8	0	0	2	3
UML_UI	15	Mean	0.53	1.27	0.80	1.33	2.33
		Sum	8	19	12	20	35
UML*i*_UI	12	Mean	0.67	0.00	0.00	1.67	1.18
		Sum	8	0	0	20	8
UML_UI/ UML_noUI	7.5	Mean	**0.13**	—	—	1.33	1.56
		Sum	1	—	—	**10**	**11.67**
UML_UI/ UML*i*_UI	1.25	Mean	0.8	—	—	0.8	**3.50**
		Sum	1	—	—	1	**4.37**

Table 7.2: CK metrics of the models of the `ConnectToSystem` service. Values in bold are rations of metrics higher than 2 or lower than 0.2.

- Measurements of WMC in the models are restricted to the classes originally belonging to the UML_noUI since the consolidated WMC in the three models are the same (sum of WMC of 8).

- On average each class in UML_UI has around one ancestor (mean DIT of 1.27) and one child (mean NOC of 0.80). This means that the inheritance tree is shallow, as can be observed in Figure 3.9. However, this shallowness does not prevent an increase in the consolidated DIT and consolidated NOC for classes in UML_UI when compared with these measurements for the classes in UML_noUI (consolidated DIT from 0 to 19 and consolidated NOC from 0 to 12). In UML*i*_UI models, however, the effects of adding UIs in UML_UI are completely neutralised in UML*i*, which returns the measurements of the consolidated DIT and consolidated NOC in UML*i*_UI to 0, the same value as in UML_noUI.

- The mean CBO of classes in UML_noUI is slightly lower than the mean CBO of classes in UML_UI and UML*i*_UI. However, the consolidated CBO of classes in UML_UI, which has the same measurement for the classes in UML*i*_UI, is 10 times the consolidated CBO of the classes in UML_noUI.

- In absolute terms, the consolidated RFC of classes in UML_UI is 11.67 times the consolidated RFC of classes in UML_noUI. Moreover, even considering that there are 7.5 times more classes in UML_UI than classes in UML_noUI, the mean RFC of classes in UML_UI is just 1.56 times higher than of the classes in UML_noUI. The consolidated RFC of 35 for the classes in UML_UI is significantly reduced to a consolidated RFC of 8 for the classes in UML*i*_UI.

Table 7.3 presents the behavioural and visual metrics of the models of the `ConnectToSystem` service where the following can be observed:

Model	Behav. Complexity	Visual Complexity	
	$\nu(ConnectToSystem)$	DCBOD	DCCD
UML_noUI	3	0.0015	0.0023
UML_UI	5	0.0022	0.0006
UML*i*_UI	4	0.0087	0.0017
UML_UI/UML_noUI	1.67	1.48	**0.25**
UML_UI/UML*i*_UI	1.25	**0.26**	0.32

Table 7.3: Behavioural and visual metrics of the models of the `ConnectToSystem` service.

- The increase of 67% in $\nu(ConnectToSystem)$ of the activity diagrams in UML_UI with respect to the $\nu(ConnectToSystem)$ of the activity diagrams in UML_noUI is a significant increase in behavioural complexity. This increase in complexity is a result of the necessity of specifying two aspects of UIs: system feedback to users in case of invalid user details and a possibility to cancel the service. The implicit modelling of the cancelling of the service in UML*i* reduces the impact of modelling behavioural aspects of UIs in activity diagrams from an original increase of 67% observed between the $\nu(ConnectToSystem)$ of activity diagrams in UML_UI and UML_noUI to a final increase of 33% between the $\nu(ConnectToSystem)$ of activity diagrams in UML*i*_UI and UML_noUI.

- DCBOD in the UML*i*_UI model is significantly higher than in the UML_noUI and UML_UI models, suggesting that visual complexity is lower in the UML*i*_UI model than in the UML_noUI and UML_UI models.

- DCCD in the UML_noUI model is 3.8 times higher than in the UML_UI model. However, DCCD in UML*i*_UI model is 2.83 times higher than in the UML_UI model. These measurements suggest that visual complexity in UML_UI is higher than in the UML_noUI and UMLi_UI models. However, the UML_noUI is less visually complex than UMLi_UI.

The measurement of the metrics of the complete Library System is the next step in order to test Hypotheses 1 and 2 using more generic measurements than those presented in this section.

7.5 A Metric Assessment of the Library System Models

Table 7.4 introduced in page 173 presents the values for the CK metrics measured in the models of the Library System. Table 7.5 introduced in page 178 presents the values for the behavioural and visual metrics measured in the models of the Library System. There, the consolidated metrics measured in UML_UI quantify dimensions of complexity not lower than those quantified by the values measured for the same consolidates metrics in UML_noUI, confirming Hypothesis 1. Moreover, the values for the consolidated metrics measured in UMLi_UI quantify dimensions of complexity not higher than those quantified by the values measured for the same consolidated metrics in UML_UI, confirming Hypothesis 2.

The following three analyses are provided for each selected metric in order to characterise better the conclusion above.

i. An analysis of the consolidated metrics of the complete Library System identifies which properties of the UML models are substantially affected by the modelling of UIs. This analysis consists of a comparison between the CK metrics of UML_noUI and UML_UI models in Table 7.4 and a comparison between the metrics of UML_noUI and UML_UI models in Table 7.5.

ii. An analysis of the consolidated metrics of the complete Library System identifies which properties of models are substantially affected by the use of UMLi rather than UML for modelling interactive applications. This analysis consists of a comparison between the CK metrics of UML_UI and UMLi_UI models in Table 7.4 and a comparison between the metrics of the UML_UI and UMLi_UI models in Table 7.5.

Model	# Classes	Data	WMC	DIT	NOC	CBO	RFC
UML_noUI	14	Mean	3.43	0.14	0.14	2.79	4.36
		Median	2.5	0	0	2	2.5
		Max.	9	1	2	7	10
		Min.	0	0	0	0	0
		Std.Dev.	2.56	0.36	0.53	1.76	4.09
		Sum	48	2	2	39	61
UML_UI	91	Mean	0.58	1.40	0.86	2.76	6.76
		Median	0	2	0	2	5
		Max.	9	2	27	7	63
		Min.	0	0	0	0	0
		Std.Dev.	1.59	0.73	4.56	1.72	8.40
		Sum	53	127	78	251	615
UMLi_UI	88	Mean	0.54	0.02	0.02	2.85	1.18
		Median	0	0	0	2	0
		Max.	9	1	2	7	26
		Min.	0	0	0	0	0
		Std.Dev.	1.60	0.15	0.21	1.67	3.50
		Sum	48	2	2	251	104
UML_UI/ UML_noUI	**6.50**	Mean	**0.17**	**9.77**	**6.00**	0.99	1.55
		Sum	1.10	**63.50**	**39.00**	**6.44**	**10.08**
UML_UI/ UMLi_UI	1.03	Mean	1.07	**61.41**	**37.71**	0.97	**5.72**
		Sum	1.10	**63.50**	**39.00**	1.00	**5.91**

Table 7.4: CK metrics of the models of the Library System. Values in bold are rations of metrics higher than 2 or lower than 0.2.

iii. An analysis of the metrics of the `ConnectToSystem` service in Section 7.4 against the metrics of the complete case study in Section 7.5 identifies the accumulative effects on metrics of the UML_noUI, UML_UI and UMLi_UI models. This analysis consists of a comparison between the CK metrics of the case study in Table 7.4 and the CK metrics of the `ConnectToSystem` service in Table 7.2, and of a comparison between the cyclomatic complexity, DCBOD and DCCD of the complete case study in Table 7.5 and the cyclomatic complexity, DCBOC and DCCD of `ConnectToSystem` service in Table 7.3.

Finally, a conclusion in terms of complexity is suggested from (i), (ii) and (iii) for each metric.

Structural Complexity Table 7.4 presents the values for the CK metrics measured in the models of the Library System.

- *Analysis of WMC.*

i. The WMC of a class reused verbatim from a pre-defined class is zero. Thus, the low WMC of the classes in UML_UI with respect to the WMC of the classes in UML_noUI indicates that widgets, which are the 77 classes that UML_UI has in addition to the 14 classes of UML_noUI, are often reused verbatim from toolkits rather than developed from scratch.

ii. The slight decrease of mean WMC of the classes in UML_UI with respect to the WMC of classes in UMLi_UI (from 0.58 to 0.54) is insignificant since the mean WMCs in both models are low.

iii. The increase on the mean WMC of classes in UMLi_UI with respect to the mean WMC of classes in UML_UI in Table 7.2 (from 0.54 to 0.67) is an insignificant effect specific to the `ConnectToSystem` service. Indeed, it is insignificant since the consolidated WMC is the same in all models of the service in Table 7.2. Further, it is specific to the `ConnectToSystem` service since the mean WMC of classes in UMLi_UI is lower than of the classes in UML_UI in the context of the Library System.

The measurements for WMC do not suggest a significant increase in the complexity of the UML models of the Library System when including UIs although they do not suggest any decrease in complexity, confirming Hypothesis 1. The measurements for WMC suggest a slight decrease in the complexity of models when using UMLi rather than UML, neither confirming nor conflicting with Hypothesis 2.

- *Analysis of DIT.*

i. Although the mean DIT of classes in UML_UI is low, the mean DIT of classes in UML_noUI is much lower. The ratios between the DIT of classes in UML_UI and UML_noUI is high due to the fact that the mean DIT of classes in UML_noUI is almost zero (0.14).

ii. The ratio of DIT of classes in UML_UI and in UMLi_UI is the same as the ratio of DIT of classes in UML_UI and in UML_noUI. This means that UMLi is able to neutralise the effects on DIT of modelling the UI in UML.

iii. The slight increase in the mean DIT of classes in Table 7.4 in comparison with the mean DIT of classes in Table 7.2 indicates that there are proportionally more classes with a DIT of 2 in the case study than in the service. Actually, DIT in UML_noUI appears to be generally low, but it is zero for the `ConnectToSystem` service.

The measurements for DIT suggest an increase in the complexity of the UML models of the Library System when including UIs, confirming Hypothesis 2, and a complete neutralisation of the effects in DIT of modelling UI when using UMLi, confirming Hypothesis 2. Furthermore, for a more concrete interaction model, DIT would probably be a decisive aspect of complexity to impact the specification of UIs.

- *Analysis of NOC.*

i. The high standard deviation of NOC of classes in UML_UI (4.56) indicates that the NOCs of few of the classes in UML_UI are significantly higher than the NOCs of the other classes in the same model. The low mean NOC of classes in UML_UI indicates that the inheritance tree in the model is shallow, but also confirms that the classes with high NOC are few. These metrics indicate that the complexity in terms of inheritance is centred in a few classes, which are probably those of the class library.

ii. The reduction of the consolidated NOC of classes in UMLi_UI with respect to the consolidated NOC of classes in UML_UI (from 78 to 2) is the same reduction of the consolidated NOC of classes in UML_noUI with respect to the consolidate NOC of classes in UML_UI. This means that UMLi is able to neutralise the effects on NOC of modelling the UI in UML.

iii. The profiles of the statistics of the NOC metric in the service and in the case study are similar.

The effects on NOC of modelling UIs using UMLi rather than UML are similar to these effects on DIT. Thus, recalling the analyses of DIT, Hypotheses 1 and 2 are also tested for NOC. However, it must be observed that the overall impact of NOC on complexity may not be significant.

- *Analysis of CBO.*

  i. The increase in the consolidated CBO of classes in UML_UI with respect to the consolidated CBO of classes in UML_noUI is proportional to the increase in the number of classes between these models (approximately 6), which is significant and noteworthy.

  ii. The consolidated CBO of 251 is the same in the classes of both UML_UI and UML*i*_UI models. The difference in the means of these metrics is due to the abstract `InteractionClass`, `InvokeActionInteractionClass` and `PrimitiveInteractionClass` classes, which are explicitly specified in the UML models and implicitly specified in the UML*i* metamodel.

  iii. The comparison between the CBOs in Tables 7.2 and 7.4 demonstrates that the CBO of the classes in the `ConnectToSystem` service is slightly lower than the CBO of classes of the Library System. However, CBO is high even for `ConnectToSystem`.

  The measurements of CBOs of the UML_noUI and UML_UI models suggest a significant increase of 543% in the CBO dimension of complexity of the UML models of the Library System when including UIs, confirming Hypothesis 1. Furthermore, it suggests that complexity related to CBO is not affected at all by UML*i* even with a model 29% of the size of the UML_UI model, as indicated in Table 7.1. However, the metrics also show that CBO has not increased in UML*i*_UI with respect to UML_UI, neither confirming nor conflicting with Hypothesis 2.

- *Analysis of RFC.*

  i. RFC is 10.08 times higher in classes of UML_UI than in classes of UML_noUI. The mean RFC of classes of UML_noUI (4.36) is high but the mean RFC of classes of UML_UI is substantially higher (6.76). In fact, the mean RFC of interaction classes is 1.55 times higher than the mean RFC of domain classes.

  ii. RFC of classes in UML*i*_UI has been reduced by 83% with respect to the RFC of classes in UML_UI since much of the behaviour of classes in

UMLi_UI is embedded within the UMLi constructs, making the models more straightforward and easier to maintain. Figure 7.3 presents a graphical representation of the distribution of RFC per number of classes. In this distribution, small areas are better than big areas since low RFCs are better than high RFCs.

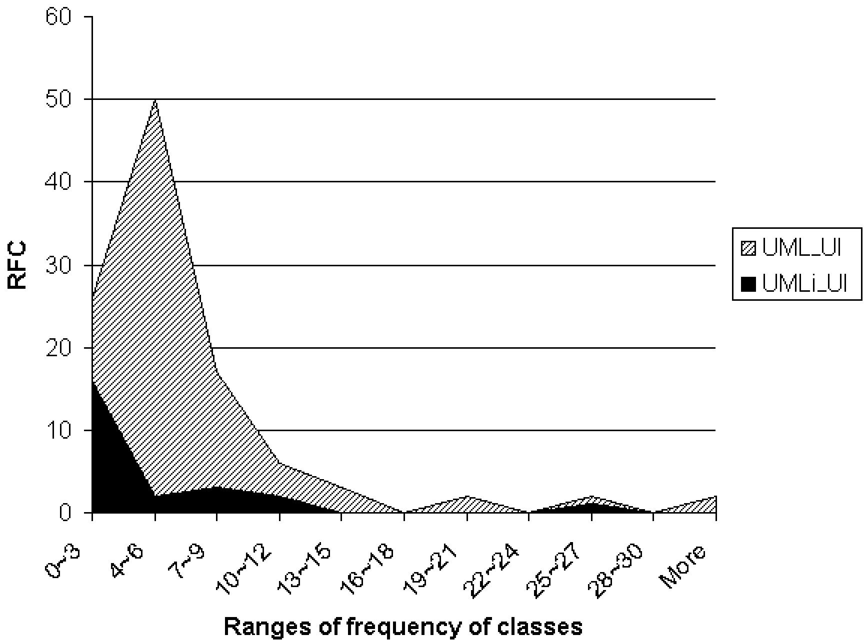


Figure 7.3: Distribution histogram comparing the RFC of the UML_UI and UMLi_UI models.

iii. The high RFC in Table 7.2 is an indication that RFC could represent a potential problem for UI designs. The high RFCs in Table 7.4 have confirmed our suspicions.

The measurements for RFC suggest a substantial increase of 908% in the RFC dimension of complexity of the UML models of the Library System when including UIs, confirming Hypothesis 1. The high overall RFCs confirm the suggestion that UML may be inappropriate for modelling actions in both domain and interaction classes, but it suggests how complex the situation can become when interaction classes are specified. The main reason for this may be the non-declarative approach used by activity diagrams to represent actions. Using UMLi, however, a significant reduction of 87% in

Model	Behav. Complexity	Visual Complexity	
	$\nu(LibrarySystem)$	DCBOD	DCCD
UML_noUI	50	0.0026	0.0033
UML_UI	77	0.0020	0.0006
UMLi_UI	66	0.0080	0.0021
UML_UI/UML_noUI	1.54	0.77	0.18
UML_UI/UMLi_UI	1.17	0.25	0.29

Table 7.5: Behavioural and visual metrics of the models of the Library System.

RFC can be observed, confirming Hypothesis 2. Therefore, the reduction in RFC is a significant achievement of UMLi.

Behavioural Complexity Table 7.5 presents a summary of the cyclomatic complexity $\nu(LibrarySystem)$ measured in the complete case study.

i. The $\nu(LibrarySystem)$ is 54% higher in UML_UI than in UML_noUI. This increase of $\nu(LibrarySystem)$ in UML_UI with respect to UML_noUI is due to the specification of the behaviour of the UI to provide: feedback for user actions; feedback to some users about actions performed in methods of domain classes; support for cancellation actions. Indeed, different action results may require different responses that must be specified in behavioural models.

ii. The $\nu(LibrarySystem)$ is 14% lower in UMLi_UI than in UML_UI. This is due to the ≪*cancels*≫ and ≪*confirms*≫ stereotypes that eliminate the necessity of specifying how activities can be cancelled by users and how OptionalSelectionStates can be confirmed by users. This reduction may have a significant impact in the design of large-scale interactive systems since it provides a uniform treatment for the cancelling of tasks in order to allow users to reverse their actions, a UI usability requirement [126].

iii. The UML_UI/UML_noUI and UML_UI/UMLi_UI rations of $\nu(ConnectToSystem)$ in Table 7.3 and $\nu(LibrarySystem)$ in Table 7.5 are very similar, suggesting an increase in behavioural complexity when adding UI components in the models, and a partial compensation for this increased complexity to an intermediate value for the behavioral complexity when using UMLi to specify UI concepts.

The measurements for $\nu(LibrarySystem)$ suggest a slight increase in the complexity of UML models when including UIs, confirming Hypothesis 1. These measurements also suggest a reduction on the increase in the complexity of models above when using UML*i*, but not compensating the increase caused by the modelling of UI specifications, confirming Hypothesis 2.

Visual Complexity The values of the DCBOD and DCCD metrics measured in the Library System are presented in Table 7.5. It may be appropriate to recall the assumption of these metrics that say that high values for the metrics indicate low complexity of the diagrams.

i. The DCBOD is insignificantly affected when comparing the values for UML_UI and for UML_noUI. The DCCD, however, is significantly affected by the modelling of UI specifications in UML_UI, resulting in values 5 times lower than in UML_noUI, which means 5 times more complex.

ii. DCCD in UML*i*_UI (0.0021) reduces the high visual complexity in UML_UI represented by its low DCCD in UML_UI (0.0006). Thus, DCCD is improved in UML*i*_UI with respect to UML_UI becoming 3.5 times higher. There is also an improvement of DCBOD in UML*i*_UI that is more remarkable than that of the DCCD in UML*i*_UI since it reduces by 75% the DCCD dimension of complexity by increasing DCBOD (from 0.0020 to 0.0080).

iii. There is a proportionality between most of the measurements of DCBOD and DCCD in Tables 7.3 and 7.5. Actually, the DCBOD in UML_noUI is the only metric that is particularly low in the `ConnectToSystem` (0.0015) with respect to the complete Library System (0.0026). As a consequence, the DCBOD ration between UML_UI and UML_noUI models are different in the tables. This difference, however, does not appears to be significant since the DCBOD in UML_noUI has a low variation of less than 0.7.

The measurements of DCBOD and DCCD suggest that the modelling of UI components in UML models results in diagrams that required more graphical elements to represent structural and behavioural complexity than required by the domain part of the models, confirming Hypothesis 1. The measurements of DCBOD and DCCD, however, suggest that the capability of UML*i* to visually represent both structural and behavioural complexity is higher than that of UML.

In fact, with a fixed number of graphical elements, UMLi should be able to represent more relevant information than UML, confirming Hypothesis 2.

7.6 Summary

The modelling of the `ConnectToSystem` service using standard UML has demonstrated that the design of UIs has increased by 650% the number of classes required for modelling such a simple service. Moreover, the metrics for `ConnectToSystem` have demonstrated that this increase in classes was followed by an increase in the measurements of metrics representing the structural, behavioural and visual complexity of the model. This supports Hypothesis 1, which says that complexity of UI designs can significantly increase the complexity of standard UML models that omit UI aspects. Furthermore, these metrics have demonstrated that despite the overall increase in complexity, that response for a class (RFC) (900% of increase in structural complexity), coupling between object classes (CBO) (1066% of increase in structural complexity) and density of cyclomatic complexity in diagrams (DCCD) (283% of increase in visual complexity) were the most affected aspects of the models.

The modelling of the `ConnectToSystem` service in UMLi_UI has demonstrated that UMLi requires an equivalent number of classes to that required by UML_UI for modelling the service. However, the increase in the number of classes in UMLi_UI with respect to UML_noUI has not been followed by an increase of the measurements of metrics representing the dimensions of the structural, behavioural and visual complexity of the models. In fact, the measurements of the metrics in UMLi_UI and UML_UI models indicate an overall reduction in complexity of the UMLi_UI models. This supports Hypothesis 2, which says that standard UML models are structurally, behaviourally and visually more complex than UMLi models when describing the same set of properties of an interactive system.

Therefore, useful metrics have been obtained by reusing the modelling techniques applied in the `ConnectToSystem` service in the eight additional services available in the Library System described in Section 3.1. These metrics have confirmed the results produced by the modelling of the `ConnectToSystem` service confirming Hypotheses 1 and 2 for the Library System. In particular, the metric study has demonstrated that structural and visual complexity are much more a concern when modelling UIs than behavioural complexity (it is important to observe that structural complexity is represented in both structural and behavioural diagrams of UML). More precisely, the RFC is the key metric for identifying the difficulty of specifying actions in UML. Indeed, with a non-declarative approach used in their behavioural diagrams, UML models tend to be very complex. Further, although not increasing as much as RFC, CBO is also significantly affected by the modelling of UIs. In this case, CBO is affected more by the increase in the number of interaction classes demanding a lot of collaboration among them than by the increase in the collaboration between domain and interaction classes. Indeed, the value of CBO of classes in UML_UI and UML_noUI are the same.

In terms of the development of interactive systems, the results in this chapter demonstrated that UML*i* reduced the complexity of modelling the Library System when compared with UML. Significant reductions in structural complexity (reduction of 87% in RFC) and visual complexity (increase of 402% in DCBOD and 345% in DCCD) along with considerable reductions in behavioral complexity (reduction of 14% in cyclomatic complexity) were achieved in UML*i* models by improving the support of UML for UI design. Therefore, where developers do not model UIs, they are neglecting to provide guidance for the implementation of a substantial and complex part of typical interactive systems. Indeed, this may be a reason why UIs are often implemented in an *ad hoc* way.

In terms of user interface design assessments, the metric study in this thesis is a contribution on its own. Indeed, it demonstrates that quality in UI designs can be evaluated even before the generation of any other artifact from the models such as prototypes or UI code.

Chapter 8

Conclusions and Future Work

The thesis concludes with a review of the work presented and an assessment of the extent to which the objectives set out in Section 1.5.2 have been met. The principles set out in Section 1.5.1 are revisited. Some reactions of the research community to the ideas of the dissertation are discussed. Finally, directions for future work are suggested.

8.1 Work Overview and Contributions

Many issues were addressed during the process of developing and assessing UML*i*. The resulting research contributions are presented along with an overview of the work developed in this thesis.

Chapter 2 presented a survey of the MB-UIDE literature identifying the aspects usually described by user interface models in MB-UIDEs. This chapter has contributed through the development of a comparison framework for MB-UIDEs that has identified a set of 15 common modelled characteristics of user interfaces. The framework is composed of the elements presented in Tables 2.1 and 2.2. Table 2.1 presents four categories of model often used to describe relevant aspects of user interfaces: *application models*, *task-dialogue models*, *abstract presentation models* and *concrete presentation models*. Table 2.2 presents 15 categories of construct used to compose the models in Table 2.1. Table 8.1 presents the compositions of UML*i* constructs representing the categories of constructs in Table 2.2.

Comp. model	Construct	UML*i* construct(s)
AM	CLASS	Class
	ATTR	Attribute
	OPER	Operation
	RELAT	Association
TDM	TASK	UseCase and Activity connected by a Realisation
	GOAL	name of UseCase
	ACTION	ActionState
	SEQ	combination of Transition(s) and PseudoState(s)
	PRE	OCL expression in Activity
	POST	OCL expression in the default Transition leaving an Activity
APM	VIEW	FreeContainer
	AIO	PrimitiveInteractionClass or ActionInvoker
CPM	WINDOW	Class bound to FreeContainer in a pattern's Framework
	CIO	Class bound to PrimitiveInteractionClass or ActionInvoker in a pattern's Framework
	LAY	none

Table 8.1: UML*i* user interface model.

Chapter 3 presented a modelling study where the the facilities provided by the standard UML were exploited to model the user interface aspects identified in the comparison framework for MB-UIDEs presented in Chapter 2. The major contribution of this modelling study is the identification of the weaknesses of UML for modelling user interfaces, outlined as *UI Modelling Difficulties*. Thus, concerning the behavioral aspects of user interfaces, the modelling study demonstrated the difficulty of:

- modelling tasks using use cases, as suggested by UI Modelling Difficulty 1;
- identifying application entry-points, as suggested by UI Modelling Difficulty 2;
- specifying actions that instances of interaction classes can perform when collaborating with other interaction classes and with domain classes, as suggested by UI Modelling Difficulty 3;
- specifying some categories of interactive behaviours, e.g., optional and order independent behaviours, using activity diagrams, as suggested by UI Modelling Difficulty 4;
- specifying temporal dependencies using sequence diagrams, as suggested by UI Modelling Difficulty 7.

Concerning the structural aspects of user interfaces, the modelling study demonstrated the difficulty of:

- identifying containment among interaction classes, as suggested by UI Modelling Difficulty 5;
- identifying abstract roles that interaction classes can play in user interfaces, e.g., displaying information to users, receiving information from users, triggering actions, etc., as suggested by UI Modelling Difficulty 6.

Chapter 4 proposed UML*i*, a conservative extension of UML, that provides new constructs and models for coping with the UI Modelling Difficulties of UML identified in Chapter 3. The UML*i* proposal is the contribution of this chapter to the thesis. A description of how the UI Modelling Difficulties are addressed in UML*i* is used to summarise the new features of the proposal.

- UI Modelling Difficulty 1 is addressed by well-established links between use case diagrams and activity diagrams, which explain how user requirements identified during requirements analysis are described in the application design.
- UI Modelling Difficulty 2 is addressed by the InitiateInteraction construct in the UML*i* activity diagram that provides a way for modelling application entry-points.
- UI Modelling Difficulty 3 is addressed by the use of stereotypes in interaction object flows, which simplify the specification of the relationships between visual components of the user interface and domain objects.
- UI Modelling Difficulty 4 is addressed by the use of selection states in activity diagrams, which provide a simplified modelling of interactive system behaviour.
- UI Modelling Difficulty 5 is addressed by the UML*i* user interface diagram introduced for modelling abstract user interface presentations that simplifies the modelling of the use of visual components (widgets). The notation for Containers facilitates the grouping of widgets.

- UI Modelling Difficulty 6 is addressed by the notation for InteractionClasses that facilitates the visual identification of the abstract roles of widgets in user interfaces.

Chapter 4 also described how the UML*i* constructs were specified in terms of the standard UML metamodel.

Chapter 5 proposed a semantics for UML*i* based on the UML metamodel [99] with the specific aim of providing a precise meaning for the UML*i* constructs. Moreover, a one step model checking of complete specifications of interactive systems is an additional contribution achieved as a side-effect of the provision of a formal semantics for UML*i*. The contributions of this chapter can be summarised as follows.

- The development of a semantics for UML*i* based on the LOTOS specification language [16, 63], which provided an opportunity to bring the formalisation of interactive systems proposed by Markopoulos [82] and Paternò and Faconti [104] into the context of UML. Thus, the semantics of the InteractionClass construct of UML*i* was defined as an ADC interaction [82], demonstrating that the proposed UML*i* semantics is in line with the requirements for developing user interfaces.

- The development of a strategy for mapping UML models into LOTOS specifications has demonstrated how some structural and behavioural constructs of UML could be mapped into complete and correct LOTOS specifications. Moreover, many potential problems resulting from the design of UML*i* models such as deadlocks, livelocks and unreached states can be identified using LOTOS tools, as discussed in Section 5.6.

Chapter 6 presented the design and implementation of ARGO*i*, a UML*i*-based modelling environment. The main contributions of the effort of designing and implementing ARGO*i* can be summarised as follows.

- ARGO*i* is an extended version of Argo/UML [119], a generic UML-based tool. Thus, the implementation of ARGO*i* demonstrated that concerning the notation and metamodel, UML*i* is an conservative extension of UML that can be implemented in generic UML-based tools;

- ARGO*i* identified two tool facilities (wizards) that can be implemented for UML*i* in order to simplify some tasks frequently performed during the modelling of user interfaces: the *temporal-relation wizard* that exploits the use of the UML*i* SelectionStates (Section 6.4) can be used to model common UI behaviour; The *integration wizard* (Section 6.6) can be used to preserve, when modelled, the integration between interaction and domain Classes.

- The use of ARGO*i* has demonstrated that UML*i* is an conservative extension of UML. Thus, UML*i* models can be developed from UML models developed originally in Argo/UML, a standard UML tool.

Chapter 7 described a metric evaluation of UML*i* models when compared with their corresponding models described using standard UML. The metric study compared the cost in terms of design metrics of modelling an interactive system with UML*i* and standard UML. Two major results of the metric study constitute the main contribution of this chapter.

- One part of the metric study demonstrated that the standard UML does not scale up well for modelling the Library System, a typical interactive application. In fact, the development of a case study comparing the cost in terms of design metrics of modelling the Library System with and without user interface concerns demonstrated that all the metrics considered in the study were affected. The metrics also demonstrated that despite the overall increase in complexity, that RFC, CBO, DCBOD and DCCD were the affected metrics.

- The other part of the metric study demonstrated that UML*i* reduced the complexity of modelling the Library System when compared with UML. Significant reductions in structural complexity (reduction of 87% in RFC) and visual complexity (increase of 402% in DCBOD and 345% in DCCD) along with considerable reductions in behavioral complexity (reduction of 14% in cyclomatic complexity) were achieved in UML*i* models by improving the support of UML for UI design.

8.2 Revisited UML*i* Principles

The UML*i* principles for guiding the integration of interface models with UML introduced in Chapter 1 are as important as the contributions since they have established the context for the work described in this dissertation. Therefore, they are now revisited to illustrate how UML*i*, as specified in Chapters 4 and 5, is in conformance with them:

Principle 1 The UML*i* proposal should be a conservative extension of UML – standard UML should be retained as a subset. *UMLi makes no changes to the standard UML metamodel, although it does introduce some extensions. Any model constructed using the standard Argo/UML tool – which is driven from the UML metamodel – is also a valid model in ARGOi.*

Principle 2 The UML*i* proposal should introduce as few new models and constructs into the UML as possible. *UMLi introduces no truly new models into UML. User Interface Diagrams are a notational change to class diagrams, and activity diagrams have simply been extended using a macro notation.*

Principle 3 The UML*i* proposal should support the expectations of current UML modellers, whose experience with UML should be of benefit when using interface-specific extensions. *As UMLi supports UML as as subset, all existing facilities retain their role and semantics. When modelling user interfaces, UML models can use UMLi facilities as and when they choose, and can link the UMLi extensions with class diagrams in well-defined ways. Furthermore, task modelling is supported within the context of familiar activity diagrams.*

Principle 4 The UML*i* proposal should support the expectations of user interface modellers who have experience using existing interface modelling techniques. Such users should not feel that they are having to design interfaces with less supportive facilities than are provides by MB-UIDEs. *This principle is in tension with the previous one. UMLi seeks to provide intuitive facilities for task modelling that are familiar to user interface practitioners, but supports these within the notational context of activity diagrams, with which interface specialists will not generally be familiar.*

Principle 5 The UML*i* proposal should support the modelling of complete applications, so the links between user interface models and existing UML models should be well-defined and close. *The means by which UMLi models are integrated with other models in an unambiguous way is by including UMLi into the standard UML metamodel. Therein, UMLi task modelling is supported through a mapping to UML activity diagrams, and object flow states associated to interaction objects make explicit how an interface uses application data.*

8.3 Conclusions

The work in this thesis certainly does not provide a definitive answer to the question of how best to model user interfaces using UML. However, it does present a comprehensive approach to improving the design of user interfaces without needlessly complicating UML with new models that substantially overlap with existing models in their representational capabilities. For instance, it provides simple solutions to the questions about how to model UIs in UML raised during the development of the library case study using standard UML. Further, these solutions have proved to be effective for reducing complexity of UML-based models, which is one of the main motivations for providing specialised support for user interface modelling.

There are people in the HCI community that do not believe in the use of state-transition diagrams, such as activity diagrams, for modelling the behaviour of interactive systems. We can identify with their views that the development of a proper task concept and notation may be a viable alternative to activity diagrams. However, the study in this thesis provides a framework for comparing the modelling of interactive systems using task models and activity diagrams by describing many attributes associated with tasks in activity-based models. Thus, the study in this dissertation may help to support a proposal of a non-conservative extension of UML in the future.

A further possible criticism of UML*i* from people of the software engineering community is that the level of detail in the models is more like a visual programming of user interface software that the design of user interfaces. Again, we can identify with such a position, in this case that the level of detail in UML*i* models may greater than is desirable. Indeed, we believe that the UML*i* support for UI modelling can be further improved.

Finally, the next challenge for UML*i* may be outside the research environment. Indeed, UML designers are struggling to model interactive systems using UML, as can be observed in discussion lists of OOPL and UML users. Some designers are already using UMLi. For instance, 43 users have downloaded ARGOi and the UMLi web-page was visited 1,170 times between 18th October 2001 and 17th February 2002.

8.4 Future Work

Concerning the inherent problems of the MB-UIDE technologies presented in Section 2.4, UMLi is an answer to the problem of not having a common notation for user interface models. Some inherent problems of MB-UIDEs such as the post-editing problem also identified in Section 2.4 are not addressed by UMLi, which remains a challenging research problem. Indeed, a list of open research problems is presented as follows.

Developing web applications using UML*i*. The web architecture provides some challenges to the use of UMLi. For instance, it must be observed that every page of a web application can be used as an entry point to the application. A solution to this particular requirement of web applications when using UMLi may not be difficult to be explained. However, an in-depth study of the use of UMLi for developing web applications may be required to identify if the web has other special requirements that may not be addressed by UMLi.

Interpretation of model checking problems. Model checking can be used to identify problems in LOTOS specifications. However, it would be desirable if model checking problems could be described in terms of UML constructs when the LOTOS specifications are translations of UML models using the UMLi semantics. This is not a trivial problem, since the mappings between LOTOS and UMLi constructs are not one-to-one.

Implementing support for concrete UI presentation. UMLi relies on the use of design patterns to map UI diagrams into concrete presentation models, as described in Section 3.5. Thus, it would be desirable if the latest results in terms of UI design guidelines could be described within the UMLi framework.

Implementing additional features in ARGO*i*. New features can be implemented in ARGOi to increase its acceptability by designers of interactive systems. The generation of user interface code from the user interface aspects modelled in UML*i* models is the next natural candidate feature. Support for a collaborative design environment is another candidate feature that would have a significant impact for designers, especially for teams composed of application designers and user interface designers.

Extending UML*i* to non-form-based user interfaces. There are important categories of non-form-based user interfaces that probably cannot be modelled in UML*i*. However, it is unclear whether UML*i* would be appropriate to model these other categories of UIs. Thus, the identification of the problems that UML*i* may present when modelling non-form-based user interfaces would be the first step towards a version of UML*i* suitable for use with other kinds of interactive application.

Appendix A

Additional Semantics for UML*i*

A.1 Operation Specification

In addition to the attributes, the `BookCopy` class has seven Operations, as described in Figure 3.2. A LOTOS specification for the `renewLoan()` and `getCopyCode()` Operations is presented in this section. The `renewLoan()` Operation implements a functionality of the Library System, as described in Section 3.1. Rather than implementing Library System functionalities, the `getCopyCode()` Operation is responsible for encapsulating the `copyCode` Attribute.

An explanation of how the `renewLoan()` and `getCopyCode()` Operations can be specified in LOTOS indicates how Classifier's Operations can be specified in LOTOS. Each Operation is associated with a `CallAction`, which is composed of a pair of `Messages`. The Message and CallAction and Operation constructs are defined in terms of LOTOS operators as follows:

UCD 35 *A* Message *is the specification of an observable action of CLASS_CLS in the behaviour expression of CLASS_CLS or any of its subprocesses.*

UCD 36 *A* CallAction *is a pair of* Messages, *e.g.,* $< callinvoker , callresponse >$. *In the behaviour expression of the* Classifier *process of an* Object *invoking the* Operation, *callinvokers must come before callresponses. In the* Classifier *process of an* Object *where the* Operation *is invoked, the* CallAction Messages *are used as specified in UCD 37.*

The **accept ... in** construct of LOTOS is required to introduce the Operation's UCD. The **accept ... in** is used to assign the results of a process preceding an *enabling* (>>) construct to a set of LOTOS variables defined after the **accept** keyword. Thus, the Operation construct can be defined in terms of LOTOS operators as follows.

UCD 37 *An* Operation *is the specification of a subprocess in CLASS_CLS, e.g., OPERATION, defined as follows:*

```
process CLASS_CLS [callinvoker, callresponse, destroy_class]
          (c: Class) : exit :=
  ( (
    ...
    []
    ( callinvoker;
      OPERATION[callresponse](c) >>
      accept upd_c: Class in CLASS_CLS[callinvoker,
                   callresponse, destroy_class](upd_c) )
    []
    ... ) [> destroy_class; exit
  where
    process OPERATION[callresponse](c) :
              exit(Class) :=
      i ; callresponse; exit(any Class)
    endproc
endproc
```

According to UCD 36, a pair of *CallInvoker* and *CallResponse* observable actions is specified for each Operation. Further, the Operation process, called *operation process*, preserves the state of its Classifier, receiving and returning the state of their Classifier's process, as in UCD 37. For example, in Figure A.1, the RENEWLOAN process receives the `bc` object of type `BookCopy`, returning it on the **exit**(**any** `BookCopy`). The **any** operator of LOTOS specifies that any value in the domain of the specified type, e.g., `BookCopy`, can be returned. Moreover, the value of c declared in UCD 11 may be unaffected. For instance, if the Operation in the UML model is specified with the `isQuery` Attribute set to `TRUE` then the **any** `Class` in **exit** must be replaced by the parameter c of CLASS_CLS. An Operation's parameters are added to the Operation process's parameter list. Return values are added as a LOTOS event to the CallResponse action. In the case of the GETCOPYCODE process, the string returned by the Operation, e.g., `getccode(bc)`, is added to `getcc_res`, the GETCOPYCODE's CallResponse action.

```
process BOOKCOPY_CLS[getcc, getcc_res, renew, renew_res,
                destroy_bookcopy] (bc: BookCopy) : exit :=
  ( getcc; GETCOPYCODE[getcc_res](bc) >>
    accept upd_bc:BookCopy in
      BOOKCOPY_CLS[getcc, getcc_res, renew, renew_res,
                   destroy_bookcopy](upd_bc)
  []
    renew; RENEWLOAN[renew_res](bc) >>
    accept upd_bc:BookCopy in
      BOOKCOPY_CLS[getcc, getcc_res, renew, renew_res,
                   destroy_bookcopy](upd_bc) )
  [> destroy_bookcopy; exit
 where
  process GETCOPYCODE[getcc_res](bc:BookCopy) : exit(BookCopy) :=
    i; getcc_res!getccode(bc); exit(any BookCopy)
  endproc
  process RENEWLOAN[renew_res](bc:BookCopy) : exit(BookCopy) :=
    i; renew_res; exit(any BookCopy)
  endproc
endproc
```

Figure A.1: Specification of Operations in the BookCopy process.

Finally, behavioural expressions of Operation processes are specified by an *i* unobservable action of LOTOS prefixing a CallResponse action, prefixing an **exit** operation. The *i* action specifies that some action should happen during the execution of the method, but nothing can be said about this action. Broadly speaking, LOTOS can be used for the specification of implemented software systems. UML, however, is intended to be used during the design phase of the development process of a software system. Therefore, some *under-specifications* are expected in LOTOS specifications generated from UML models [19]. Most under-specifications are implicitly represented in the LOTOS specification. For example, Activities that are not refined into ActionStates, such as the connect Activity in Figure 5.3, are under-specifications. In the case of Operations, however, the *i* actions in their behaviour expressions are explicitly under-specifications. These *i* actions are used in LOTOS, in this case, to represent an internal event that may not be influenced by any process (non-determinism). In fact, it is assumed that the specification of methods is an implementation concern rather than a design concern. Nevertheless, LOTOS could be used to specify the implementation of methods.

The BOOKCOPY_CLS version with the encapsulated attributes does not specify attributes. In fact, this is a simplification in order to keep the translation of operations concise. However, a proper specification of the BookCopy class should preserve the attribute specification that may be hidden from other processes using the **hide** operator of LOTOS.

A.2 Association and ClassifierRole Specifications

Let $\mathcal{C}$ be the finite set of Classifiers of a UML design of $\mathcal{S}$, and $\mathcal{A} = \mathcal{C} \times \mathcal{C}$. Thus, an Association can be defined as follows.

Definition 1 *An Association (α) is a binary relationship between* Classifiers ($\alpha \in \mathcal{A}$).

An Association, as in Definition 1, is not a UCD. Indeed, in this paper, the Association's UCD is defined by the ClassifierRoles related to an Association. Thus, let $\mathcal{R} = \mathcal{A} \times \mathcal{C}$. From $\mathcal{R}$ and Definition 1 we can see that $\forall \alpha \in \mathcal{A} \Rightarrow \exists \rho_1, \rho_2 \in \mathcal{R} \bullet \rho_1 \neq \rho_2 \wedge \Pi_{\mathcal{A}}(\rho_1) = \Pi_{\mathcal{A}}(\rho_2) = \alpha$, where $\Pi_{\mathcal{A}}(\rho_z)$ is a projection of the value in the domain of $\mathcal{A}$ in ρ_z. Then a ClassifierRole can be defined as follows.

UCD 38 *A* ClassifierRole *(ρ) is a binary relationship between a* Classifier *and an* Association ($\rho \in \mathcal{R}$) *specified as a* CallAction, *where its CallResponse action returns an* Enumeration *of related instances of the associated* Classifier[1] *for the current instance. The* Enumeration *is generated by a GEN_ENUM process which precedes the CallResponse. For an associated Class2* Classifier, *the GEN_ENUM is specified as follows.*

```
process GEN_ENUM[] : exit(Enumeration_Class2) :=
  exit(any Enumeration_Class2)
endproc
```

Association's UCD can be specified from UCD 38. In fact, ClassifierRoles are synchronisations between processes that are equivalent to Associations mathematically defined as in Definition 1 [3]. Further, the UML specification says that there are two instances of AssociationEnd for each instance of Association. Moreover, the AssociationEnd has an attribute `aggregation` which can have the values `none`, `aggregate` or `composite`. Therefore, Table A.1 introduces a set of UCDs based on UCD 38 and the possible combination of types of AssociationEnds in an Association. There, α_1 is an Association between a $P1$ Classifier playing a ρ_1 ClassifierRole and a $P2$ Classifier playing a ρ_2 ClassifierRole. Furthermore, the execution of the `CREATE_CLASS_AS[]` instantiates Class, as described in Section 5.5.1.

[1] As described in Section 5.3, there is an enumeration type definition for every Classifier type definition translated from a UML model.

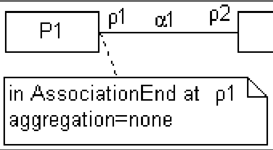
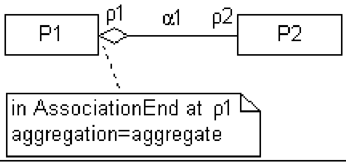
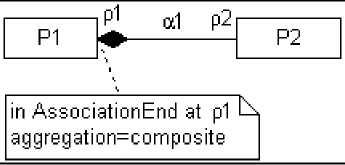
UCD	U	$\Phi(U)$
39		$(\exists\Phi(\rho_1) \vee \Phi(\rho_2))$ where $\Phi(\rho_1)$ and $\Phi(\rho_2)$ are defined as in UCD 38.
40		$(\exists\Phi(\rho_1) \vee \Phi(\rho_2))$ $\wedge$ (P1_CLS has parameter of P2 type)
41		$(\exists\Phi(\rho_1) \vee \Phi(\rho_2))$ $\wedge$ (P1_CLS has parameter of P2 type) $\wedge$ (execution of CREATE_P1_AS[] $\Rightarrow$ execution of CREATE_P2_AS[]) $\wedge$ (execution of destroy_p1 $\Rightarrow$ execution of destroy_p2)

Table A.1: Association mapping.

The BookCopy Class is associated to the Loan and Book Classes, as presented in Figure 3.2. Thus, the example in Figure A.2 presents the LOTOS specification of the onLoan ClassifierRole in BOOKCOPY_CLS. There, the Eloan returned by the onloan_res is an enumeration of type Enumeration_Loan. The invocation of the hasBookCopy ClassifierRole may be specified in the specification of the Methods of the BookCopy Operations later in the implementation phase of the Library system.

```
process BOOKCOPY_CLS[ setcc , ... , onloan , onloan_res , hasbookcopy ,
                hasbookcopy_res , destroy_bookcopy ] ( bc: BookCopy ) : exit :=
   ( (* previously defined attributes and operations *)
     []
     onloan ; ONLOAN[ onloan_res ]( bc) >>
     accept upd_bc:BookCopy in
       BOOKCOPY_CLS[ setcc , ... , onloan , onloan_res , hasbookcopy ,
                     hasbookcopy_res , destroy_bookcopy ]( upd_bc )
   [> destroy_bookcopy ; exit
 where
   (* previously defined operation processes *)
   process ONLOAN[ onloan_res ]( bc:BookCopy ) : exit (BookCopy) :=
     i ; GEN_ENUM >> accept Eloan: Enumeration_Loan in
        onloan_res ! Eloan ; exit (any BookCopy)
     where
       process GEN_ENUM[] : exit ( Enumeration_Loan ) :=
         exit (any Enumeration_Loan )
       endproc
   endproc
endproc
```

Figure A.2: Specification of Associations in the BookCopy process.

A.3 Signal, Sender and Receiver Specifications

The interaction between objects, as presented so far, is represented by Operations performed in a synchronous manner. For instance, methods performing CallActions need to wait for the conclusion of the triggered Operation. However, there may be actions that must to be performed in an asynchronous way. In the running case study, for instance, a Signal can be raised every time a `BookCopy` object is returned. In fact, `returnBook()` is the Sender Operation related to the `ReturnedCopy` Signal in Figure 3.2. Moreover, the invocation of the `returnBook()` Operation that raises the `ReturnedCopy` Signal is a SendAction for the Signal. Nevertheless, the Signal construct is responsible for providing such a facility modelled in Figure 3.2.

UCD 42 *A* Signal *is a* Message *specified immediately after the* **in** *keyword in the invocation of the* Signal*'s associated* Operation *process (UCD 37).*

UCD 43 *A* SendAction *is the specification of a* Signal *in the behavioural expression of an* ActionState *(UCD 27).*

Thus, UCD 42 can be used to implement the required asynchronous action presented above. For instance, `returnedcopy_sig` in Figure A.3 is raised every time the RETURNBOOK operation of BOOKCOPY_CLS is performed.

```
process BOOKCOPY_CLS[..., returnbook, returnbook_res,
          returnedcopy_sig, destroy_bookcopy] (bc: BookCopy) : exit :=
  ( (* prev. defined attributes, operations and associations *)
    []
    ( returnbook?new_bc: Bookcopy; RETURNBOOK[returnbook_res](bc, new_bc) >>
        accept upd_bc:BookCopy in returnedcopy_sig!upd_bc;
        BOOKCOPY_CLS[..., returnbook, returnbook_res,
                     returnedcopy_sig, destroy_bookcopy](upd_bc) ) )
  [> destroy_bookcopy; exit
 where
  (* other operation and association processes *)
endproc
```

Figure A.3: Specification of Signals in the `BookCopy` process.

The use of Signals has more impact on the structure of Classes acting as Receivers than in Operations acting as Senders. Instances of the `Reservation` Class must receive the `returnedcopy_sig` Signal raised by BOOKCOPY_CLS in an asynchronous way. In Figure A.4, the *interleave* operator (|||) specifies that the

```
process RESERVATION_CLS[..., returnedcopy_sig, destroy_reservation]
          (res:Reservation) : exit :=
  (
    (* behaviour expression for attributes,
          operations and associations *)
  )
  |||
  ( returnedcopy_sig?bc:BookCopy;
    notify!bc; notify_res;
    RESERVATION_CLS[..., returnedcopy_sig, destroy_reservation](res))
  [> destroy_reservation; exit
 where
    (* definitions of operation and association processes *)
endproc
```

Figure A.4: Specification of the `Reservation` process as a Receiver of the `ReturnBook` Signal.

`returnbook_sig` is not synchronised with any other action that may be performed within RESERVATION_CLS.

The RESERVATION_CLS is acting as a handler of the `returnedcopy_sig` since the Signal is invoking a `Notify` Operation also defined in the RESERVATION_CLS. However, it may be the case that RESERVATION_CLS could rely on other Classifiers that could act as Handlers as well.

A.4 Generalisation Specification

As presented so far, a Classifier is defined by its type and process. The `BookCopy` classifier is defined by the BOOKCOPY type and the BOOKCOPY_CLS process. Thus, a Classifier can be generalised as long it can inherit the type and process actions provided by its superclass.

In terms of type there is no difficulty in implementing this. For instance, supposing that the `Person` classifier is already specified in LOTOS, the type of the `Borrower` classifier can be specified in the way presented in Figure A.5. For instance, the `getname(castPerson(aBorrower))` type operation can return the `name` Attribute defined in the `Person` type specification.

Moreover, Classifier features, viz. Attributes, Operations, Associations and Signals are specified as observable actions. Therefore, a full synchronisation between PERSON_CLS and BORROWER_CLS (e.g., *PERSON_CLS* || *BORROWER_CLS*) makes `Borrower` inherit the features of `Person`. In this case, `Borrower` must only to implement the role actions of its association with the `Loan`

```
type BORROWER is PERSON with
   sorts Borrower
   opns mk_borrower:             -> Borrower
        isPerson: Person         -> Borrower
        castPerson: Borrower -> Person
endtype
```

Figure A.5: Specification of `Borrower` as a specialisation of `Person`.

class, which is what makes it different from `Person`.

A.5 A Generic Specification for Classifiers

The presented UCDs have demonstrated how to generate a LOTOS specification for a Classifier and its Attributes, Operations, Associations and Signals. Thus, assume that a Classifier can be represented by a tuple $\mathcal{T} =< \Gamma, \Omega, \Sigma_s, \Sigma_r >$, where:

- Γ is a finite set of Attributes;
- Ω is a finite set of Operations;
- Σ_s is a finite set of Signals where the Classifier is a Sender;
- Σ_r is a finite set of Signals where the Classifier is a Receiver

For each tuple $\mathcal{T}$ there exists one LOTOS specification of a CLASS type and a CLASS_CLS process. For instance, suppose that for a specific Classifier, the cardinality of Γ is w ($\#\Gamma = w$), which means, $\Gamma = \{ a_1, a_2, ..., a_w \}$. Further, suppose that each attribute of Γ has a corresponding type a_1type, a_2type, ..., a_wtype. Figure A.6 presents the generic CLASS type.

Moreover, suppose that $\#\Omega = x$ ($\Omega = \{ o_1, o_2, ..., o_x \}$), $\#\Sigma_s = y$ ($\Sigma_s = \{ s_1, s_2, ..., s_y \}$); and $\#\Sigma_r = z$ ($\Sigma_r = \{ t_1, t_2, ..., t_z \}$). Then, a set of parameters and their types is defined which is provided by each Signal of Σ_r, $\{t_ip_1$: t_ip_1type, t_ip_2: t_ip_2type, ... $\}$. So, for each Operation, e.g., o_i, is defined:

- a *CallInvoke* action, o_i, and a *CallResponse* action, o_ires;
- a set of parameters along with their respective types, $\{o_ip_1$: o_ip_1type, o_ip_2: o_ip_2type, ... $\}$; and

```
type CLASS is   a1type , ... ,  awtype
  sorts Class
  opns  mk_Class:                              -> Class
        mk_Class2:  a1type , ... ,  awtype -> Class
        set a1 :  Class ,  a1type              -> Class
        get a1 :  Class                         -> a1type
        set a2 :  Class ,  a2type              -> Class
        get a2 :  Class                         -> a2type
        ...
        set aw :  Class ,  awtype               -> Class
        get aw :  Class                          -> awtype
  (* eqns specifications *)
endtype
```

Figure A.6: A type for a generic Classifier.

- a set of results along with their respective types, $\{o_ir_1$: o_ir_1type, o_ir_2: o_ir_2type, ... $\}$.

Finally, to simplify the representation of CLASS_CLS, let G be the ordered set of gates: $\{a_1inp$, a_1out, a_2inp, a_2out, ..., a_winp, a_wout, o_1, o_1res, o_2, o_2res, ..., o_x, o_xres, s_1, s_2, ..., s_q, ..., s_y, t_1, t_2, ..., t_z, destroy_class$\}$. Thus, Figure A.7 presents the generic CLASS_CLS process.

A.6 The OrderIndependentState Specification

In the context of the `GENERIC_ACT` Activity used to define the OptionalState in Section 5.5.6, an OrderIndependentState defined within the `GENERIC_ACT` and also having the range `SUB1_ACT` $\cdots$ `SUBn_ACT` of processes representing its selectable Activities is defined as follows.

UCD 44 *An* OrderIndenpendentState *is defined as the* `ORDERINDEPENDENT_ACT` *process specified as follows.*

```
process GENERIC_ACT[abort] : exit :=
  ... >> ORDERINDEPENDENT_ACT[..., abort_oi] >> ...
where
  process ORDERINDEPENDENT_ACT[..., abort_oi] : exit :=
    ( resumesub1; aoutsub1; suspendsub1;
      SUB1_ACT[abort_oi]                         |||
      (* an action state behaviour expression *)  |||
      ...                                        |||
      resumesubn; aoutsubn; suspendsubn;
      SUBn_ACT[aout_oi] )
```

```
process CLASS_CLS [G](c: Class) : exit :=
  (
    (   a1inp?p1:a1type;    CLASS_CLS[G](seta1(c,p1))
    []  a1out!a1;            CLASS_CLS[G](c)
        ...
    []  awinp?pw:awtype;    CLASS_CLS[G](setaw(c,pw))
    []  awout!aw;            CLASS_CLS[G](c)
    []  o1?o1p1:o1p1type?o1p2:o1p2type...;  OPER1[o1res](c, o1p1, o1p2, ...) >>
            accept upd_c:Class in CLASS_CLS[G](upd_c)
        ...
    []  ox?oxp1:oxp1type?oxp2:oxp2type...;  OPERx[oxres](c, oxp1, oxp2, ...) >>
            accept upd_c:Class in CLASS_CLS[G](upd_c)
    ||| (t1?t1p1:t1p1type?t1p2:t1p2type?...; ...;  CLASS_CLS[G](...))
    ...
    ||| (tz?tzp1:tzp1type?tzp2:tzp2type?...; ...;  CLASS_CLS[G](...))
  ) [> destroy_class; exit
  where
    process OPER1[o1res]
          (c: Class, o1p1: o1p1type, o1p2: o1p2type, ...) : exit(Class) :=
      i; o1res!o1r1!o1r2!...;  exit(any Class)
    endproc
    ...
    process OPERx[oxres]
          (c: Class, oxp1: oxp1type, oxp2: oxp2type, ...) : exit(Class) :=
      i; oxres!oxr1!oxr2!...;  exit(any Class)
    endproc
endproc
```

Figure A.7: A process for a generic Classifier.

```
    [> abort_oi; exit
  where
    process SUB1_ACT[..., abort_oi]
      (* SUB1_ACT specification *)
    endproc
    process SUB3_ACT[..., abort_oi]
      (* SUB1_ACT specification *)
    endproc
    ...
    process SUBn_ACT[..., abort_oi]
      (* SUBn_ACT specification *)
    endproc
  endproc
endproc
```

The `resumesub1`, `suspendsub1`, `aoutsub1` $\cdots$ `resumesubn`, `suspendsubn` *and* `aoutsubn` *are gates of either* ActionInvokers *associated to subactivities by* ≪activates≫ ObjectFlowStates *(UCD 45) or* PrimitiveInteractionClasses *associated to subactivities that are* ActionStates *by* ≪interacts≫ ObjectFlowStates *(UCD 33).*

UCD 44 shows that the OrderIndependentState is a special case of the Fork and Join PseudoStates (UCD 8). In fact, the kind of similarity between the OrderIndependentState and the Fork and Join PseudoStates can be compared to the kind of similarity between the OptionalState and the Branch PseudoState. For instance, the behaviour expression of the OrderIndependentState, in contrast with the Fork and Join, depends on the existence of ActionInvokers and PrimitiveActionStates to provide the gates used in its composition. Moreover, the same comments concerning the specification of behaviour expressions of subactivities that are ActionStates of OptionalStates discussed in Section 5.5.6 are valid for OrderIndependentState.

A.7 The ≪*activates*≫ Stereotype Specification

The ≪*activates*≫ stereotype provides a mechanism that allows users to trigger Activities that can also be ActionStates.

UCD 45 *The* ≪activates≫ *stereotype is the prefixing of the* `aout` *gate of the associated* ClassifierInState *of type* ActionInvoker *to the* `ASSOC_ACT` *process of the associated* Activity *as follows.*

... >> <u>resumeactv</u>; <u>aoutactv</u>; <u>suspendactv</u>; <u>ASSOC</u>_ACT [ ] >> ...

In the UCD 45 it can be observed that the ≪*activates*≫ stereotype is also responsible for enabling and disabling the ActionInvoker in order to make it ready for interaction just before the performance of the associated Activity.

A.8 The RepeatableState Specification

An Activity having just one outgoing Transition to itself would produce a control-flow live-lock if it is unable to be confirmed. A RepeatableState, however, allows the specification of Activities having a self-addressed Transition that does not create a control-flow live-lock since it can be confirmed by a ≪*confirms*≫ interaction object flow (UCD 29). Thus, a RepeatableState is defined as follows.

UCD 46 *A* RepeatableState *is a* `REPEATABLE_ACT` *process as define in UCD 29 with a* `SUB1_ACT` *subprocess defined as follows.*

process GENERIC_ACT [..., abort] : **exit** :=
  ... >> <u>REPEATABLE</u>_ACT [..., <u>aoutcn</u>, <u>resumecn</u>, <u>suspendcn</u>] >> ...
where

```
  RETEATABLE_ACT [ . . . , aoutcn, resumecn, suspendcn] : exit :=
    resumecn >> (SUB1_ACT [ . . . , aoutcn] |||
     aoutcn; suspendcn;
      (* suspend associated InteractionClasses *) ; exit )
  where
    process SUB1_ACT [ . . . , aoutcn]
        (* SUB1_ACT specification *) >>
        REPEATABLE_ACT[ aoutcn, resumecn, suspendcn]
    endproc
  endproc
endproc
```

The SUB1_ACT *process must represent either an* Activity *triggered by an* ≪activates≫ ObjectFlowState *(UCD 45) or an* ActionState *with an* ≪interacts≫ ObjectFlowState *(UCD 33).*

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Nabeel Al-Shamma, Robert Ayers, Richard Cohn, Jon Ferraiolo, Martin Newell, Roger K. de Bry, Kevin McCluskey, and Jerry Evans. Precision Graphics Markup Language (PGML). World Wide Web Consortium (W3C) Note 10-April, 1998.

[3] Alessandro Artale, Enrico Franconi, Nicola Guarino, and Luca Pazzi. Part-Whole Relations in Object-Centered Systems: An Overview. *Data & Knowledge Engineering*, 20(3):347–383, 1996.

[4] Genera AS. Systemator. http://www.genera.no.

[5] Helmut Balzert. From OOA to GUI – The JANUS-System. In *Proceedings of INTERACT'95*, pages 319–324, London, UK, June 1995. Chapman & Hall.

[6] Helmut Balzert, Frank Hofmann, Volker Kruschinski, and Christoph 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.

[7] Peter J. Barclay, Tony Griffiths, Jo McKirdy, Norman W. Paton, Richard Cooper, and Jessie B. 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.

[8] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Software Engineering*, 22(10):751–761, October 1996.

[9] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How Reuse Influences Productivity in Object-Oriented Systems. *Communications of ACM*, 39(10):104–116, 1996.

[10] Bernhard Bauer. Generating User Interfaces from Formal Specifications of the Application. In *Computer-Aided Design of User Interfaces*, pages 141–157, Namur, Belgium, 1996. Namur University Press.

[11] James M. Bieman and Byung-Kyoo Kang. Measuring Design-level Cohesion. *IEEE Trans. Software Engineering*, 24(2):111–124, February 1998.

[12] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, I. Provot, B. Sacre, and Jean 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.

[13] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, I. Provot, and Jean Vanderdonckt. A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype. In *Proceedings of DSV-IS'94*, pages 25–39, Bocca di Magra, June 1994.

[14] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, and Jean Vanderdonckt. Computer-Aided Window Identification in TRIDENT. In *Proceedings of INTERACT'95*, pages 331–336, London, UK, 1995. Chapman & Hall.

[15] François Bodart and Jean Vanderdonckt. Widget Standardisation Through Abstract Interaction Objects. In *Advances in Applied Ergonomics*, pages 300–305, Istanbul - West Lafayette, May 1996. USA Publishing.

[16] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Network ISDN Systems*, 14(1), 1987.

[17] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, second edition, 1994.

[18] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[19] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, and Bernhard Rumpe. Towards a Formalization of the Unified Modelling Language. In *Proceedings of ECOOP'97*, LNCS, pages 344–366, Jyväskylä, Finland, June 1997. Springer-verlag.

[20] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Defining and Validating Measures for Object-Based High-Level Design. *IEEE Trans. Software Engineering*, 25(5):722–743, September/October 1999.

[21] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the Relationship between Design Measures and Software Quality in Object-Oriented Designs. *Journal of Systems and Software*, 51(3):245–273, 2000.

[22] T. Browne, D. Dávila, S. Rugaber, and K. Stirewalt. *Formal Methods in Human-Computer Interaction*, chapter Using Declarative Descriptions to Model User Interfaces with MASTERMIND. Springer-Verlag, 1997.

[23] R. G. G. Cattell, Douglas K. Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Harry Strickland, and Drew Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Francisco, CA, 1997.

[24] Shyam R. Chidamber and Chris F. Kemerer. A Metric Suite for Object Oriented Design. *IEEE Trans. Software Engineering*, 20(6):476–493, 1994.

[25] Robert G. Clark and Ana M. D. Moreira. Use of E-LOTOS in Adding Formality to UML. *Journal of Universal Computer Science*, 6(11):1071–1087, 2000.

[26] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice-Hall, 1991.

[27] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice-Hall, 1991.

[28] Dava Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, Redwood City, CA, 1995.

[29] Joëlle Coutaz. PAC, an Object Oriented Model for Dialog Design. In *Proceedings of INTERACT'87*, pages 431–436. North-Holland, 1987.

[30] Joëlle Coutaz and Richard N. Taylor. Introduction to the Workshop on Software Engineering and Human-Computer Interaction: Joint Research Issues. In *Proceedings of the Software Engineering and Human-Computer Interaction'94*, volume 896 of *Lecture Notes In Computer Science*, pages 1–3, Berlin, May 1995. Springer-Verlag.

[31] Bill Curtis and Bill Hefley. A WIMP No More – The Maturing of User Interface Engineering. *ACM Interactions*, 1(1):22–34, 1994.

[32] Desmond D'Souza and Alan Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison Wesley, Reading, MA, 1998.

[33] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monograph Series*. Springer-Verlag, 1985.

[34] Thomas Elwert and Egbert Schlungbaum. Modelling and Generation of Graphical User Interfaces in the TADEUS Approach. In *Designing, Specification and Verification of Interactive Systems*, pages 193–208, Vienna, 1995. Springer.

[35] Douglas C. Engelbart and William K. English. A Research Center for Augmenting Human Intellect. In *Proceedings of 1968 Fall Joint Computer Conference*, pages 395–410, San Francisco, CA, December 1968. AFIPS Press.

[36] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In *Proceedings of the UML'98*, volume 1618 of *LNCS*, pages 336–348, Mulhouse, France, June 1998. Springer-Verlag.

[37] Andy Evans and Stuart Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. In *Proceedings of the UML'99*, volume 1723 of *LNCS*, pages 140–155, Fort Collins, CO, October 1999. Springer-Verlag.

[38] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thonson, London, UK, second edition, 1996.

[39] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP - A Protocol Validation and Verification Toolbox. In *Proceedings of CAV'96*, volume 1102 of *LNCS*, pages 437–440, New Brunswick, NJ, July 1996. Springer.

[40] James D. Foley. History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based Systems for User Interface Design and Implementation. In *Proceedings of DSV-IS'94*, pages 3–14, Vienna, 1995. Springer-Verlag.

[41] James D. Foley, Won Chul Kim, Srdjan Kovacevic, and Kevin Murray. UIDE – An Intelligent User Interface Design Environment. In *Intelligent User Interfaces*, pages 339–384. Addison-Wesley, ACM Press, 1991.

[42] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[43] Chris Gane and Trish Sarson. *Structured System Analysis: Tools & Techniques*. Prentice-Hall, Englewood Cliff, NJ, 1977.

[44] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[45] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[46] Mark Green. A Survey of Three Dialogues Models. *ACM Transaction on Graphics*, 5(3):244–275, July 1986.

[47] Tony Griffiths, Peter J. Barclay, Jo McKirdy, Norman W. Paton, Philip D. Gray, Jessie B. Kennedy, Richard Cooper, Carole A. Goble, Adrian West, and Michael 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 Computer Society.

[48] Tony Griffiths, Peter J. Barclay, Norman W. Paton, Jo McKirdy, Jessie B. Kennedy, Philip D. Gray, Richard Cooper, Carole A. Goble, and Paulo Pinheiro da Silva. Teallach: A Model-Based User Interface Development Environment for Object Databases. *Interacting with Computers*, 14(1):31–68, December 2001.

[49] Tony Griffiths, Jo McKirdy, G. Forrester, Norman W. Paton, Jessie B. Kennedy, Peter J. Barclay, Richard Cooper, Carole A. Goble, and Philip D. Gray. Exploiting Model-Based Techniques for User Interfaces to Database. In *Proceedings of Visual Database Systems (VDB) 4*, pages 21–46, Italy, May 1998. Chapman & Hall.

[50] Anthony Hall. Taking Z Seriously. In *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 89–91, Reading, UK, April 1997. Springer.

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

[52] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Software Engineering and Methodology*, 5(4):293–333, October 1996.

[53] David Harel, Amir Pnueli, Jeanette Schmidt, and R. Sherman. On the Formal Semantics of Statecharts (Extended Abstract). In *Proceedings of the LICS'87*, pages 54–64, Ithaca, NY, June 1987. IEEE Computer Society.

[54] H. Rex Hartson, Antonio C. Siochi, and Deborah Hix. The UAN: A User-Oriented Representation for Direct Manipulation Interface Design. *ACM Trans. Information Systems*, 8(3):181–203, July 1990.

[55] Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner. Design Alternatives for User Interface Management Systems Based on Experience with COUSIN. In *Proceedings of SIGCHI'85*, pages 169–175. Addison-Wesley, April 1985.

[56] Deborah Hix and H. Rex Hartson. *Developing User Interfaces*. Wiley, 1993.

[57] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[58] Allen I. Holub. Building User Interfaces for Object-Oriented Systems. *JavaWorld*, July-November 1999, January 2000, March 2000.

[59] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Engineering*, 23(5):279–295, 1997.

[60] IBM Corporation. *Object-Oriented Interface Design: IBM Common User Access Guidelines*. QUE Corp., 1992.

[61] Daniel H. H. Ingalls. The Smalltalk-76 Programming System: Design and Implementation. In *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, pages 9–15, Tucson, AZ, 1978. ACM Press.

[62] ISO/IEC. *Enhanced LOTOS*, 2000. ISO/IEC 15437.

[63] ISO/IEC-JTC1/SC21/WG1/FDT/C. *LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, February 1989. IS 8807.

[64] Ari Jaaksi, Juha-Markus Aalto, Ari Aalto, and Kimmo Vättö. *Tried & True Object Development: Practical Approaches with UML*. Cambrigdge University Press, Cambridge, UK, 1999.

[65] Robert J. K. Jacob. A Specification Language for Direct Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

[66] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, Reading, MA, 1992.

[67] Ivar Jacobson, Martin Griss, and Patrick Jonsson. *Software Reuse: Architecture, Process and Organization for Business Seccess*. ACM Press, New York, NY, 1997.

[68] Christian Janssen, Anette Weisbecker, and Jürgen Ziegler. Generating User Interfaces from Data Models and Dialogue Net Specifications. In *Proceedings of InterCHI'93*, pages 418–423, New York, NY, 1993. ACM Press.

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

[70] Peter Johnson, Hilary Johnson, and Stephanie Wilson. Rapid Prototyping of User Interfaces Driven by Task Models. In *Scenario-Based Design*, pages 209–246, London, UK, 1995. John Wiley.

[71] Barry H. Kantowitz and Robert D. Sorkin. *Human Factors: Understanding People-System Relationships.* John Wiley and Sons, New York, NY, 1983.

[72] Won Chul Kim and James D. Foley. DON: User Interface Presentation Design Assistant. In *Proceedings of UIST'90*, pages 10–20. ACM Press, October 1990.

[73] B. Kirwan and L. Ainsworth. *A Guide to Task Analysis.* Taylor & Francis, London, UK, 1992.

[74] Cris Kobryn. UML 2001: A Standardization Odyssey. *Communications of the ACM*, 42(10):29–37, October 1999.

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

[76] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[77] Diogo Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[78] Wei Li and Sallie M. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[79] Johan Lilius and Ivan Paltor. Formalising UML State Machines for Model Checking. In *Proceedings of UML'99*, volume 1723 of *LNCS*, pages 430–445, Fort Collins, CO, October 1999. Springer.

[80] Frank Lonczewski and Siegfried Schreiber. The FUSE-System: an Integrated User Interface Desgin Environment. In *Computer-Aided Design of User Interfaces*, pages 37–56, Namur, Belgium, 1996. Namur University Press.

[81] Ping Luo, Pedro A. Szekely, and Robert Neches. Management of interface design in HUMANOID. In *Proceedings of InterCHI'93*, pages 107–114, April 1993.

[82] Panos Markopoulos. *A Compositional Model for the Formal Specification of User Interface Software*. PhD thesis, Queen Mary and Westfield College, University of London, March 1997.

[83] Panos Markopoulos, Peter Johnson, and Jon Rowson. Formal architectural abstracttions for interactive Systems. *International Journal of Human Computer Studies*, 49(5):679–715, 1998.

[84] Panos Markopoulos and Peter Marijnissen. UML as a representation for Interaction Designs. In *Proceedings of OZCHI 2000*, pages 240–249, 2000.

[85] Panos Markopoulos, J. Pycock, Stephanie Wilson, and Peter Johnson. Adept – A task based design environment. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, pages 587–596. IEEE Computer Society, 1992.

[86] Christian 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.

[87] Thomas J. McCabe and Charles W. Butler. Design Complexity Measurement and Testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[88] Stephen J. Mellor, Stephen R. Tockey, Rodolphe Arthaud, and Philippe Leblanc. An Action Language for UML: Proposal for a Precise Execution Semantics. In *Proceedings of the UML'98*, volume 1618 of *LNCS*, pages 307–318, Mulhouse, France, June 1998. Springer-Verlag.

[89] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.

[90] Microsoft Corporation. *The Windows Interface: An Application Design Guide*. Microsoft Press, Redmond, WA, 1992.

[91] Robin Milner. *Communication and Concurrency.* Prentice Hall, New York, NY, 1989.

[92] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, II. *Information and Computation*, 100(1):41–77, 1992.

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

[94] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad T. Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, 1990.

[95] Brad A. Myers and Mary Beth Rosson. Survey on User Interface Programming. In *Proceedings of SIGCHI'92*, pages 192–202, 1992.

[96] Novosoft. NSUML – Novosoft Metadata Framework and UML Library. http://nsuml.sourceforge.net.

[97] Nuno J. Nunes and João Falcão e Cunha. Towards a UML profile for interaction design: the Wisdom approach. In *Proceeding of UML2000*, volume 1939 of *LNCS*, pages 101–116, York, UK, 2000. Springer.

[98] Nuno J. Nunes and João Falcão e Cunha. Wisdom – A UML Based Architecture for Interactive Systems. In *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 191–205, Limerick, Ireland, June 2000. Springer-Verlag.

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

[100] Dan R. Olsen. A Programming Language Basis for User Interface Management. In *Proceedings of SIGCHI'89*, pages 171–176, May 1989.

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

[102] Fabio Paternò. Integrating Model Checking and HCI Tools to Help Designers Verify User Interface Properties. In *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 135–150, Limerick, Ireland, June 2000. Springer-Verlag.

[103] Fabio Paternò. Towards a UML for Interactive Systems. In *Proceedings of EHCI2001*, LNCS, pages 7–18, Toronto, Canada, May 2001. Springer.

[104] Fabio Paternò and G. Faconti. On the Use of LOTOS to Describe Graphical Interaction. In *People and Computers VII: Proceedings of the HCI'92*, pages 155–173, York, UK, September 1992. Cambridge University Press.

[105] Dorina C. Petriu and Yimei Sun. Consistent Behaviour Representation in Activity and Sequence Diagrams. In *Proceedings of UML2000*, volume 1939 of *LNCS*, pages 369–382, York, UK, October 2000. Springer.

[106] Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 207–226, Limerick, Ireland, June 2000. Springer-Verlag.

[107] Paulo Pinheiro da Silva, Tony Griffiths, and Norman W. Paton. Generating User Interface Code in a Model Based User Interface Development Environment. In *Proceedings of AVI2000*, pages 155–160, Palermo, Italy, May 2000. ACM Press.

[108] Paulo Pinheiro da Silva and Norman W. 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.

[109] Paulo Pinheiro da Silva and Norman W. Paton. Object Modelling of User Interfaces in UMLi. Submitted for publication, 2001.

[110] Paulo Pinheiro da Silva and Norman W. Paton. User Interface Modelling with UML. In *Information Modelling and Knowledge Bases XII*, pages 203–217, Amsterdam, The Netherlands, 2001. IOS Press.

[111] Angel R. Puerta. The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development. In *Computer-Aided Design of User Interfaces*, pages 19–36, Namur, Belgium, 1996. Namur University Press.

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

[113] Angel R. Puerta and Jacob Eisenstein. Interactively Mapping Task Models to Interfaces in MOBI-D. In *Design, Specification and Verification of Interactive Systems*, pages 261–273, Abingdon, UK, June 1998.

[114] Angel R. Puerta and Jacob Eisenstein. Towards a General Computational Framework fo Model-Based Interface Development Systems. In *Proceedings of IUI'99*, pages 171–178, Los Angeles, CA, January 1999.

[115] Angel R. Puerta and David Maulsby. Management of Interface Design Knowledge with MODI-D. In *Proceedings of IUI'97*, pages 249–252, Orlando, FL, January 1997.

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

[117] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.

[118] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. ARGO: A Design Environment for Evolving Software Architectures. In *Proceedings of ICSE'97*, pages 600–601, Boston, MA, May 1997. ACM Press.

[119] Jason E. Robbins and David F. Redmiles. Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Information & Software Technology*, 42(2):79–89, 2000.

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

[121] James E. Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[122] Egbert Schlungbaum. Model-Based User Interface Software Tools - Current State of Declarative Models. Technical Report 96–30, Graphics, Visualization and Usability Center, Georgia Institute of Technology, 1996.

[123] Siegfried Schreiber. Specification and Generation od User Interfaces with the BOSS-System. In *Proceedings of EWHCI'94*, volume 876 of *Lecture Notes in Computer Sciences*, pages 107–120, Berlin, 1994. Springer-Verlag.

[124] Siegfried Schreiber. The BOSS System: Coupling Visual Programming with Model Based Interface Design. In *Proceedings of DSV-IS'94*, Focus on Computer Graphics, pages 161–179, Berlin, 1995. Springer-Verlag.

[125] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, second edition, 1992.

[126] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, third edition, 1997.

[127] Keng Siau and Q. Cao. Unified Modeling Language (UML) – A Complexity Analysis. *Journal of Database Management*, 12(1):26–34, 2001.

[128] Jon Siegel. *CORBA: Fundamentals and Programming*. John Wiley, New York, NY, 1996.

[129] Gurminder Singh and Mark Green. A high-level user interface management system. In *Proceedings of SIGCHI'89*, pages 133–138, May 1989.

[130] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series of Computer Science. Prentice-Hall International, Englewood Cliffs, NJ, March 1992.

[131] Kurt Stirewalt. *Automatic Generation of Interactive Systems from Declarative Models*. PhD thesis, Georgia Institute of Technology, December 1997.

[132] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition, 1997.

[133] Pedro A. Szekely. Template-Based Mapping of Application Data to Interactive Displays. In *Proceedings of UIST'90*, pages 1–9. ACM Press, October 1990.

[134] Pedro A. Szekely, Ping Luo, and Robert Neches. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In *Proceedings of SIGCHI'92*, pages 507–515, May 1992.

[135] Pedro A. Szekely, Piyawadee Noi Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, and Ewald 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.

[136] R. Chung-Man Tam, David Maulsby, and Angel R. 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.

[137] Jean-Claude Tarby and Marie-France Barthet. The DIANE+ Method. In *Computer-Aided Design of User Interfaces*, pages 95–119, Namur, Belgium, 1996. Namur University Press.

[138] The precise UML group. http://www.cs.york.ac.uk/puml/.

[139] Tigres.org. GEF – Java Library for Connected Graph Editors. http://gef.tigres.org.

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

[141] Jean Vanderdonckt. *Conception assistée de la présentation d'une interface homme-machine ergonomique pour une application de gestion hautement interactive*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, July 1997.

[142] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and Integrating the Dynamic Model within OMT. In *Proceeding of ICSE'97*, pages 45–55, Boston MA, May 1997. ACM Press.

[143] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley, 1999.

[144] Charles Wiecha, William Bennett, Stephen J. Boies, John D. Gould, and Sharon Greene. ITS: A Tool for Rapidly Developing Interactive Applications. *ACM Trans. Information Systems*, 8(3):204–236, July 1990.

[145] Charles Wiecha and Stephen J. Boies. Generating user interfaces: principles and use of ITS style rules. In *Proceedings of UIST'90*, pages 21–30. ACM Press, October 1990.

[146] Stephanie Wilson and Peter Johnson. Bridging the Generation Gap: From Work Tasks to User Interface Designs. In *Computer-Aided Design of User Interfaces*, pages 77–94, Namur, Belgium, 1996. Namur University Press.

[147] Martin Wirsing. Algebraic Specification. In *Handbook of Theoretical Computer Science*, pages 676–788. North Holland, 1990.

[148] Sherif M. Yacoub, Hany H. Ammar, and T. Robinson. A Matrix-Based Approach to Measure Coupling in Object-Oriented Designs. *JOOP*, pages 8–19, November 2000.

[149] Douglas A. Young. *Object-Oriented Programming with C++ and OSF/MOTIF*. Prentice-Hall, Englewood Cliffs, NJ, 1992.