Faculty of Engineering of the University of Porto

Informatics and Computing Engineering Master Program

Universidade do Porto
FEUP Faculdade de Engenharia

Development of a Visual GUI Modelling Front-End

MIEIC 2008 – Master Thesis

by

Rodrigo Manuel Lopes de Matos Moreira

Supervised by

Assistant Professor Ana Cristina Ramada Paiva Pimenta

April 2008, Porto – Portugal

Abstract

Nowadays, most of software systems usually feature graphical user interfaces (GUIs). They are a means to provide functionalities to the users whenever they interact with them. GUIs' appearance, correctness and usability qualities contribute to the users' decision of using such software systems. However, GUI testing is difficult, extremely heavy, and with a limited set of both tools and techniques to assist in the testing process.

Methods and test practices based on formal specifications, or formal models, might help to systematise and automate GUI testing to higher levels, due to their potential to generate test cases automatically. Nevertheless, existing GUI testing approaches feature some limitations (problems) that are seen as obstacles in the promotion to their adoption in industrial environments: the time and effort required for building GUI formal models; modellers and testers are reluctant to write formal textual specifications/models; and there is the need to define coverage criteria best adapted for GUI testing, such as, coverage of navigation maps (the behaviour related to the possible paths an user can walk through in order to open/close the several windows of a GUI application).

This dissertation addresses some of the above problems. In order to overcome such problems, this research work started with an analysis on the state of the art towards GUI modelling. This task became highly profitable since it guided to innovative ideas that led to the design of a new visual notation for GUI modelling. Such visual notation aims to hide, as much as possible, the formal details inherent to models typically used in model-based testing (MBT) approaches and promote the use of MBT in industrial environments providing a visual front-end for modelling which is more attractive to testers/modellers than textual notation.

A set of rules was defined in order to automatically translate visual GUI models to a textual object-oriented model-based formal specification language (Spec#). GUI models developed with the visual notation are then translated automatically to Spec# that can be then completed manually with additional behaviour which was not included in the visual model for being developed in a higher level of abstraction. This translation is performed by means of a software tool (developed during the course of this work), that aims to maintain consistency between the visual model and the textual formal (Spec#) model. Once Spec# model is completed, it can be used as input to Spec Explorer (model-based testing tool) which generates test cases and executes those tests automatically.

The approach proposed in this research work is illustrated and validated by a case study performed on the Notepad application that ships with the Microsoft Windows operating system. Overall, the visual notation together with the automatic translations mechanism provided savings in the time spent with the modelling activity around 40% when compared with the GUI modelling directly in Spec#.

A scientific paper was submitted to an international conference as a way to reveal the approach proposed, the outcomes produced and results achieved by this research work.

Acknowledgements

I would like to express my gratitude to my Supervisor for all the assistance and support I've received for this dissertation. I would like to thank for all the guidance, invaluable help, intellectual rigor and exchange of ideas. The encouragement and guidance of Assistant Professor Ana Cristina Ramada Paiva Pimenta proved invaluable throughout the process.

I express a deep appreciation to my parents who provided me items of greatest worth: responsibility, respectability and integrity. Thank you for standing by me through the many trials and decisions of my educational career. I express a deep appreciation for their efforts to help me get where I am today.

Contents

Chapter 1

Introduction

A noteworthy branch of the businesses all over the world depend upon the software industry for product development, production, marketing, support and services. That's why the importance of improving software quality becomes more evident. By conducting software tests in order to check its correctness and completeness during its development and before the release of the products, it is possible to increase the confidence in the software quality. Nowadays, the complexity and size of the software is growing and manual testing is becoming impractical. Software testing has evolved throughout years leading to improvements on overall testing techniques, such as, test cases construction, test automation, among others.

Nowadays, most of these software systems often rely on graphical user interfaces (GUIs) instead of offering textual menus and built-in typed commands. GUIs embody an important role on the acceptance of a software system to its end-users. They correspond to the front-end of the system, making them the intermediary between the user and the back-end. Its correctness, graphical power, ease of use and appropriateness settle on whether system users will eventually make use of the software.

GUI testing is an area of increasing importance, where the tests are performed from the end users point of view. Software companies have the best of interests on finding defects on their products before their costumers' do, not only to meet user demands and therefore increase confidence in relation to their software, but also to induce correctness and commitment with them. For these reasons, GUI testing is extremely necessary. It is particularly time consuming, labour-intensive, expensive and difficult. Presently used GUI testing methods are almost *ad hoc* and require test engineers to manually develop the necessary scripting to perform test execution, and though evaluate if the GUI is effectively tested. However, there are some tools that can help improving GUI testing process. Some of these tools exploit a broadly accepted method that generates GUI test scripts which relies on the capture/playback technique. Such technique requires testers to perform labour-intensive interaction with the GUI via mouse events and keystrokes. During interaction user events are recorded into scripts which and can be automatically played later for GUI testing. However, when different inputs are required to conduct the test or even if the GUI changes, it is then required to re-generate the test scripts. In addition, it is hard to cover all possible test cases for all GUI components and capture/playback method often records redundant data [1].

The use of a model to describe the behaviour of a system is an established and key advantage regarding testing. Models can be used in numerous ways, for instance, to improve quality of software documentation, code generation and test case generation. Model-based testing represents the automation of the design of black-box tests. The usage of a model to describe the behaviour of a GUI in combination with an automated test tool to generate test cases, execute those tests and report errors found, can dramatically reduce the time required meant for testing software.

Recently, model-based testing has been receiving attention due to the potential to automate test generation and increasing model driven software engineering practices. Nevertheless, the usage of uncommon and unfeasible modelling notations, the lack of integrated tool environments and support, the difficulties inherent to the constructions of models, the test case explosion problem, the gap between the model and the implementation, remain as obstacles regarding the adoption of model-based GUI testing approaches. In addition, the models used are often textual models and usually testers and modellers prefer working with visual/graphical notations.

1.1 Scope

The establishment of UML as a standard for object-oriented design has contributed for increasing confidence towards modelling. GUI modelling represents the process required to create a representation of a GUI in a higher level of abstraction, by defining its behaviour, as well as relationships and restrictions among its components. Models can be constructed using several types of notations, ranging from informal to formal specification languages. Depending on the situation in hands, it is necessary to choose the notation that suits the best of interests. The richness of modelling techniques in software design and the impact of modelling on software testing, has guided to the definition of a set of testing practices, known as Model-Based Testing (MBT).

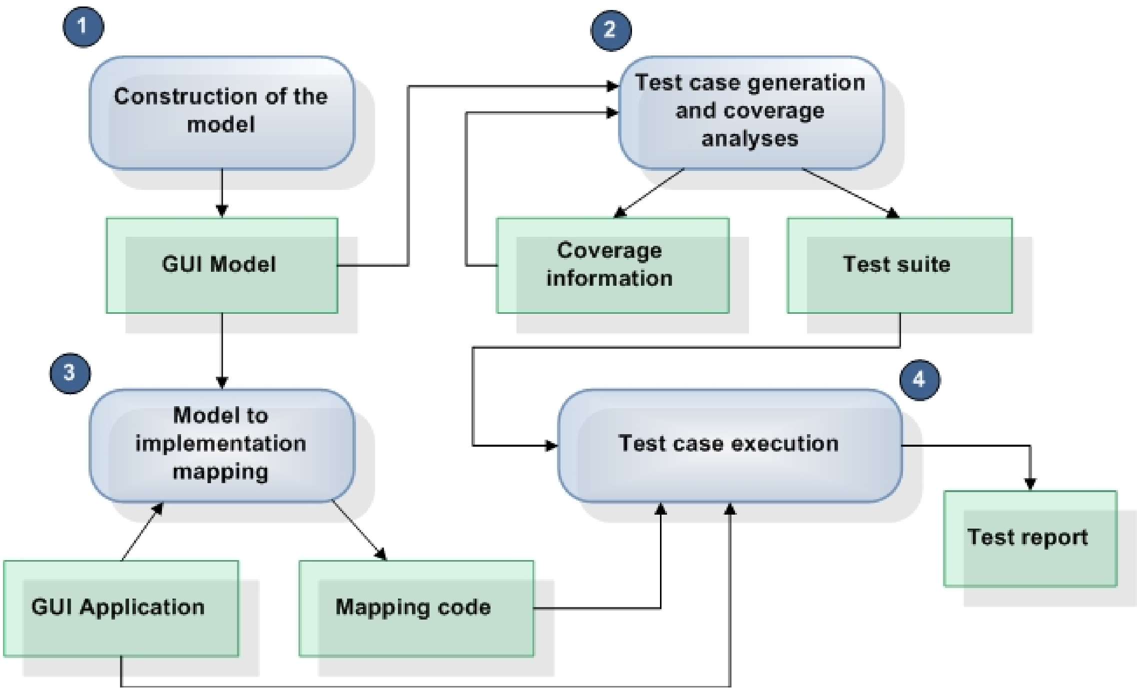


Figure 1 – Simplified view of the generic model-based GUI testing process

Figure 1 illustrates the activities and artefacts involved in the generic model-based GUI testing process, which embraces the following steps [2]:

1. **Construction of the model** – the model-based testing process starts with the construction of the model of the application under test (AUT). The model captures the requirements of the AUT [3].

2. **Test case generation and coverage analyses** – the model is then used as input to generate test cases according to given coverage criteria.

3. **Model to implementation mapping** – the model can normally be translated by means of a tool, to interactively relate the abstract user actions defined in the model with actions on physical GUI objects in the application under test, generating code automatically.

4. **Test case execution** − test cases are executed over the AUT and the results obtained and states reached are compared with the expected results and expected states described in the model.

1.2 Problem

One of the several activities that testers must perform during a model-based testing process is **build models**. Most of the times, these models are **textual** and require deep knowledge of the formalism details which may involve hard work making modelling an intensive task. Much often, testers do not have the acceptance and even patience to build such models.

UML is a graphical notation widely used in industry. However, their limitations for constructing GUI models are known [4] and their inconsistencies [5] are a drawback when trying to use UML on a model-based testing process.

Nowadays, software applications are becoming bigger, therefore more complex with composite GUIs, making **manual testing** of such GUIs a much more difficult activity than before. In addition, when a GUI is modified, the tester needs to start the process from the beginning, redefining the test suite and executing the tests over again. With MBT it is possible to automate more of the testing process. If the GUI model changes, test cases are automatically calculated and executed.

Another problem regarding GUI testing the difficult task of choosing the appropriate coverage criteria to test different features of the GUI, such as, navigation map (describes how the navigation - opening and closing of windows – of the system happens) and other abstraction levels such as windows dialogs' behaviour.

1.3 Goals

The main goal of the research work described in this dissertation is to overcome some of the problems with GUI testing some of which were identified in the previous section:

- Develop a **visual modelling front-end**, hiding as much as possible, the formal details from both modellers and testers. These visual models will be comprised by a set of diagrams to describe, at a high level of abstraction, the structure and behaviour of GUIs. The notation should look familiar to modellers and testers, and should be supported, as much as possible, by existing editors. The graphical notation is planned to hide formalism details from testers, diminishing the modelling effort required for GUI modelling;

- Delineate a set of rules to translate the graphical notation into a textual object-oriented formal specification language, such as Spec# [6];
- Develop a tool to support the translation process and maintain consistency between the visual model (a set of diagrams) and the textual formal model. Diagrams should be treated as views over the formal model. The formal model may have additional details that are not present in any of the visual diagrams. Expressions shown in selected elements of the diagrams should be written in the underlying formal notation (Spec#).

1.4 Outcomes

The current dissertation expects the following outcomes to be accomplished:

- A new modern visual abstract notation comprised by a set of UML profiles and a set of internal rules, with which consistent models may be constructed if following such rules (VAN4GUIM);
- A set of rules to translate the visual abstract notation into Spec#;
- A software tool supporting the translation mechanism between the visual notation and Spec# (XMI2Spec#);
- A scientific paper with the purpose to describe the approach proposed in this dissertation and the results achieved.

1.5 Dissertation Outline

This dissertation is organized in five main chapters. **Chapter 1** gives an introduction regarding overall GUI testing and a further presentation over model-based GUI testing. It also gives an insight over the scope of this dissertation, describing the model-based GUI testing process. The problems concerning GUI testing and the activities that testers must perform during a model-based testing process are indentified. In addition, it describes the goals that have been defined to overcome the acknowledged problems, and also provides an overview of the outcomes that are expected to be realized. **Chapter 2** presents several GUI modelling approaches. At the end, a discussion about these approaches is opened. **Chapter 3** describes the UML Profiles of the newly created notation - Visual Abstract Notation for GUI modelling (VAN4GUIM). In addition, it explains the proposed model-based testing process as well as the structure of the GUI model that is created whenever using VAN4GUIM. Topics such as coverage criteria over the visual model, the VAN4GUIM translation rules to Spec# and model-based testing with Spec Explorer are also covered in this chapter. **Chapter 4** features the achieved results of a case study in order to evaluate and validate the proposed approach. Finally, **Chapter 5** presents the achievements of this dissertation and discusses future expansion issues for it.

Chapter 2

GUI Modelling

The concepts of visually modelling software, as well as the tools to support this activity, have been around for a few years. Modelling represents the practice of creating abstractions of a system to maintain as a concept or a set of concepts flow. When applied to software, the world modelling naturally leads to UML diagrams. By making use of software modelling one is capable to: develop, communicate and discuss software models; assure that the system under specification is being correctly built; iteratively develop software, where the models and other higher abstraction levels facilitate rapidly changes or updates. There are at least, three kinds of models that are worth noting: models that track software development process; models to automate software generation and models for MBT.

Most of GUI models used are not adequate for MBT. MBT requires description of both behaviour and structure of the GUI under test. Pre and post formal notations, such as VDM++ [7] and Z [8], are good at representing the state but they are not indicated to represent the behaviour. It is possible however, to extend such notations in order to overcome such boundaries.

GUI modelling addresses quite a few challenges such as, the description of windows behaviour, use case scenarios and navigation maps. Modelling the behaviour of a window induces an eminent setback, the state explosion problem [3]. Scenarios are used to structure the GUI model in different levels of abstraction with the purpose to model user interactions. Modelling the behaviour of a GUI represents the notable challenge in GUI Modelling.

A more detailed study was performed over several GUI modelling approaches. As such, the current chapter presents and discusses these approaches.

2.1 UML Extensions for GUI Modelling

Currently, GUIs play an important role in most of software systems, as they represent the fore-front of the system. UML is a natural candidate for GUI modelling since it represents a standard notation for object-oriented modelling of applications. GUIs can be decomposed in two main groups: a dynamic or **behaviour** group and a static or **layout** group [9]. While the dynamic group can be modelled using existing UML diagrams and elements, GUI layout cannot, due to the fact that all existing UML diagrams are not layout-aware. In addition, it is not clear and simple to identify how UI elements, such as user tasks and display, are supported by UML. So, it is necessary to make use of UML extension mechanisms, like **constraints**, **tagged values**, and **stereotypes**, in order to provide more flexibility to the existing UML notation. With these extension mechanisms it becomes possible to construct several **UML profiles** particularly suited for GUI modelling.

UML profiles represent a clear-cut mechanism for adapting an existing meta-model with constructs that are specific to a particular domain, platform, or method. Furthermore, they can be used to create additional constraints to an existing meta-model. Stereotypes are one of the most important elements of UML 2.0 profiles. A **stereotype** extends a class of the meta-model by adding additional semantics and offers the possibility of modifying its syntax. The notation of stereotypes can be changed by defining icons in order to provide an intuitive meaning. This step becomes useful as it eases the task of matching designers' draft to models. **Constraints** are used to define the semantics of the stereotype and correspond to Boolean expressions [10].

2.1.1 The UML Profile for GUI Layout

Several researchers have recognized the lack of support for GUI's layout information in UML and thus have followed different approaches. Kai Blankenhorn and Wilhelm Walter [9] have developed an UML Profile for GUI Layout, which is a UML 2.0 profile that uses **Diagram Interchange** to store layout information while staying fully conform to standards. The diagram-interchange specification originates XMI from the XML metadata interchange format, which is used for storing information about the elements of a UML diagram. Although it serves quite well this purpose, it fails to express the graphical layout information of the diagram. Diagram Interchange, as an integral part of UML 2.0, adds drawing and layout information to every UML element, including size and position. This additional information enables new applications for UML that could not be handled properly with previous versions of the language. They claim their profile is one of the first to explore these new possibilities. The profile's meta-model makes use of stereotyped classes that are linked by constrained associations, taking benefit from UML 2.0 extension mechanisms.

The meta-model of the UML profile for GUI Layout can be seen in Figure 2.1. The main concept of the profile is a base element called *ScreenArea*, which is a stereotyped class representing the rectangular area of the screen. The base element is comprised by two subclasses, ***FunctionalScreenArea*** and ***ContainerScreenArea***. The first is used to present a fraction of functionality to the user, for instance, images, texts, links and forms. Finally, the *ContainerScreenArea* acts as a container of other *ScreenArea*s. Other elements worth notice are *StaticUIFunctionalities* – an element responsible to present a visual element on the screen - and *ActivatableUIFunctionalities*, which performs an action if clicked.

In order to improve the usefulness of the graphical language and to transfer the general look of designer sketches to models, creators have developed a set of stereotype icons, as illustrated in Figure 2.2. The right column displays the original screen elements, as extracted from designers' sketches.

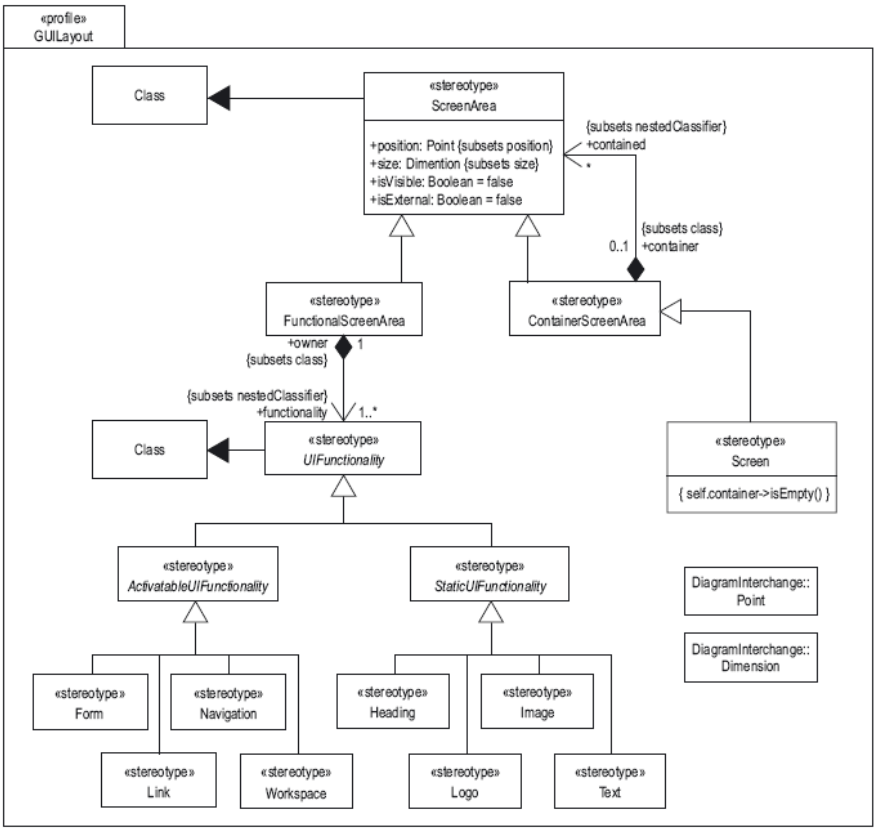


Figure 2.1 – Meta-model of the UML profile for GUI Layout

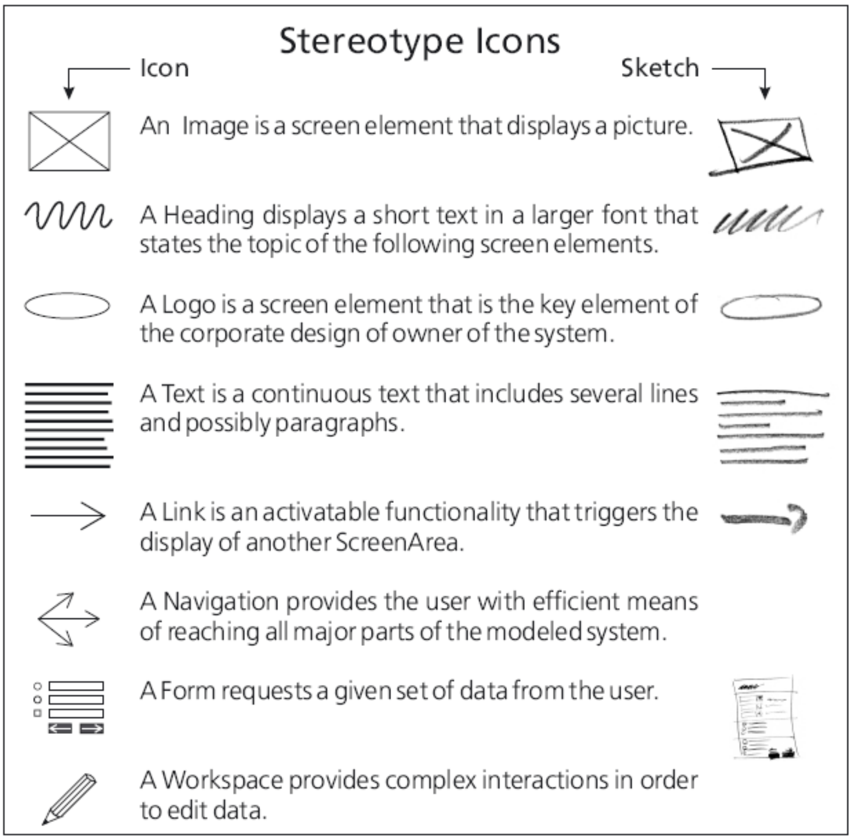


Figure 2.2 – Stereotype Icons

The diagram below (Figure 2.3) represents a rough layout of a web page. It contains the mentioned *ScreenAreas*, which dictate the functionality of the modelled web page. Three of the *ScreenAreas* are given names and are denoted as being abstract by having their name written in italic. A particular interest is that since Abstract *ScreenAreas* (Figure 2.1) cannot be instantiated, and therefore cannot be directly displayed on screen, they may however have concrete specialized *ScreenAreas*, which can be displayed instead.

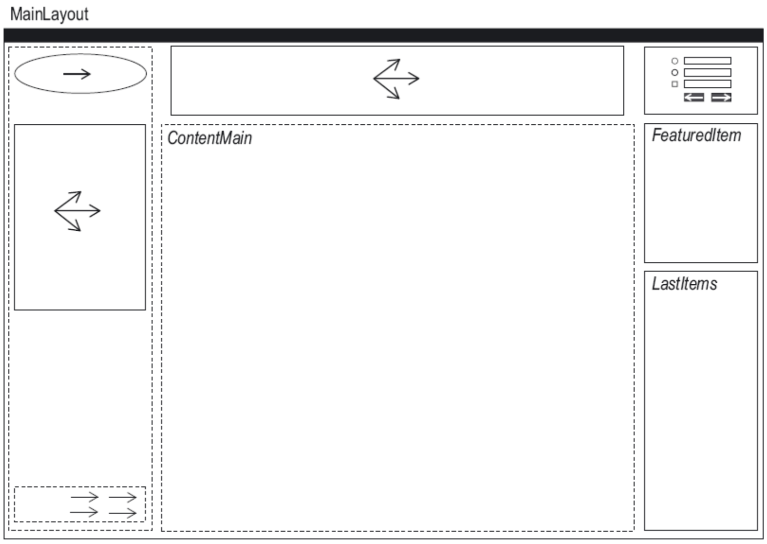


Figure 2.3 – A rough layout of a web page

It is possible to create and use a template for multiple pages that share the same layout, by specializing *ScreenAreas*. As stated, the rough layout displayed above can be used as a template, taking benefit of *ScreenArea* sub-classing and specialization, as can be seen in Figure 2.4.

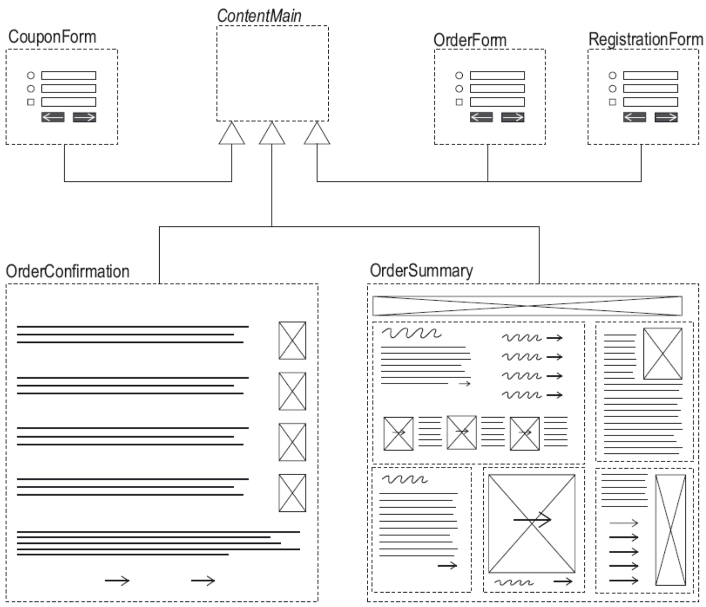


Figure 2.4 - *ScreenArea* sub-classing and specialization

2.1.2 UMLi

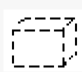
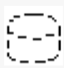
The Unified Modelling Language (UML) has been broadly accepted by software developers, but not so much by UI designers. In addition, there are no modelling languages and tools capable to support round-trip between developers and designers. In addition, UML lacks support for modelling certain aspects of GUIs [11]. To overcome this problem, a new notation was born to diminish the gap between developers and UI designers. The UMLi notation aims to be a light-weight extension to the UML notation with the purpose to provide greater support for UI design. This approach was proposed by [12], and is a result of a research held at the University of Manchester.

UMLi notation has been influenced by model-based user interface development environment (MB-UIDE) technology. In fact, MB-UIDEs provide a context within which declarative models can be constructed and related as part of the user interface design process. In addition, the authors of UMLi believe that the MB-UIDE technology offers many insights into the abstract description of user interfaces that can be adapted for use with the UML technology, such as techniques for specifying **static** and **dynamic** aspects of user interfaces using declarative models.

With UMLi, it is possible to model both structure and behaviour of a system. The notation defines three distinct types of models: **presentation** model, **domain** model and **behaviour** model. The presentation model represents the visual part of the user interfaces that can be modelled using object diagrams composed of interaction objects. Domain models specify classes and objects that represent the system entities, the domain elements. Behaviour models describe object collaboration and common interaction behaviour, such as tasks, actions and events.

Regarding domain models it is typical and indicated the usage of class diagrams. Concerning the presentation model, the interaction objects mentioned before, are also called *widgets* or ***visual components***. The tasks of selecting and grouping such objects are crucial for modelling UI presentations. It is found hard to perform these tasks because of the amount of interaction objects with different functionalities. Regarding UML, these tasks are not doable because UML does not provide graphical distinction between domain and interactional objects. However, using the UMLi *user interface diagram notation* it became possible to overcome such complicated tasks.

A set of stereotypes were defined in order to specify the UMLi user interface diagram notation. The user interface diagram is composed of six constructors that specify the role of each interaction object in a UI presentation and are defined in Table 2.1.

Stereotype	Icon	Description
«Free Container»		Element that represents a top-level interaction object that cannot be contained by any other interaction object.
«Container»		Element capable to group all interaction objects, except **Free Containers**.

«Inputer»		Element responsible for receiving information from users.
«Displayer»		Element responsible for sending visual information to users.
«Editor»		Interaction object that simultaneously represents an **Inputer** and a **Displayer**.
«ActionInvoker»		Element responsible for receiving information from users in the form of events.

Table 2.1 – UMLi user interface diagram' stereotypes

To model the behaviour of collaborative objects, activity diagrams come into place. Due to the ability to describe object collaboration and common interaction behaviour, UMLi activity diagrams provide a greater support for UI design than regular UML activity diagrams, by the usage of UML extension mechanisms.

Modelling transitions are useful for **selection states**. A combination of transitions, forks and joins is adequate for 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. However, for the designing of interactive applications there are situations where these constructors can be held to be rather low-level, leading to complex models. The following behaviours are common interactive application behaviours, but usually result in complex models.

1. Order **independent** behaviour is presented in Figure 2.5.

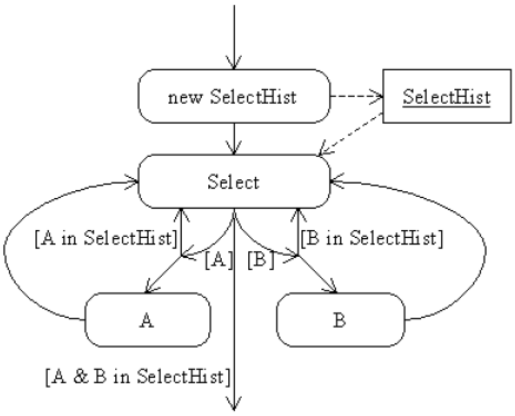


Figure 2.5 – Order independent behaviour

Activities A and B are called selectable activities since they can be activated in either order on demand by users who are interacting with the application. Thus, every selectable activity should be executed once during the performance of an order independent behaviour. Further, users are responsible for selecting the execution order of selectable activities. An order independent behaviour should be composed of one or more selectable activities. An object with the execution history of each selectable activity (in point 1 above) is required for achieving such behaviour.

2. The **optional** behaviour is presented in Figure 2.6.

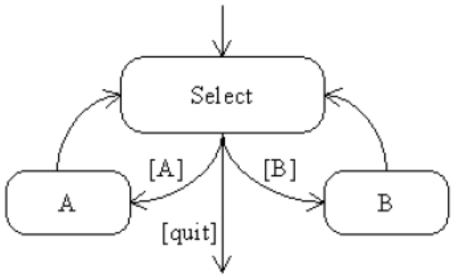


Figure 2.6 – Optional behaviour

Users can execute any selectable activity any number of times, including none. In this case, users should explicitly specify when they are finishing the **Select** activity. Like the order independent behaviour, the optional behaviour should be composed of one or more selectable activities.

3. The **repeatable** behaviour is presented in Figure 2.7.

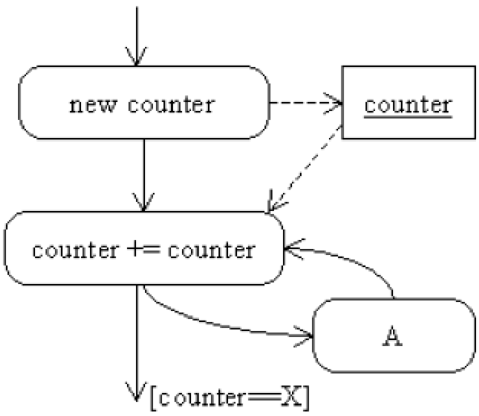


Figure 2.7 – Repeatable behaviour

Unlike the order independent and optional behaviours, a repeatable behaviour should have only one associated activity. **A** is the associated activity of the repeatable behaviour. Further, a specific number of times that the associated activity can be executed should be specified. In the case of the diagram in Figure 2.8 (c), this number is identified by the value of X.

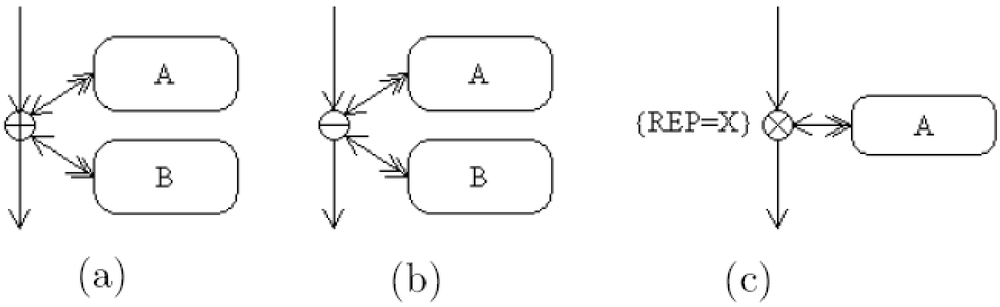


Figure 2.8 – The UMLi modelling of an order independent selection state in (a), an optional selection state in (b), and a repeatable selection state in (c).

Order independent and repeatable behaviours are common in interactive systems [12], UMLi proposes a simplified notation for them. The notation used for modelling an order independent behaviour is presented in Figure 2.8 (a). There we can see an order independent selector, rendered as a circle overlying a plus signal, ⊕, connected to the activities A and B by return transitions, rendered as solid lines with a single arrow at the selection state end and a double arrow at the selectable activity end. The order independent selector identifies an order independent selection state. The double arrow ended on return transitions identifies the selectable activities of the selection state. The distinction between the selection state and its selectable activities is required when selection states are also selectable activities. Furthermore, a return transition is equivalent to a pair of statechart transitions, one single transition connecting the selection state to the selectable activity, and one non-guarded transition connecting the selectable activity to the selection state, as previously modelled in Figure 2.8 (a). In fact, the order independent selection state notation can be considered as a macro-notation for the behaviour described in Figure 2.8 (a). The notations for modelling optional and repeatable behaviours are similar, in terms of structure, to the order independent selection state. The main difference between the notations of selection states is the set of symbols used by their selectors. The optional selector, which identifies an optional selection state, is rendered as a circle overlaying a minus signal. The repeatable selector which identifies a repeatable selection state is rendered as a circle overlaying a times signal. The repeatable selector additionally requires a REP constraint, as shown in Figure (c), used for specifying the number of times that the associated activity should be repeated. The value X in this REP constraint is the X parameter in Figure 2.8 (c). The notations presented in Figures 2.8(b) and (c) can be considered as macro-notations for the notation modelling the behaviours presented in Figures 2.8 (b) and (c) [12].

2.1.3 Wisdom Profile

The Wisdom Profile is proposed by Nunes and Cunha [13], for the documentation, specification and design of interactive systems. They propose a minimal set of extensions for a UML profile for interactive systems development taking advantages on human-computer interaction domain knowledge under the notation and semantics of the UML. The extension was named Wisdom UML Profile due to the fact that it was originally created with the purpose to maintain the Wisdom (Whitewater Interactive System Development with Object Models) software development method. Not only the authors believe that, despite the lack of support for building complex interactive systems, UML is the most universally used modelling language, but also that Wisdom extensions are broadly applicable in different process contexts.

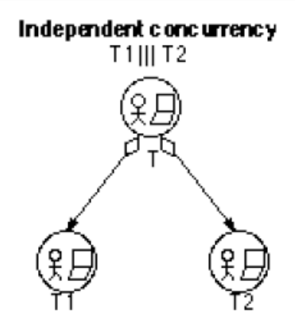
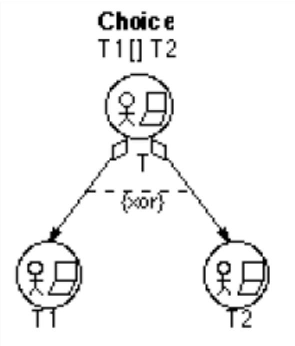
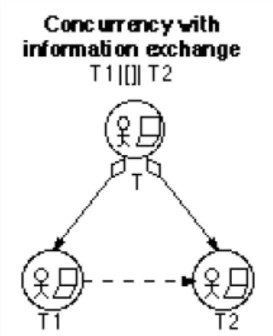
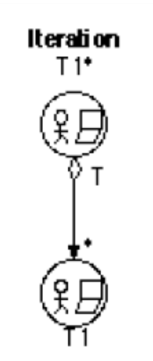
The Wisdom approach suggests two important models: the **analysis model** and the **interaction model**. The latter includes the information, dialogue and presentation dimensions, mapping the conceptual architectural models for interactive systems, while maintaining the desired separation of concerns. The analysis model encompasses the UML profile architecture and shared information. During the design phase, the interaction model embraces two other models: the **dialogue model** and the **presentation model**. The former specifies the dialogue structure of the application, using an UML based approach of the ConcurTaskTrees (CTT) notation. One of the most important features of the CTT notation is the capability to express temporal relationships between tasks. Accordingly, the creators of the Wisdom Profile took advantage of the UML constraint extension mechanism to express such relationships between tasks. The result of the adaption of the CTT temporal relationships can be seen in Table 2.2. Furthermore, the **dialogue model** requires an additional definition of stereotypes, constraints and tagged values. As such, authors define the «Refine Task» **stereotype**, which is an association between two tasks, denoting that the target class (subtask) specifies the source task (parent task) at a lower level of detail. The task association is unidirectional and can only be used between task classes. As for the Refine task association it implies a set of constraints, namely:

- {xor} – represents an UML constraint to be applied to a set of associations. It indicates that over the set, exactly one is manifest for each associated instance;
- {sequence} – symbolizes a constraint with the purpose to be applied to a set of «refine task». It specifies that over that particular set, associated instances become active in sequential order, i.e., one associated instance activates the other when it becomes inactive;
- {deactive} – this constraint is applied to two «refine task» associations, specifying that one associated instance explicitly deactivates the other, when it becomes active.

In order to perform task allocation the following tagged values are applied to the task classes:

- {abstract task} – tasks that require complex activities and whose performance cannot be univocally allocated;

- {user task} – represents all the tasks performed by the used that do not imply any interaction with the system;
- {application task} – tasks completely executed by the application;
- {interaction task} – tasks performed by the user interaction with the system.

Temporal Relationships	Description
	**Independent concurrency (T1|||T2)** – denotes that actions belonging to two tasks (T1 and T2) can be performed in any order without any specific constraint;
	**Choice (T1[]T2)** – denotes that it is possible to choose form a set of tasks and once the choice has been made the chosen task can be performed, while other tasks are not available;
	**Concurrency with Information Exchange (T1|[]|T2)** – same has independent concurrency but the tasks have to synchronize in order to exchange information;
	**Iteration (T*)** – denotes the task (T) is performed repeatedly until the task is deactivated by another task;

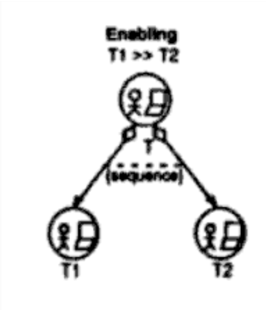
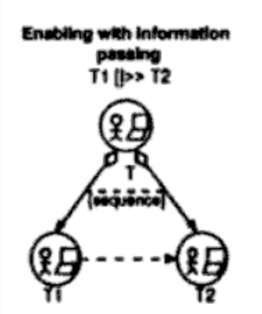
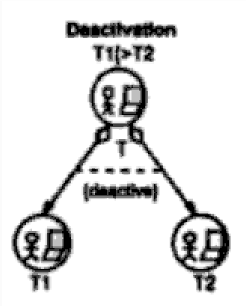
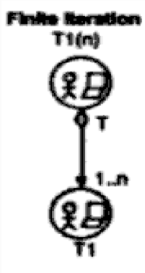
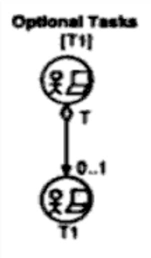
	**Enabling (T1>>T2)** – denotes that the second task (T2) is activated once the first task (T1) terminates;
	**Enabling with Information passing (T1 [] >> T2) -** denotes that the second task (T2) is activated once the first task (T1) terminates and sends information to the task T2.
	**Deactivation (T1[>T2)** – denotes that the first task (T1) is definitely deactivated once the second task (T2) terminates;
	**Finite Iteration(s) (T1(n))** – same as iteration but the task (T) is performed n times;
	**Optional Tasks ([T])** – denotes that the performance of a task is optional.

Table 2.2 – UML adaptation of ConcurTaskTrees. The circles with the person icon and computer represent a stereotype «task». Furthermore, all the associations between «task» classes are stereotyped «refine task» associations. Finally {xor}, {sequence} and {deactivate} are UML constraints [13].

As to the design model, it defines the physical realization of the interactive system, centring on the structure of the different presentation entities in order to realize the physical interaction with the user. Like dialogue model, the authors propose a set of UML extensions to support the design model. They have defined the following stereotypes:

- «Interaction Space» - stereotype responsible for receiving and presenting information to the users supporting their task. Typically, interaction spaces are organized in hierarchies and containment relationships can occur between interaction spaces;
- «Navigate» - represents an association stereotype between two interaction space classes indicating the move of a user from one interaction space to another. The navigate association can be unidirectional or bidirectional. The latter means that there is an implied come back in the navigation. Users navigate in interaction spaces while performing complex tasks and a change between interaction spaces generally requires a switch of thinking from the user;
- «Contains» - is an association stereotype between two interaction space classes and representing that the source class (container) contains the target class (content). The «contains» association can only be used between interaction space classes and is unidirectional.
- «input element» - is an attribute stereotype denoting information received from the user, typically information that the user can operate with;
- «output element» - is an attribute stereotype denoting information presented to the user. For instance, information that the user can recognize but cannot manipulate;
- «action» - is an operation stereotype indicating something that a user can perform in the physical user interface that will cause a significant change in the internal state of the system, i.e., changes in the long term information of the system (entities), request for signification functionality, changes in context of the user interface, and others.

2.2 usiXML

User Interface Extensible Markup Language (usiXML) represents a paradigm which allows designers to develop user interfaces, based on the User Interface Description Language (UIDL). Hence, it becomes possible to specify a user interface **at** and **from** different levels of abstraction while maintaining the mappings between these levels, whenever required. In addition, the development phase can start from any level of abstraction, where the final user interface may be composed of several contexts of use at other levels of abstraction. The usiXML model has been developed in the context of the Cameleon research project being held by Jean Vanderdonckt [14].

A considerable number of User Interface Description Languages (UIDL) subsists to attend to different characteristics of user interfaces. However, despite the fact UDLI addresses aspects such as portability, device independency and user-centred design, it induces a broadcast of several dialects that are not both largely used and do not support interoperability between tools that were developed in the order of UIDL. Furthermore, some UIDLs lack support to attend both traditional GUIs and Multimodal User Interfaces (MUI) needs. As response, a new notation called usiXML emerged to overcome the above problems. This notation is based on five main concepts: **expressiveness of UI**, **central storage of models**, **transformational approach**, **multiple development paths**, **and flexible development approaches**. The **expressiveness of UI** remarks that any UI is expressed depending on the context of use due to a suite of models analyzable and editable by software. As for the **central storage of models** key, it denotes each model is stored in a model repository where all UI models are expressed according to the same UIDL. Regarding the **transformational approach** concept, each stored model in the model repository may be subject to one or many transformations supporting various development keys. The **multiple development paths** refer to the development steps that can be combined together to form development paths that are compatible with the organization's constraints, conventions, and context of use. Finally, the **flexible development approaches** point to top-down, bottom-up, wide-spreading approaches. They are supported by flexibly following alternate development paths and enable designers to freely change between these paths depending on the modifications imposed by context of use.

UsiXML is a language designed to support **multi-directional UI development**. MDUI is based on a framework – *Cameleon Reference Framework* [15] – that defines UI development steps for multi-context interactive applications. This framework is illustrated in Figure 2.9. The CR framework structures development processes for **two** contexts of use into four development steps: **Final UI (FUI)**, **Concrete UI (CUI)**, **Abstract UI (AUI)**, and **Task & Concepts (T&C)**. **FUI** represents the operational UI, any UI running on a particular platform. Regarding **CUI**, it concretizes an abstract UI for a given context of use into Concrete Interaction Objects (CIOs), in order to define widgets layout and interface navigation. In addition, it converts a FUI into a UI definition independently of the computing platform. Although a CUI makes explicit the final Look & Feel of a FUI, it is still a mock-up that runs only within a particular environment. **AUI** defines interaction spaces by grouping subtasks according to various criteria, a navigation scheme between the interaction spaces and selects Abstract Interaction Objects (AIOs) for each concept so that they are independent of any context of use. An AUI abstracts a CUI into a UI definition that is independent of any modality of interaction. Finally, **T&C** describes the various tasks to be carried out and the domain-oriented concepts as they are required by these tasks to be performed. These objects are considered as instances of classes representing the concepts manipulated.

The framework defines three basic transformation types: **Reification**, **Abstraction**, and **Translation**. MDUI development refers to a UI tool that enables a designer to start a development activity from any entry point of the framework, and to a support in the performance of the basic transformation types and combinations from Figure 2.9. Hence, in order to perform such development, the authors gathered two crucial requirements:

a) A language that enables both *expression* and *manipulation* of the model at each development steps and for each context of use.

b) A mechanism to express design knowledge that would provide a substantial support to the designer in the realization of transformations.

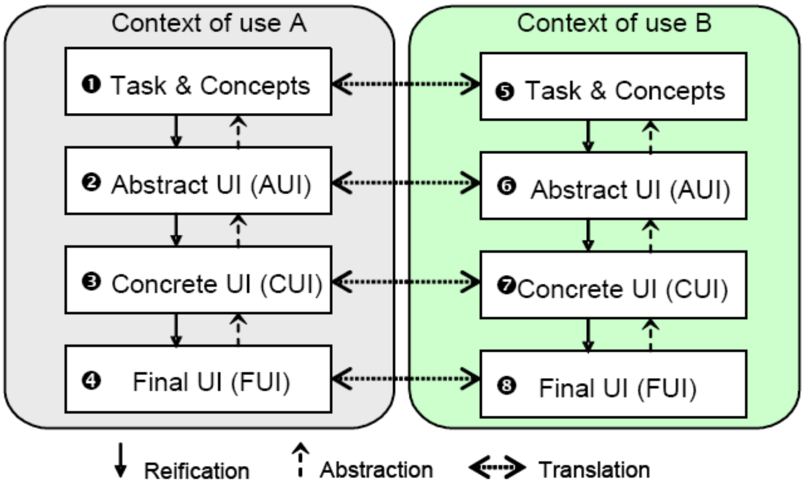


Figure 2.9 – Cameleon Reference Framework

As a response for the requirements above, usiXML is introduced and defined, and a set of graphical transformation techniques is introduced and defined, respectively.

Since usiXML comprises **visual**, **formal** and **seamless** features, graphical transformation techniques were chosen to formalize the language. The latter feature refers to the capability offered by usiXML in representing manipulated artefacts and rules within a single formalism. Consequently, this formalism applies equally to all levels of abstraction.

The authors propose that each level of Figure 2.9 could be decomposed into two sub-levels:

- At the FUI level, the rendering materializes how a particular UI coded in one language (markup, programming or declarative) is rendered depending on the UI toolkit, the window manager and the presentation manager. For example, a push button programmed in HTML at the code sub-level can be rendered differently, here on MacOS X and Java Swing. Therefore, the code sub-level is materialized onto the rendering sub-level.

- Since the CUI level is assumed to abstract the FUI independently of any computing platform, this level can be further decomposed into two sub-levels: platform-independent CIO and CIO type. For example, a HTML push-button belongs to the type "Graphical 2D push button". Other members of this category include a Windows push button and XmButton, the OSF/Motif counterpart.

- Since the AUI level is assumed to abstract the CUI independently of any modality of interaction, this level can be further decomposed into two sub-levels: modality-independent AIO and AIO type. For example, a software control (whether in 2D or in 3D) and a physical control (e.g., a physical button on a control panel or a function key) both belong to the category of control AIO.

- At the T&C level, a task of a certain type (here, download a file) is specified which naturally leads to AIO for controlling the downloading.

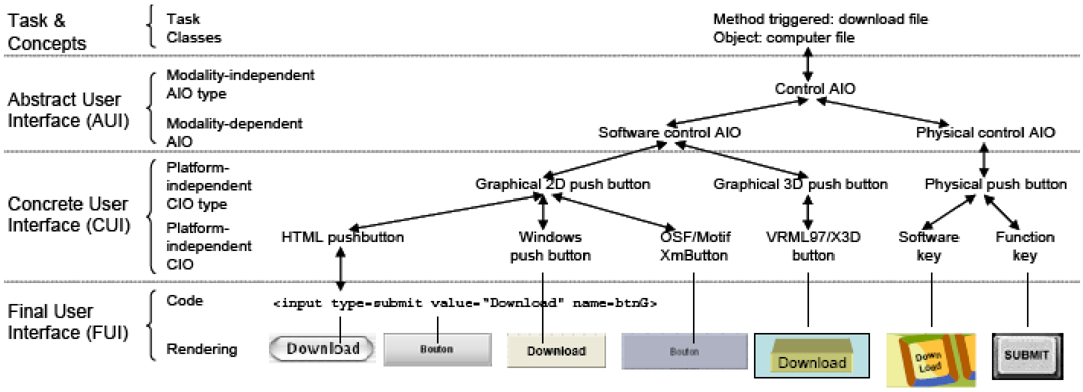


Figure 2.10 – Example of transformations in usiXML

Thanks to these four abstraction levels, it is possible to establish mappings between instances and objects found at the different levels and to develop transformations that find abstractions or reifications or combinations. In addition, the usiXML language provides tool support at the various levels identified in Figure 2.10.

In order to face express multi-directional development of UIs, usiXML features a collection of basic UI models, such as domain model, task model, AUI model, CUI model, context model and mapping model (Figure 2.11) and a transformation model (Figure 2.12) [14].

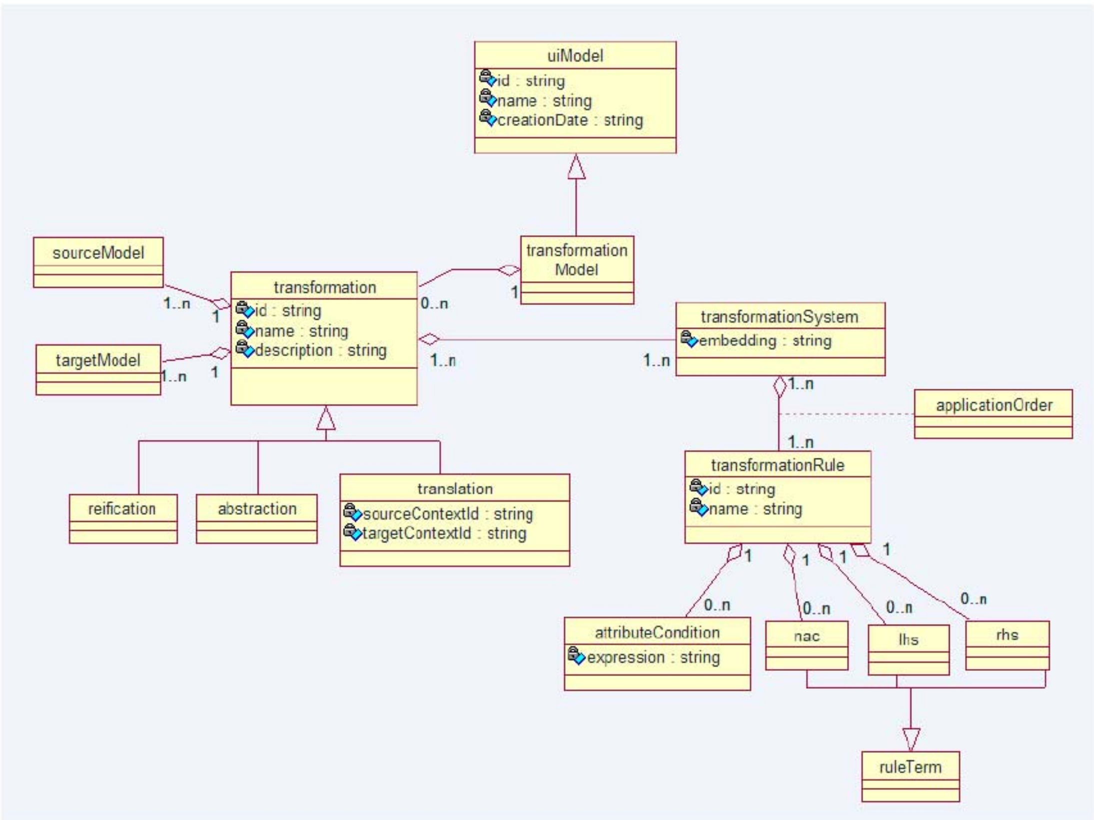


Figure 2.11 – Transformation model as defined in usiXML

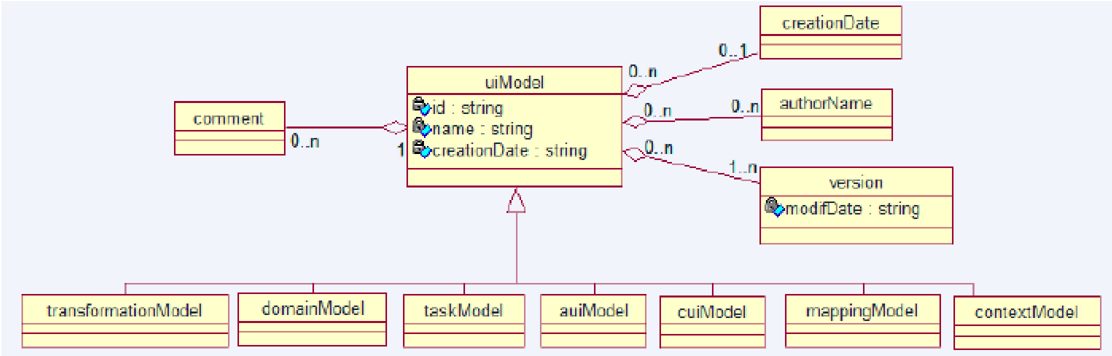


Figure 2.12 – usiXML model collection

Beyond the AUI and CUI models that reflect the AUI and CUI levels, the other UI models are defined as follows [14]:

- **taskModel**: is a model describing the interactive task as viewed by the end user interacting with the system. A task model represents a decomposition of tasks into sub-tasks linked with task relationships. Therefore, the decomposition relationship is the privileged relationship to express this hierarchy, while temporal relationships express the temporal constraints between sub-tasks of a same parent task.

- **domainModel**: is a description of the classes of objects manipulated by a user while interacting with a system.

- **mappingModel**: is a model containing a series of related mappings between models or elements of models. A mapping model serves to gather a set of inter-model relationships that are semantically related.

- **contextModel**: is a model describing the three aspects of a context of use in which an end user is carrying out an interactive task with a specific computing platform in a given surrounding environment. Consequently, a context model consists of a user model, a platform model, and an environment model.

- **uiModel**: is the topmost super class containing common features shared by all component models of a UI. A uiModel may consist of a list of component model without any order and any number, such as task model, a domain model, an abstract UI model, a concrete UI model, mapping model, and context model. A UI model does not need to include one of each model component. Moreover, there may be more than one of a particular kind of model component.

2.3 Canonical Abstract Prototypes

The concept of abstract user interface prototypes offer designers a form of representation for specification and exploration of visual and interaction design ideas that is intermediate between abstract task models and realistic or representational prototypes. They represent an intermediate form that can speed the user interface design process and improve the quality of the results. As abstractions, they can serve as an intermediate bridge between task models and realistic designs, smoothing, simplifying, and systematizing the design process.

Abstract prototyping is specified by using **content models**. A content model represents the contents of a user interface, without regard for the details of what the user interface looks like and how it behaves. In a content model, a user interface is a ***collection of materials***, ***tools*** and ***working spaces***. They are all described in the terms of their purposes or how they are intended to be used. Furthermore, content models help to simplify what a user interface must contain and how its contents are assembled before entering into a deep detail. The abstraction levels that materials, tools and workspaces imply, offer a great and simple support to describe a lot more than a simple common user interface.

Canonical Abstract Prototypes (CAP) are an extension to usage-centred design that provides a formal vocabulary for expressing visual and interaction designs without concern for details of appearance and behaviour. CAPs embody a model specifically created to support a smooth progression from abstraction toward realization in user interface design. The necessity inherent for their development arose from a growing awareness among practitioners of usage-centred design regarding the substantial conceptual gap between the task models needed to drive an effective design and the detailed, realistic prototypes needed for successful implementation. Simple content inventories had proved both too abstract and too imprecise for resolving design issues in very complex user interfaces. A sample regarding Canonical Abstract Prototypes is illustrated in Figure 2.13. Each CAP component has a specific abstract interactive function, for instance, create information or provide a notification. These functions are represented by **simple symbols**. Canonical Abstract Components model not only the interactive functions to be provided by a user interface, but also the position, size, layout, and composition of the user interface features. The notation, indeed the entire scheme for Canonical Abstract Prototypes, was devised so as to promote precise and consistent modelling while maximizing the utility for practicing visual and interaction designers working on real-world projects.

The use of a standard set of abstract components serves a number of purposes. Easy selection from a palette of available components can speed and simplify the process of abstract prototyping. The standard form facilitates comparisons of designs and makes it easier both to recognize and to describe recurring patterns or common configurations. In addition, because the abstract components are related by their specific interactive function to particular realizations, Canonical Abstract Prototypes provide direct guidance for the visual and interaction design of the user interface. Abstract Prototypes were devised for designing user interfaces in order to support design decision making at a higher level of abstraction than typical paper prototypes.

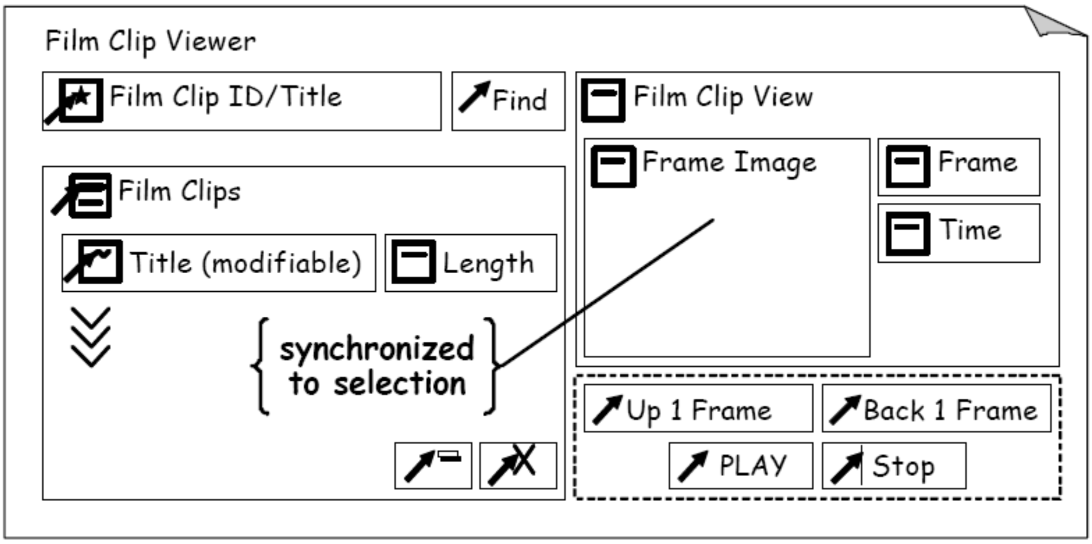


Figure 2.13 - Example of a Canonical Abstract Prototype displaying concepts of key notational elements.

Each Canonical Abstract Component is comprised by a symbolic graphical identifier and a descriptive name. The graphical symbols have as purpose to serve as learned shorthand for the various functions available. The notation is quite simple, since it is built on two basic symbols: a **generic tool** or **action** and a **generic material** or **container**. Materials are the containers, content, information or data. Tools are the actions, operators, mechanisms, or controls that can be used to create, manipulate, transform or operate upon materials. The symbolic notation is built from two glyphs: a square to represent a **container** and an arrow to represent an **action**. The combination of a container and an action form a generic hybrid component. Figure 2.14 illustrates the symbols for Canonical Abstract Components.

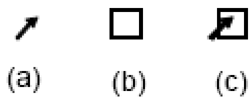
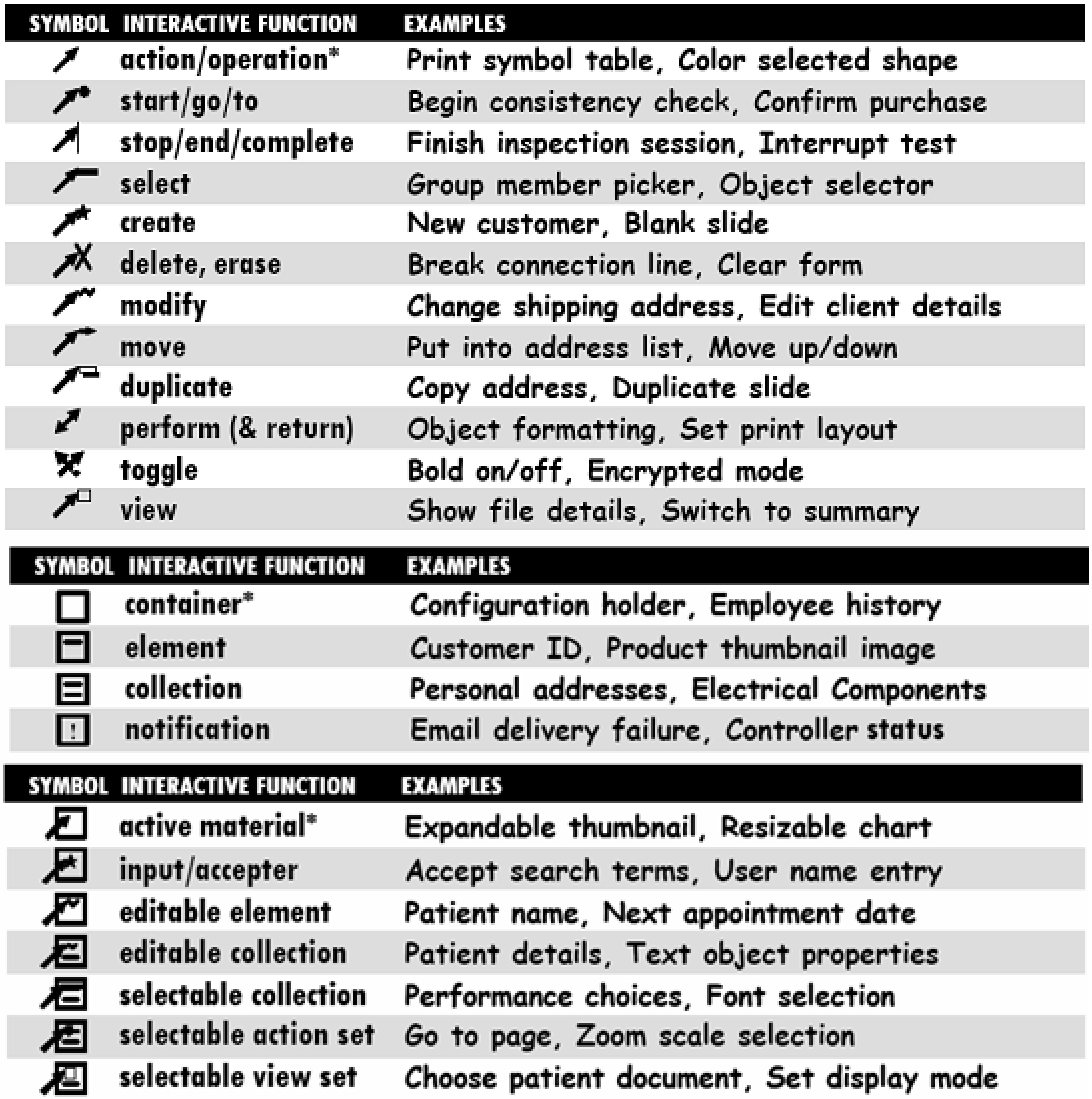


Fig. 2.14 – Basic symbols for Canonical Abstract Components where (a) represents a generic abstract tool, (b) represents a generic abstract material and (c) figures a generic abstract material.

The generic container can be used as the abstract representation of any type of material, the generic action can represent any sort of tool, and the generic hybrid can represent any component with characteristics of both.

The original set of abstract components and their symbolic representation has suffered from several improvements through the years. At the moment, the notation comprises a collection of 21 components, which includes 3 generic components, 11 specialized abstract tools, 3 specialized abstract containers, and 7 abstract hybrids. The figure bellow details the complete set of Canonical Abstract Components. It first illustrates the canonical abstract tools, then the canonical abstract materials and, finally the abstract hybrids.



SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	action/operation*	Print symbol table, Color selected shape
	start/go/to	Begin consistency check, Confirm purchase
	stop/end/complete	Finish inspection session, Interrupt test
	select	Group member picker, Object selector
	create	New customer, Blank slide
	delete, erase	Break connection line, Clear form
	modify	Change shipping address, Edit client details
	move	Put into address list, Move up/down
	duplicate	Copy address, Duplicate slide
	perform (& return)	Object formatting, Set print layout
	toggle	Bold on/off, Encrypted mode
	view	Show file details, Switch to summary

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	container*	Configuration holder, Employee history
	element	Customer ID, Product thumbnail image
	collection	Personal addresses, Electrical Components
	notification	Email delivery failure, Controller status

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	active material*	Expandable thumbnail, Resizable chart
	input/accepter	Accept search terms, User name entry
	editable element	Patient name, Next appointment date
	editable collection	Patient details, Text object properties
	selectable collection	Performance choices, Font selection
	selectable action set	Go to page, Zoom scale selection
	selectable view set	Choose patient document, Set display mode

Fig. 2.15 - Canonical Abstract Components featuring (in order of appearance): abstract tools, abstract materials and abstract hybrids. Notice that the first element of each group with a '*' placed there are called generic elements, and the rest are elements of the specialization group.

2.4 ConcurTaskTrees

Task models play an important role in human-computer interaction field as they represent the intersection between user interface design and other engineering approaches. Further, they provide designers with a means of representing and manipulating a formal abstraction of activities that should be performed to reach user goals. A goal is either a desired modification of state or an inquiry to obtain information on the current state and can be associated with multiple tasks, where there can be different tasks to fulfil the same goal. Despite the fact that task models have always been well thought-out, only recently user interface designers and developers have realized their importance and the need for engineering approaches to task models to better obtain effective and consistent solutions.

There are many proposals for representing task models. Such notations vary from GOMS (Goals, Operators, Methods, and Selection rules) [16], UAN [17], CTT [18], MAD [19] and GTA [19]. A set of dimensions must be taken into account, in order to better choose the most appropriate notation. Such dimensions are ***syntax***, ***set of operators for task composition*** and ***level of formality***. Syntax means that the notation can be either graphical or textual. For instance, GOMS are mainly textual, as opposite to ConcurTaskTrees that are graphical. Graphical representations aim to better highlight the hierarchical structure. The set of operators for task composition is a point that distinguishes the proposed notations. It intends to identify the best notation that provides the richest set of **temporal operators**, in order to provide more flexible ways to perform tasks. The comparison can be seen in Table 2.4. Finally, the level of formality represents the point of abstraction importance to define the meaning of operators.

The ConcurTaskTrees is one of the most widely used notations for task modelling, specifically tailored for user interface model-based design [20]. This notation has been developed taking into account the previous experience in task modelling and adding new features to better obtain an easy-to-use powerful notation, to describe the dialogue in interactive systems. In fact, CTT provides the concept of hierarchical structure, exposing a wide range of granularity allowing large and small structures to be reused and, enables reusable task structures to be defined at both low and high semantic level. CTT introduces a rich set of graphical temporal operators, providing more possibilities than those offered by concurrent notations. These operators can be seen in Table 2.5.

As mentioned, in a model-based GUI testing approach, task models can be used to define the behaviour of user interfaces. They correspond to the logic of the user interactive layer of the system, and they describe assumptions about how the user will interact with it. In fact, using this approach, CTT grants a considerable number of advantages: they *focus on activities*, thus allowing testers to centre on the activities that users perform; its *set of temporal operators* provides richness and intuitive guidance to clearly express user actions; and, the notation itself is becoming quite popular

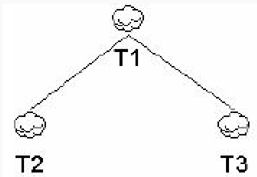
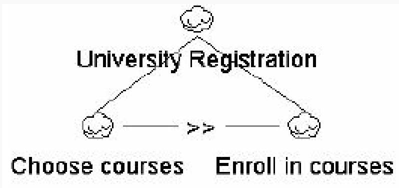
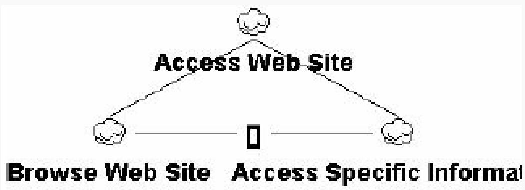
Operators	**Notation**				
	GOMS	**UAN**	**CTT**	**MAD**	**GTA**
Sequence	X	X	X	X	X
Order Independence		X	X		X
Interruption		X	X	X	
Concurrency	Only CPM-GOMS	X	X	X	X
Optional			X	X	
Iteration		X	X	X	X
Allocation			X		X
Objects			X		X
Performance	X		X		X
Pre-post condition	X		X	X	X

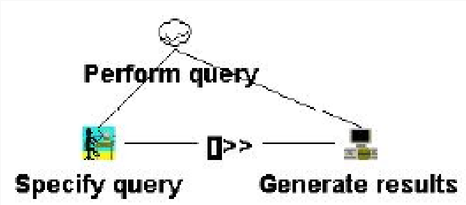
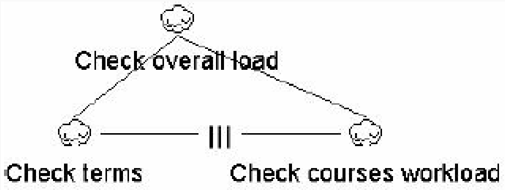
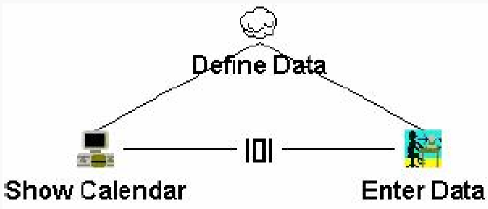
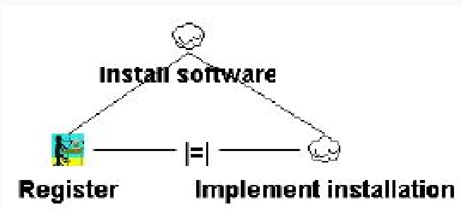
Table 2.4 – Comparison of operators among notations from task modelling

The integration of CTT into UML can be achieved by using UML extension mechanisms to represent elements and operators of the CTT notation. However, another approach consists of extending the UML meta-model, thus introducing a separate task model, and establishing relationships between the CTT elements and existing UML elements, becoming possible to represent CTT operators as UML connectors. Further, [20] have proposed a different approach that enables the integration of CTT in the UML through the extensions of UML 2.0 activity diagrams.

The first approach represents CTT as stereotyped class diagrams. Constraints associated with UML class, association and dependency stereotypes are defined to enforce the structural correctness of the CTT models. The major drawbacks of this approach are the low expressiveness of the notation and the lack of semantic validation of the CTT temporal constraints in terms of UML class diagrams. The second approach is outlined in [21] and covers the definition of an UML extension for Interactive Systems. It describes the integration points between UML models and task models in a complementary way. Yet, a unified integration at semantic and notational levels should be provided towards an effective incorporation of task models into UML.

The larger the complexity of the GUI, the more difficult the modelling task will be. Thus, tools such as CTTE [22] offer the possibility of modelling such systems, aiming to help end users in this complicated task.

Hierarchy		Tasks at same level represent different options or different tasks at the same abstraction level that have to be performed. Read levels as “In order to do T1, I need to do T2 and T3”, or “In order to do T1, I need to do T2 or T3”.
Enabling		Specifies second task cannot begin until first task performed. Example: I cannot enrol at university before I have chosen which courses to take.
Choice		Specifies two tasks enabled, then once one has started the other one is no longer enabled. Example: When accessing a web site it is possible either to browse it or to access some detailed information.

Enabling w/ info change		Specifies second task cannot be performed until first task is performed, and that information produced in first task is used as input for the second one. Example: The system generates results only after that the user specifies a query and the results will depend on the query specified.
Concurrent tasks		Tasks can be performed in any order, or at same time, including the possibility of starting a task before the other one has been completed. Example: In order to check the load of a set of courses, I need to consider what terms they fall in and to consider how much work each course represents.
Concurrent communicating tasks		Tasks that can exchange information while performed concurrently Example: An application where the system displays a calendar where it is highlighted the data that is entered in the meantime by the user.
Task independence		Tasks can be performed in any order, but when one starts then it has to finish before the other one can start. Example: When people install new software they can start by either registering or implementing the installation but if they start one task they have to finish it before moving to the other one.

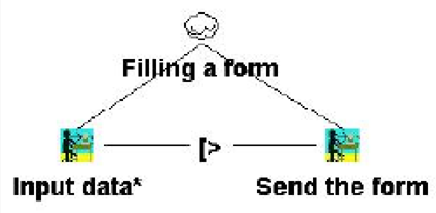
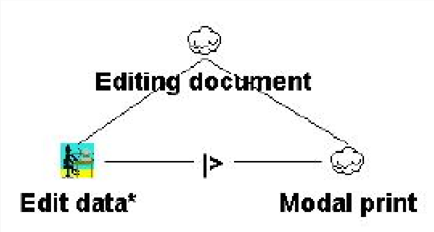
Disabling		The first task (usually an iterative task) is completely interrupted by the second task. Example: A user can iteratively input data in a form until the form is sent.
Suspend-Resume		First task can be interrupted by the second one. When the second terminates then the first one can be reactivated from the state reached before. Example: Editing some data and then enabling the possibility of printing them in an environment where when printing is performed then it is no possible to edit.

Table 2.5 – Temporal operators provided by CTT

2.5 Spec#

The Spec# programming system represents an attempt to develop a more cost effective way to maintain software in high standards. Spec# has been developed by Foundations of Software Engineering at Microsoft Research lab, in Redmond, USA. Spec# supports literate programming. It allows a Spec# program to appear spread over several separate sections in a document along documentation like text, tables, and diagrams.

The Spec# system consists of three domains: the **Spec# programming language**, the **Spec# compiler** (statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata), and the **Spec# static program verifier** (generates logical verification conditions from a Spec# program). Spec# offers quite a few key features such as: types (type aliases, constrained types, and non null types); relational and logical operators (implication and equivalence); collection types (tuples, sets, sequences and maps); patterns (literal, identifier, type, constructor and universal patterns); and quantifiers (forall and exists).

Since all of the specifications written in Spec# are executable, it is possible to specify invariants, pre-conditions, post-conditions, and executable method bodies in a high-level action language, with primitives to change the value of state variables, and even call external methods defined in .NET assemblies. Spec# provides the ability to build a formal specification of an interactive application, describing the actions that a user may perform when interacting with the system, in the terms of changes to the state of the application.

Windows' controls can be modelled using Spec#. Such interactive controls are composed of states, where each state is modelled by state variables. One or more variables are used for that purpose depending on the characteristics that are considered relevant from the modeller perspective. The user actions that take place with each control are modelled by methods. These methods are defined using pre-conditions, which include a clause to check when the window where the control is placed is enabled [3].

Regarding the modelling of windows issue, the top-level windows of the application are better modelled in separate namespaces or classes. Inside each module state variables are used to model the abstract state of that particular window and the controls inside of it, and methods are used to model the possible user actions on the specified window and on the controls of the window. All the actions inside each namespace have at least one pre-condition, corresponding to the window that is enabled. Windows can be modal or modeless. When enabled, a modal window disables all the other windows of the application. A window manager comes into hand to handle these situations, and is part of the model, where its state is also a part of the model state. For instance, when a method opens a window it should add that window to the window manager. When a window is removed, all its child windows are also removed. Concerning message boxes, they are also registered in the window manager but are not modelled with separate namespace because they have a simple structure, which only requires the user to press one of the shown buttons. As such, they can be modelled as a method with a parameter with the user's response.

2.6 Discussion

After providing some insight over a set of different GUI modelling approaches, it is time to make a discussion about the benefits and disadvantages of them.

The UML Profile for GUI Modelling proposed by Kai Blankenhorn and Wilhelm Walter is essentially oriented for layout design purposes. It does not provide any means to model the behaviour of a GUI. However, their approach is quite interesting, ambitious and supportive concerning the activity of the design process of a GUI. In addition, from the several approaches analyzed during this chapter, this approach is the first that explicitly makes uses of Diagram Interchange feature to store information about UML elements presented in their profile, making of it a valuable asset.

UMLi is a particularly known notation to be used in GUI modelling. It shares a mutual feature as the UML profile for GUI Modelling: it is layout design oriented. Despite this fact, it is possible to model both structure and behaviour of a GUI. Nevertheless, the set of icons available in the custom palette to be used in the design process of a GUI, are not visually attractive and suffer from few limitations (e.g., they are only used for describing GUI layout and are not able to describe behaviour). In contrast, its icons are simple and intuitive and serve their purpose well. Another disadvantage of UMLi is the complicated task of modelling the behaviour of the GUI, even if it is a simple GUI. The more complex a GUI is, the more difficult it becomes to model its behaviour with UMLi. However, UMLi has tool support, which can be seen as an advantage, in order to widespread the notation.

The Wisdom profile, proposed by Nunes and Cunha, allows the specification and design of interactive systems. They also have taken advantage of UML extensions mechanisms to develop their profile. Regarding the dialogue model featured in this profile, all the dialogue structure of the application is supported on an UML based approach of the CTT notation, by expressing temporal relationships between tasks. As such, it seems possible to model the behaviour of a GUI. This approach appears to care more about the usability of a GUI.

With usiXML it is possible to develop user interfaces based on UIDL, and therefore to specify a user interface at and from different levels of abstraction, while maintaining the mapping between these levels. Due to its internal structure, this approach represents an excellent paradigm for GUI design. However, it is not possible to model any kind of behaviour of a user interface. The main audiences of this language are analysts, modellers, designers, and others. This language offers a significant amount of tool support.

Regarding the Canonical Abstract Prototypes (CAP) approach, it represents an extension to usage centred design that provides a formal vocabulary for expressing visual and interaction designs without concerns with details of appearance and behaviour. They offer a rich set of abstract components with easy selection from a palette in order to speed and simplify the process of prototyping. This approach is more related with usability issues and, as mentioned, it does not provide means to model the behaviour of a GUI. In addition, it is not formal enough for being used in a model-based testing process.

CTT is one of the most widely used notations for task modelling, specifically tailored for user interface model-based design. The CTT syntax is mainly graphical and therefore aims to better highlight the hierarchical structure of the system. In addition, it does also provide means to define GUI behaviour. Due to its well documented specification, it is simple to understand how to make use of CTT elements. CTT has tool support. Moreover, this notation is applied and extended in other approaches such as, the Wisdom Profile.

Finally, with the Spec# language it is possible to model the behaviour of a GUI, basically by writing down the methods which describe possible user actions upon the GUI under test. Spec# is a formal textual notation and testers and modellers prefer graphical ones.

The approach proposed in this research work, makes use of CTT, Canonical Abstract Components and Spec#. It wishes to benefit from the best of each approach, aiming also to model GUIs graphically which provides an environment that testers and modellers prefer. The approach proposed will be described in the following chapters.

Chapter 3

VAN4GUIM Overview

This chapter will describe the Visual Abstract Notation for GUI Modelling (VAN4GUIM). It was developed using Enterprise Architect's UML tool, since it represents one of the most powerful and flexible software design tool available. In addition, such tool is one of the best ways to leverage the power and scope of UML 2.1 for modelling scenarios. This chapter includes a description of the UML Profiles as well as their translation rules into Spec#. The VAN4GUIM notation is structured into five main UML profiles: Containers, User Actions, Hybrids, CTT Connectors and Window Manager. These profiles are intended to be used in a drag and drop way, in order to model the specific GUI into analysis.

3.1 VAN4GUIM UML Profiles

UML Profiles provide the possibility of extending the UML Language, allowing to build UML models for particular domains. Such profiles are based on additional stereotypes and tagged values that are applied to elements, attributes, methods and others. A Profile characterizes a collection of extensions that together are able to describe some particular modelling problem and facilitate modelling constructs in that specific domain. In summary, an UML Profile is a specification that is able to: identify a subset of an UML meta-model; specify a set of rules beyond those specified by the subset of the UML meta-model; detail semantics, expressed in natural language; and denote common model elements, expressed in terms of the profile.

The developed VAN4GUIM UML Profiles are based on two important concepts. The first leads us to Canonical Abstract Prototyping, a notation introduced by Larry Constantine [23]. The prototypes are an extension to usage-centred design that provide a formal vocabulary for expressing visual and interaction designs without concern with details of appearance and behaviour. The second concept, is supported on a commonly accepted and widely applied notation, ConcurTaskTrees (CTT) [18], which initial goal was to give support for designers of interactive systems. The CTT is a notation for task modelling being able to graphically represent a hierarchical structure, with a set of temporal operators capable to describe concurrent behaviour.

For the VAN4GUIM UML CTT Profile, we have extended these concepts by adding additional behaviour, functionality and restrictions to the CTT operators (Spec#). Regarding the remaining VAN4GUIM Profiles, we also extended the original concept of Canonical Abstract Components by including a set of rules, restrictions and suitable behaviour to them.

The current chapter gives an insight over the VAN4GUIM notation profiles. Each profile is composed by a set of stereotypes. Each stereotype is detailed with information, such as description, example of applicability, extended or generalized elements, restrictions, properties (properties have related get and set methods which will be omitted from the description) and methods.

3.1.1 Containers

Containers' profile is a subset of Canonical Abstract Components. The idea behind containers is their correspondence to content, information and data that act as a whole receptacle of UI objects (generally called *DataStores*). The latter is applicable to the Containers' profile of VAN4GUIM. An overview of the profile is illustrated in Figure 3.1. A description of each element from the profile is presented in Table 3.1.

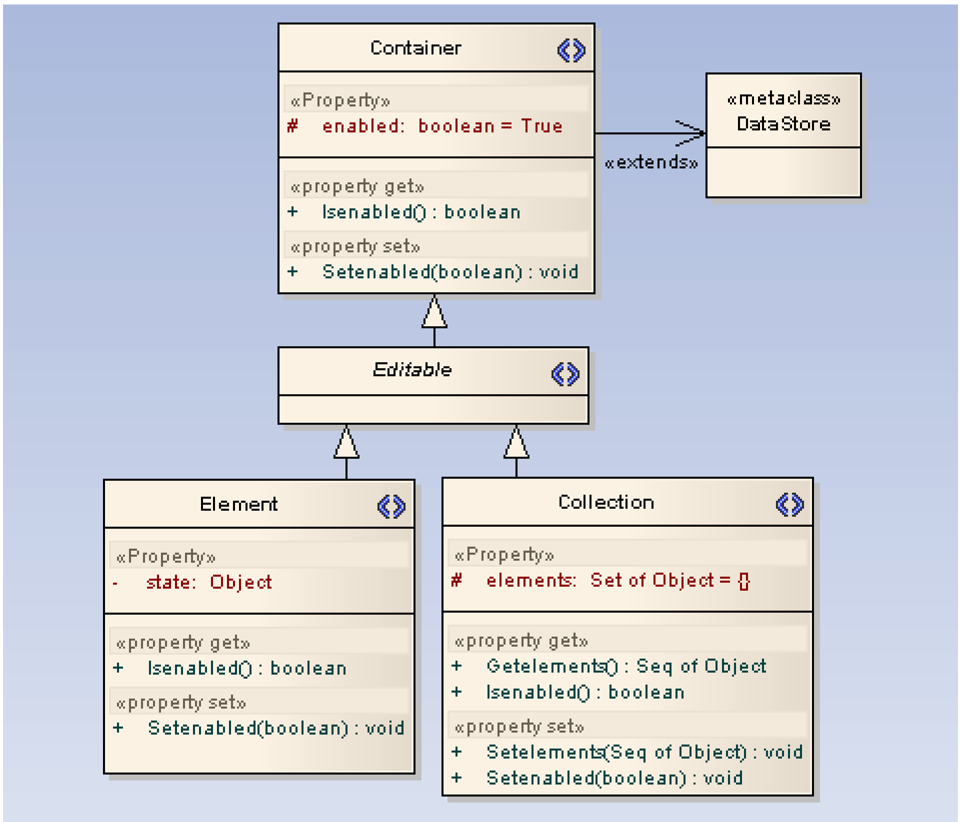


Figure 3.1 - VAN4GUIM Containers' Profile overview

Container	
Description	Represents a container of objects (UI elements) that can be seen as a holder.
Applicability	A particular area of a GUI can be seen as a holder, for instance the toolbars area, which is normally placed under the window bar. Elements like panels and canvas are seen as holders.
Extended Element	*DataStore*
Restrictions	Each method applied on Containers should have, at least, a pre-condition checking if the container where it is applied is enabled: • Pre-condition: enabled == true;

Properties	boolean enabled
Editable	
Description	Represents a generic stereotype that refers to one or more editable elements.
Applicability	NA (an abstract element cannot be instantiated)
Generalized Element	Container
Restrictions	NA
Properties	NA
Element	
Description	Represents an editable single element, which is placed inside a container. Each element has an internal state variable "state" that saves its state.
Applicability	To save the content of a string
Generalized Element	Editable
Restrictions	Inherited from Container
Properties	*object* state
Collection	
Description	Represents a set of elements that are placed inside a container.
Applicability	To save the content of a set of objects
Generalized Element	Editable
Restrictions	Inherited from Container
Properties	*Set<Object>* elements

Table 3.1 – Detailed description of VAN4GUIM Containers' Profile

3.1.2 User Actions

This profile is a subset of Canonical Abstract Components, where the user actions concept represents the actions, operators, mechanisms, or controls that can be applied to operate upon containers (from previous section and which are generally called *InteractionFunctions*). Some of the properties of each stereotype from this profile operate as restrictions for the VAN4GUIM notation. The basis of such behaviour is to create a dependency between **containers** and **user actions** profiles, making the notation more powerful, consistent and valuable. Figure 3.2 illustrates VAN4GUIM User Actions' Profile overview and a detailed description for each element of the profile is described in Table 3.2.

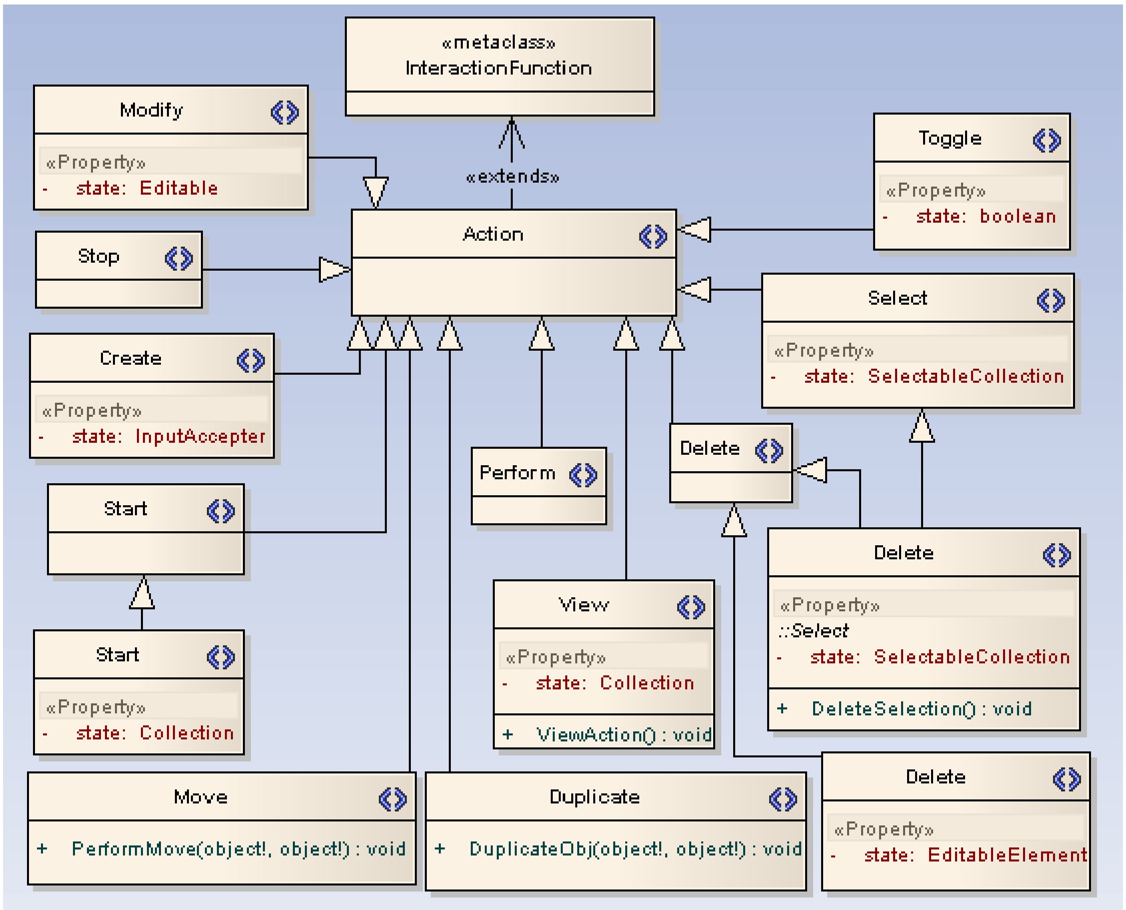


Figure 3.2 – VAN4GUIM User Actions' Profile overview

Action	
Description	Represents the concept inherent to an action. Naturally, all the stereotypes defined in this profile generalize an action.
Applicability	NA (abstract element)
Extended Element	*InteractionFunction*
Restrictions	Pre-condition: • The Container where it will act on should be enabled
Properties	NA
Start (A)	
Description	Represents an initial action state.
Applicability	When a system user clicks on a button, such action corresponds to the *Start* stereotype. Globally, this stereotype matches with a simple action that occurs in the system, which was triggered by the user. A simple action doesn't need of complex object's interaction to be triggered.
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	NA
Start (B)	

Description	Represents an initial action over of a set of objects.
Applicability	Selection of elements from a list.
Generalized Element	*Start (A)*
Restrictions	Inherited from *Action*
Properties	*Collection* state
Stop	
Description	Characterizes the interruption of an action/task.
Applicability	Press Cancel button on a GUI application
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	NA
Create	
Description	To create a new element/object/record .
Applicability	Creation of a new record in a database.
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	*InputAccepter* state
Modify	
Description	Represents the action of changing the state of an existing object.
Applicability	Activity of updating text in an application.
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	*Editable* state
Delete (A)	
Description	Represents a generic deletion of an object.
Applicability	NA (abstract)
Generalized Element	*Action*
Restrictions	NA
Properties	NA
Delete (B)	
Description	Symbolizes the removal or deletion of an editable element.
Applicability	Deletion of the content of an *editableElement.*
Generalized Element	*Delete (A)*
Restrictions	Inherited from *Action*
Properties	*EditableElement* state
Delete (C)	
Description	Represents the removal of a set of objects.
Applicability	Removal of a set of elements from a dropdown list or lines/columns from a table, etc.
Generalized Elements	*Delete (A), Select*
Restrictions	Inherited from *Action*
Properties	*SelectableCollection* state
Method	*DeleteSelection()*
Select	
Description	User action that acts as a selector of UI objects.

Applicability	Selection of an item from a dropdown list.
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	*SelectableCollection* state
View	
Description	A user action that allows selecting different view modes.
Applicability	See a document in its print layout form.
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	*Collection* state
Method	*void ViewAction()*
Duplicate	
Description	Creates a copy of an existing object.
Applicability	Copy to clipboard.
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	NA
Method	*DuplicateObj(object! source, object! target)*
Move	
Description	User action that allows moving an existing object to another location within the UI.
Applicability	Drag and drop of an UI element.
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	NA
Methods	*PerformMove(object! source, object! target)*
Toggle	
Description	Represents a change between two states (On/Off or Enable/Disable) of an UI element.
Applicability	Switch on/off of an UI element (for instance, bold button).
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	*boolean* state
Perform	
Description	User action that sets configuration information.
Applicability	Set print layout information
Generalized Element	*Action*
Restrictions	Inherited from *Action*
Properties	NA

Table 3.2 – Detailed description of VAN4GUIM User Actions' Profile

3.1.3 Hybrids

The concept behind a **hybrid** in the context of Canonical Abstract Components results in a combination between **containers** (*DataStores*) and **user actions** (*InteractionFunctions)*. It models at the same time the context and behaviour inherent to user actions that take place over specific containers. Figure 3.3 illustrates VAN4GUIM Hybrids' Profile overview and a detailed description for each element of the profile is described in Table 3.3.

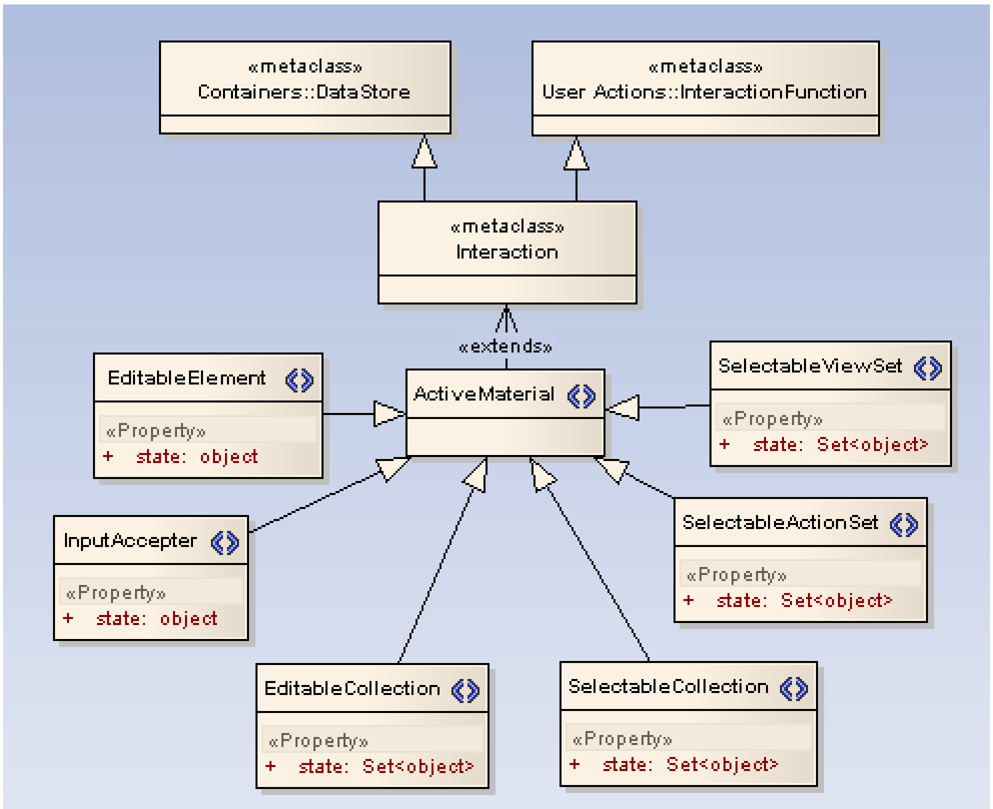


Figure 3.3 – VAN4GUIM Hybrids' Profile overview

Interaction	
Description	Metaclass that holds information regarding both *user actions* and *containers* contexts.
Applicability	NA (metaclass)
Generalized Elements	*DataStore, InteractionFunction*
Restrictions	NA
Properties	NA
ActiveMaterial	
Description	Represents the concept of an action on a container.
Applicability	NA (abstract)
Extended Element	*Interaction*
Restrictions	NA
Properties	NA
InputAccepter	

Description	Hybrid responsible for accepting data input from the user, in a specific delimited area for that purpose.
Applicability	Filling in a textbox.
Generalized Element	*ActiveMaterial*
Restrictions	-
Properties	*object* state
EditableElement	
Description	Represents an editable element.
Applicability	A text box allowing to edit, delete, copy and past text.
Generalized Element	*ActiveMaterial*
Restrictions	-
Properties	*object* state
EditableCollection	
Description	Represents a set of editable elements.
Applicability	A list box where it is possible to update items and add new ones.
Generalized Element	*ActiveMaterial*
Restrictions	-
Properties	*Set<object>* state
SelectableCollection	
Description	Represents a particular set of elements, where each element is selectable.
Applicability	Selecte of an element from a dropdown list.
Generalized Element	*ActiveMaterial*
Restrictions	-
Properties	*Set<object>* state
SelectableActionSet	
Description	Refers to a selectable set of elements that when selected trigger an action automatically.
Applicability	Select a different document layout (e.g., Normal view, Print layout, etc.)
Generalized Element	*ActiveMaterial*
Restrictions	-
Properties	*Set<object>* state
SelectableViewSet	
Description	Refers to a selectable set of elements that change visualization properties of GUI objects.
Applicability	Update screen resolution.
Generalized Element	*ActiveMaterial*
Restrictions	-
Properties	*Set<object>* state

Table 3.3 - Detailed description of VAN4GUIM Hybrids' Profile

3.1.4 CTT Connectors

VAN4GUIM Connectors are based on the temporal items of the ConcurTaskTrees notation. As such, it was decided to apply this concept to the VAN4GUIM notation in a form of UML connectors. With this unique approach, it became possible to benefit of a graphical representation of relationships between VAN4GUIM elements. Thus, it has also allowed adding specific behaviour to each connector, since it is possible to add restrictions, as pre and post conditions, to them. To provide expansibility to the notation, such restrictions aren't placed directly on the profile, but only after instantiating the desired connector. An exception is the *EnablingWithInfoExchange* connector which transfers information between its source and its target and, at the same time, sets the *enabled* property of the target to true. This is described by the following post-condition *source.enabled == true*. Finally, each connector has the same meaning as the CTT's temporal operators. For this reason, the current section won't go into further details regarding the connector's descriptions, since they were already explained in chapter 2.4 (CTT notation). A screenshot of the CTT Connector's profile is displayed below.

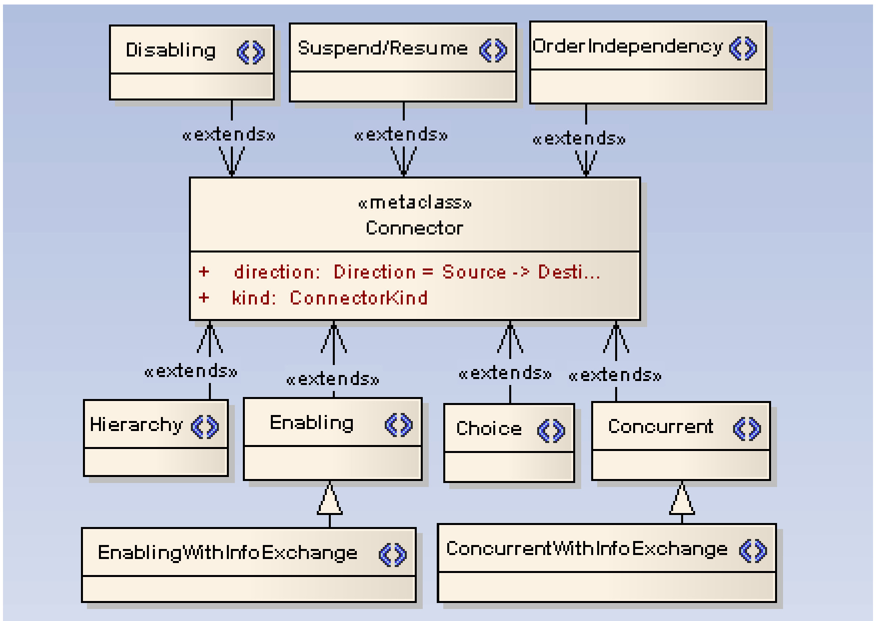


Figure 3.4 – VAN4GUIM CTT Connectors' Profile overview

As Figure 3.4 suggests, all of the CTT temporal operators are described in the profile, and all extend the *Connector* meta-class. This meta-class has two public properties: direction and kind. The *former* property indicates the source and destination of the connector, while the latter property indicates the type associated with it.

3.1.5 Window Manager

The Window Manager profile was shaped to manage all the windows' behaviour of an application. The window manager is a part of the model, and its state is part of the model state. Whenever a window is opened or closed, it becomes registered/unregistered in the window manager. However, if a window is removed, its children are also removed from the manager. Regarding message boxes, these are also registered in the window manager. Only two types of messages boxes are taken into account:, namely, acknowledge and query message boxes. The exposed concepts will be discussed in the sequel. Figure 3.5 illustrates VAN4GUIM User Actions' Profile overview and a detailed description for each element of the profile is described in Table 3.4.

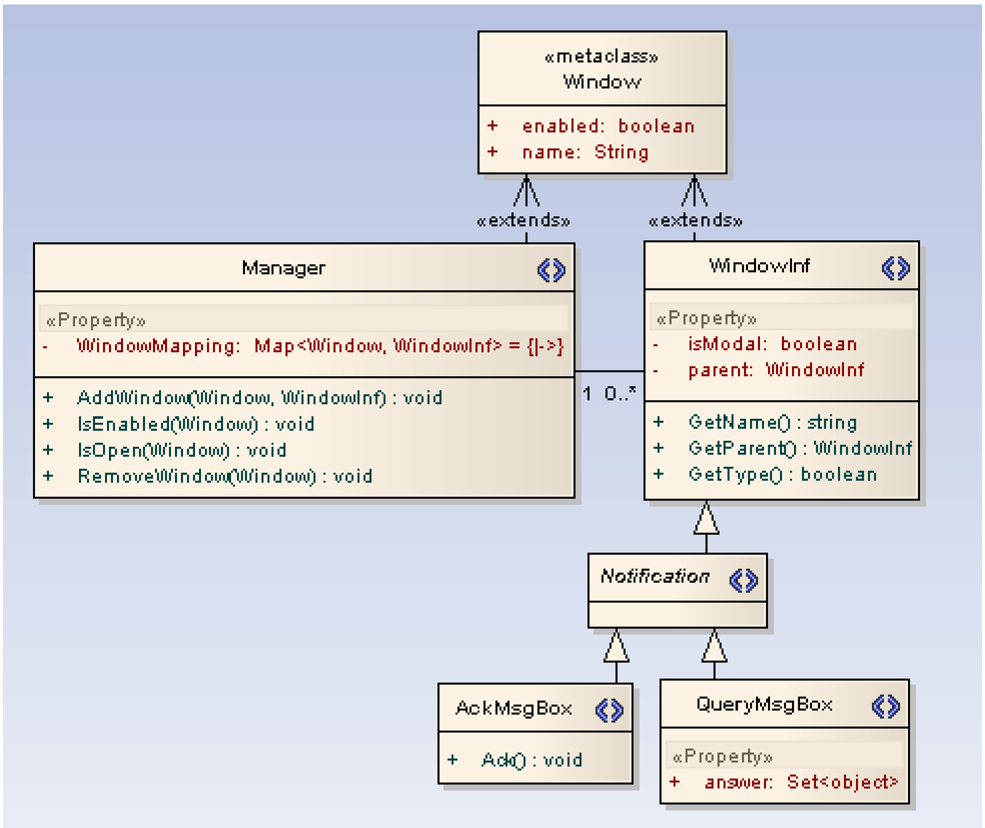


Figure 3.5 – VAN4GUIM Window Manager's Profile overview

Window	
Description	Meta-class that represents the concept of a window. A window is comprised by a name and a property to inform/check whenever the window is whether enabled or not.
Applicability	NA
Extended Element	NA
Restrictions	NA
Properties	*string* name; *boolean* enabled
Manager	

Description	Element responsible for the management of all the window mapping that occurs within an application.
Applicability	NA (The user is not aware of this concept, since it is intended to be internal).
Extended Element	*Window*
Restrictions	NA
Properties	*Map<Window, WindowInf> WindowMapping*
Methods	*void AddWindow(Window wnd, WindowInf wi);* *void RemoveWindow(Window wdn);* *IsOpen(Window wdn);* *IsEnabled(Window wdn);*
WindowInf	
Description	Stereotype that represents a window. It also states whether it is modal or not, as well as its parent.
Applicability	Allows to model the windows of a GUI application.
Extended Element	*Window*
Restrictions	NA
Properties	*boolean* isModal; *WindowInf* parent
Methods	*boolean* GetType(); *WindowInf* GetParent(); *string* GetName()
Notification	
Description	Represents a generic message window that is displayed by the system to its user upon a particular event occurs.
Applicability	NA (abstract)
Generalized Element	*WindowInf*
Properties	Inherited from *Window*, *WindowInf*
AckMsgBox	
Description	Modal windows that aim to inform the user of a particular situation. The user is allowed to proceed is work only after pressing the, usually, OK button.
Applicability	Give information to the user.
Generalized Element	*Notification*
Properties	Inherited from *Window*, *WindowInf*
Method	*void* Ack()
QueryMsgBox	
Description	Modal window presenting information to the user. Commonly, these Message Boxes present a question to the user and a set of buttons which correspond to possible answers to such question.
Applicability	Give information to the user and require user interaction answering some question
Generalized Element	*Notification*
Properties	Inherited from *Window*, *WindowInf*

Table 3.4 - Detailed description of VAN4GUIM Window Manager’s Profile

3.2 Overall testing process

The model-based GUI testing process starts with the construction of the VAN4GUIM visual model, based on the new innovative VAN4GUIM profiles. Figure below summarizes all the activities and artefacts generated in the model-based GUI testing proposed. It defines twelve steps and produces ten artefacts. Icons displayed in the bottom left activities (in blue colour) represent either a manual (hand icon) or an automatic task (mechanism icon). The approach proposed in this dissertation generates two new original concepts in the testing process, which appear with the "new" icon on the bottom right of both activities and artefacts.

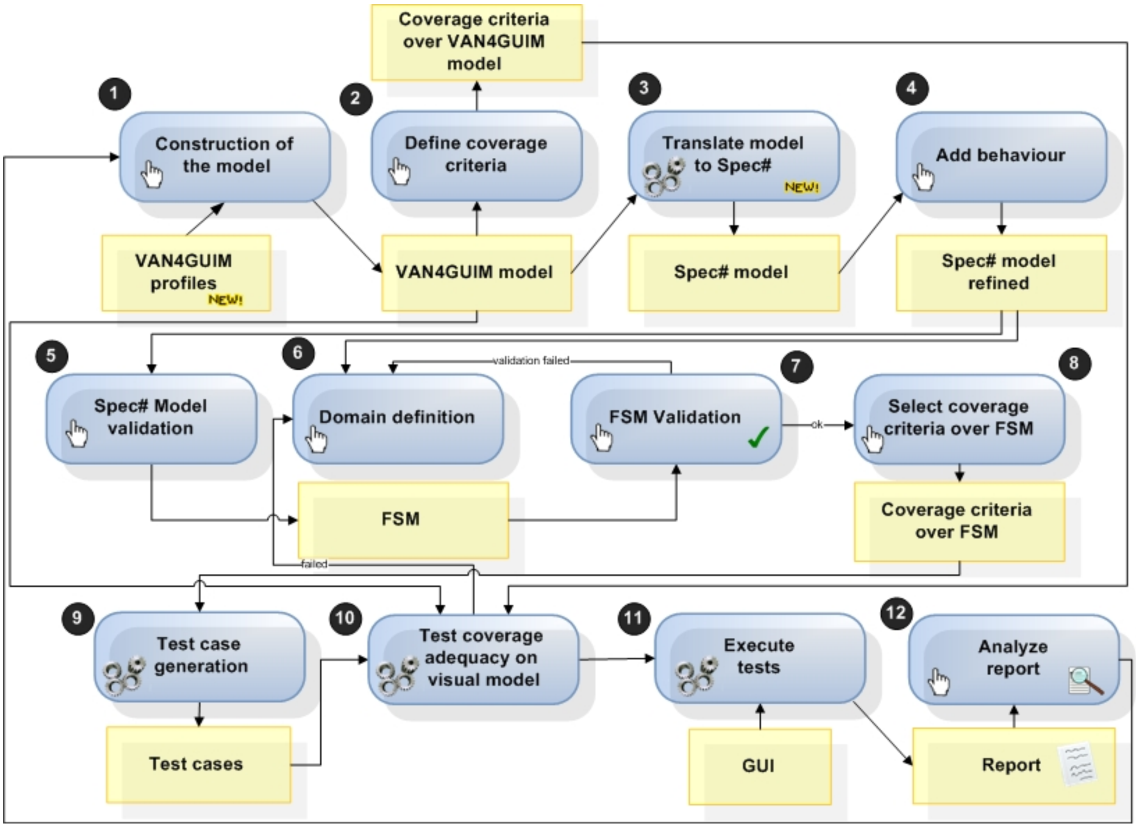


Figure 3.6 – View of the overall GUI testing process proposed

The testing process illustrated above involves the following activities:

1. **Construction of the model** – Definition of the visual model based on the new VAN4GUIM profiles. This step must be performed manually.

2. **Define coverage criteria** – Manual definition of coverage criteria over the specified visual model. This activity intends to assure that test cases, generated later, cover totally the visual model (further details are presented in section 3.4).

3. **Translate model to Spec#** – Translation to Spec# of the specified visual model. This step is accomplished by means of a software tool, which was developed during this dissertation. Its purpose is to automatically translate the visual model, previously exported to XMI, to Spec# (producing a formal model). This translation fulfils a set of rules that are defined in section 3.5.

4. **Add behaviour** – Manual refinement of the formal model by completing, for instance, some of the method bodies that were generated from the previous step. These explicit specifications should be consistent with the initial (implicit) specifications, conforming to post-conditions.

5. **Spec# model validation** – With the Spec Explorer tool, the Spec# model refined artefact is checked to guarantee its correctness. If true, it will generate a FSM. This task must be performed manually.

6. **Domain definition** – The definition of domain values is a step that must be performed manually and represents a crucial point in the testing process. Domains have an impact in the FSM generated by exploration of the model. Domain definition is an iterative process involving the need to verify full coverage of functional dependencies (further details can be seen in section 3.6). A FSM is generated by exploration of the model via Spec Explorer.

7. **FSM validation** – The Spec Explorer tool generates automatically a FSM by exploring the model within defined bounds. By default, all the states of the model that are reachable within such bounds will be explored and represented in the FSM. As such, in this step the user must check if all the states of the model are reachable. If so, the process can advance to the next step; but if not, the process will have to back go back to step number 5 (domain definition).

8. **Select coverage criteria over FSM** – Manual selection of coverage criteria over the FSM (for further details see section 3.6). This step helps in defining when to stop testing and how to evaluate the quality of the test suite.

9. **Test case generation** – Test cases are generated automatically by Spec Explorer, from the FSM model after selecting FSM coverage criteria (for further details see section 3.6).

10. **Test coverage adequacy on visual model** – Activity responsible of discovering the test coverage percentage on the visual model. If such coverage is found not to be enough or adequate, the current activity will lead to step number 6. Otherwise, it will go ahead to the next step.

11. **Execute tests** – Once generated, test cases are executed automatically on the specification and on the implementation, and the results obtained are compared. The specification plays the role of a test oracle describing the expected results [3].

12. **Analyze report** – Finally, the generated report (containing a description of all inconsistencies) is analyzed by the user. The inconsistencies should be fixed and after that the process may go through another iteration.

3.3 Structure of the GUI model

A GUI model constructed by using VAN4GUIM will have at least two levels. The first comprises a navigation map on which multiple instances of *WindowInf* stereotype (typically window) from the profile Window Manager (described in 3.1.5) occur, as well as transitions between them. These transitions are visually represented as CTT connectors (described in 3.1.4) and comply with a set of rules composed of pre and post conditions specified in Spec#. The goal is to model transitions between windows and, by doing so, it is necessary to specify a set of rules that must be fulfilled in order to make possible to occur a transition between those windows. In addition, a window may be composed by three types of profiles that VAN4GUIM offers: user actions, hybrids and containers. This composition within a window represents the second level of the visual notation. In this level, the transitions between the various instances of the profiles mentioned before are also modelled via CTT connectors with restrictions.

However, there may be software applications with more complicated GUIs that would require more levels. The VAN4GUIM notation allows multiple levels to be specified, since it when it was built it had expansibility in mind. The following pictures illustrate an example of the mentioned first and second levels, respectively.

3.4 Coverage criteria over the visual GUI model

Due to determine and evaluate the quality of the generated tests, some coverage criteria were defined over the VAN4GUIM model.

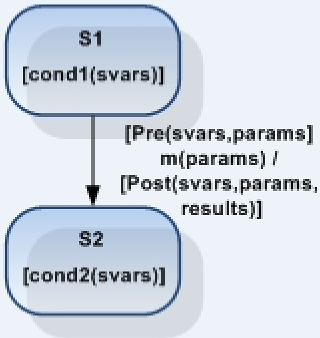
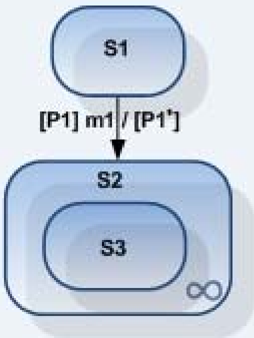
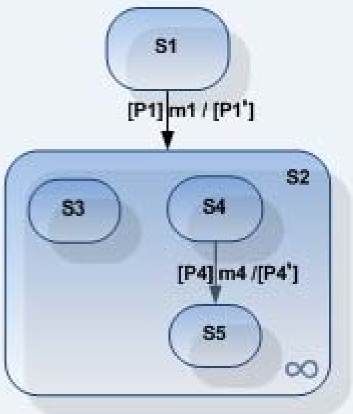
The coverage criteria over the visual GUI model have been set to:

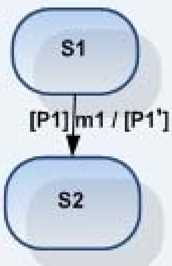
- ✓ Full coverage of navigation map where typically states are instances of window manager stereotypes and transitions are CTT connectors;
- ✓ Full coverage of the lower levels of the visual GUI model in which states are typically containers, user actions and hybrids, and transitions are CTT connectors.

One way to evaluate coverage criteria percentage for each of these criteria will be achieved by specifying views in Spec Explorer tool. This process will be explained in section 3.6.

3.5 VAN4GUIM to Spec# Translation Rules

After creating the new visual notation (a set of UML Profiles), a set of rules were built to translate the VAN4GUIM notation to the corresponding Spec#, using pre and post-conditions that trigger the state machine transitions. The translation rules can be seen in the Table 3.1. Whenever a state displays an ∞ icon, it has the purpose of representing a composite state. Composites states are only referred to windows, which in VAN4GUIM notation represent instances of the stereotyped element *WindowInf.* After defining rule number one, in order to simplify the state machines and expressions, it is assumed that ***Si*** is a condition over state variables in state ***i***; whenever states represent windows, ***Ni*** is the name of the window ***i*** and ***[Pi]mi/[Pi']*** are transitions between states in which ***[Pi]*** is a pre-condition over state variables and parameters, ***mi*** is a function with (omitted) parameters and ***[Pi']*** is a post-condition over state variables, parameters and result of the executed function.

Rule	UML Protocol State Machine	Translation to Spec#
R1 - Simple Transition S1 and S2 represent an instance of a stereotype available from any of the profiles		[**Action**] m(params) **requires** cond1(svars) && pre(svars,params); **ensures** post(svars,params,results) && cond2(svars); { //TODO } ("svars" – state variables)
R2 – Transition to a composite state S1 and S2 represent *WindowInf* instances of the stereotype defined in window manager profile; S3 represents an instance of any stereotype (except *WindowInf*) available from any profile		[**Action**] m1 **requires** S1 && [P1]; **ensures** [P1'] && S2 && S3; { //TODO }
R3 – Transition to a composite state with two or more initial states S1 and S2 represent instances of the *WindowInf* stereotype from window manager profile; S3, S4 and S5 represent instances of the stereotypes (except *WindowInf*) available from any profile		[**Action**] m1 **requires** S1 && [P1]; **ensures** [P1'] && S2 && S3 && S4; { //TODO } [**Action**] m4 **requires** S4 && [P4] **ensures** [P4'] && S5 { //TODO }

R4 – Transition to an acknowledge message box state
 	
S1 represents an instance of a stereotype (except *WindowInf*) available from any profile;
 	
S2 represents an instance of *AckMsgBox* stereotype (modal window) from
		[**Action**] m1
		requires S1 && [P1]
		ensures [P1'] && S2 &&
		IsOpen(N2);
		{
		AddWindow(S2, S1, true);
		//TODO
		}

WindowManager profile		[**Action**] Ok
		requires IsOpen(N2)
		ensures [P1’] && ! IsOpen(N2);
		{
		RemoveWindow(N2);
		
		//TODO
		}

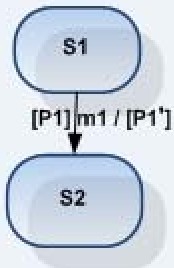
R5 – Transition to a query message box state
 	
S1 represents an instance of a stereotype (except *WindowInf*) available from any profile;
 	
S2 represents an instance of *WindowAlert* stereotype (modal window) available from the *WindowManager* profile
		[**Action**] m1
		requires S1 && [P1];
		ensures [P1’] && S2 && IsOpen(N2);
		{
		AddWindow(N2, S1, true);
		
		//TODO
		}
		
		[**Action**] ChooseOption(string op)
		requires IsOpen(N2) && S2
		{ // TODO }

Table 3.1 – Display of translation rules from VAN4GUIM notation into Spec#. State and pre/post-conditions are abbreviated after rule R1.

3.6 Model-based testing with Spec Explorer

Spec Explorer is a software development tool for advanced model-based specification and conformance testing developed by Microsoft Research. It aims to help detect errors in the design, specification and implementation of software systems. Spec Explorer presents the possibility to encode system’s intended behaviour in machine-executable form and to explore the possible runs of the specification-program as a way to systematically generate test suites. Furthermore, Spec Explorer has support for test case generation, facilities to establish maps between specification actions and implementation methods, support for test cases execution, and conformity evaluation [6].

State machines are appropriate to model reactive systems such as GUIs, since they can act in response to user actions. With a state machine, it is possible to describe a set of states and corresponding transitions between them that are triggered by actions. Such state machines are well suited to guide the testing of software applications. A specification that is written in Spec# is machine-executable. Since GUIs can be modelled using Spec#, the execution model of Spec# is based on the formalism of abstract state machines, and allows the specification to be used as a test oracle.

After defining the model of a GUI, a FSM can be generated by bounded exploration using Spec Explorer. Such FSM consists of states of the model program and corresponding method invocations that move from state to state as transitions. To explore the model by calling each of the actions available at each state, it is necessary to define the domains of the actions' parameters. If the set of possible values of that parameter is undersized, the general rule is to define the domain based on that set. These domains must allow for full coverage of the functional dependencies and full coverage of test boundary and special conditions, and coverage of the visual model in VAN4GUIM.

Since Spec Explorer does not grant support for the definition of the domains of the parameter actions, such task has to be accomplished manually by the user. This task represents a crucial point in the testing process, given the fact that domains have deep impact in the generated FSM, obtained by exploration of the model.

Formal specifications can be exploited to automatically generate test sequences, such as sequences of user actions and action parameters. The definition of the test generation is done within two phases [3]: first, a finite state machine (FSM) is generated from the specification, by exploring all the states that can be reached from a given initial state or set of initial states (each state is a possible combination of values of the state variables, and each transition corresponds to a user action with actual parameters); secondly, a test suite, comprising one or more test sequences, is generated from the FSM, so that all states and transitions are covered.

In addition there is also the possibility to express coverage criteria over the generated FSM, such as [3]:

- ✓ Full coverage of functional dependencies (check if the chosen domains allow showing all variables affect independently the behaviour of the system);
- ✓ Full coverage of the test boundary and special conditions (check if the FSM contains the states or sequences of states that describe boundary and special conditions).

The selection of domain values must be chosen carefully and wisely in order to cover all states and transitions. A strategy/approach to find the appropriate domain values can be found in [3].

In addition to the definition of domain values there are also other techniques that may be used in order to set boundaries to the exploration process. All these techniques have the purpose to generate a FSM with manageable size:

- **State filtering** – this technique is based on Boolean expressions to be defined by the tester. State filter exclude from the exploration process all states where the specified state conditions do not hold.
- **Additional pre-conditions** – extra pre-conditions are defined in order to limit the applicability of actions [3].

- **State groups** – the definition of state groups represents a technique that allows limiting the exploration of the model using Spec Explorer tool. Generally, this technique is performed in two steps: firstly, the state groups' expression is defined; secondly, a bound is defined for the exploration setting the number of states one wants to cover in each state group. Furthermore, state groups can also be used to define different views of the model. This feature can be helpful to define different levels of abstraction of the same model and also to notice some specific features of a huge model that otherwise could not be analyzed. For instance, in this approach, states groups are used to verify coverage criteria over the visual model as well as coverage criteria of high level properties.

- **On-the-fly** – this technique combines test derivation from a model and test execution [3] into a single algorithm. This solves non-determinism by getting immediate feedback from the implementation and avoiding the pre-computation of the possible huge test case with all possible responses of the system under test.

It is feasible to generate test cases from the obtained FSM, and thus, such generation is made automatically. Several coverage criteria over the FSM can be applied, for instance: full transition coverage; shortest path; and random walk. Regarding full transition coverage, it aims to check if the generated test suite covers all of the transitions of the FSM. As for the shortest path criterion, the generated test suite is the shortest path that reaches a specified goal state. Finally, the random walk criterion generates a test suite with a single sequence of invocations. At each state, one of the outgoing transitions is randomly selected.

The execution detects errors whenever:

- Probe actions return different values, both in the model and implementation;
- One tries to interact with a window which is not opened or is not enabled;
- One tries to interact with a control which is not available.

Chapter 4

Case Study

The construction of a new visual notation (VAN4GUIM) and a tool to automate the translation from a visual model – based on VAN4GUIM – to Spec# represented a new approach in GUI testing process. To prove its correctness and usefulness, it is essential to simulate a new process introducing such approach. The current chapter focuses on a particular case study, the Microsoft Notepad application.

4.1 Notepad

Windows' Notepad is a simple text editor that has been incorporated within all versions of Microsoft Windows since Windows 1.0 in 1985 [24]. Notepad is a plain text editor, with a simple GUI, that can be used to create, edit and view single text files. An example of its GUI is displayed below.

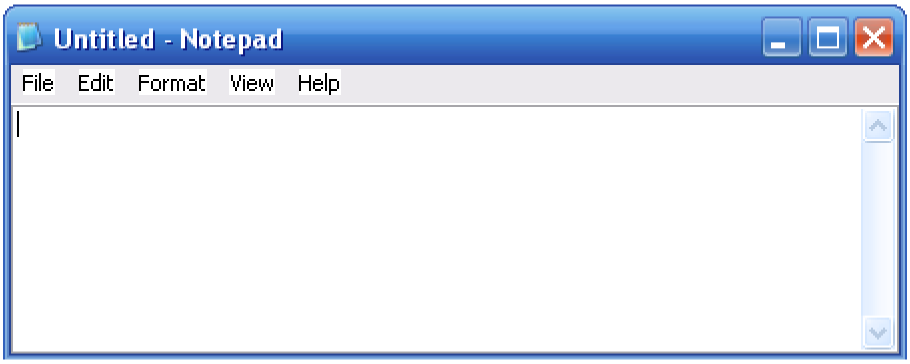


Figure 4.1 – Notepad main window

The notepad application is used along the current chapter as a case study in order to validate and demonstrate the proposed specification (VAN4GUIM).

4.2 VAN4GUIM Notepad Specification

The navigation map concept represents one of the views that result from the structure of the GUIs. A navigation map describes how to open and close windows of the application and also how to switch between the windows of the same application. Figure 4.2 illustrates the navigation map of the Microsoft Windows Notepad application, based on the VAN4GUIM notation.

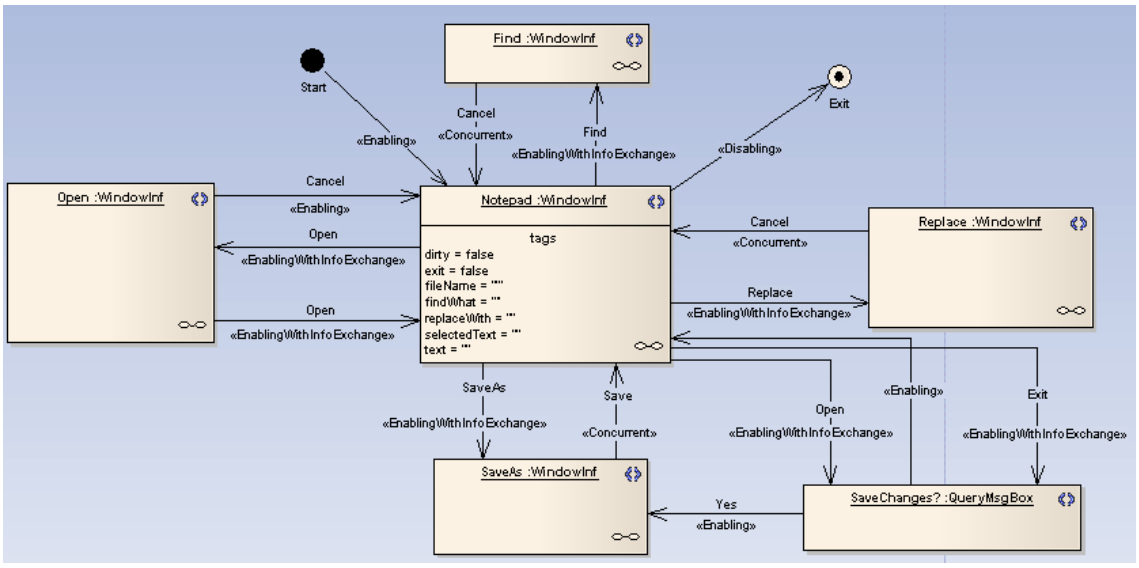


Fig. 4.2 – Navigation map for the Notepad application

The navigation map defines the following substance:

- A set of windows which are instances from the window manager profile from the VAN4GUIM notation;
- A set of rules shaped in the form of pre and post conditions (written in Spec#), based on the CTT connectors profile from the VAN4GUIM notation;
- The first level of abstraction of the GUI model;

Figure 4.2 is the first level of abstraction of the GUI model, the navigation map. The second levels of the GUI model are detailed information of the elements (from the first level) with an ∞ icon on the bottom right corner.

A description of each element from the navigation map will be presented next. In the first level, six windows are modelled as instances of *WindowInf* stereotype from the *WindowManager* profile. The modelled windows are Find, Open, Notepad (shell), Replace, SaveAs and SaveChanges. Each level is comprised by a set of elements, where each element is defined by its name, instance, composite status, tags and connectors. All the connectors are characterized by their source and destination elements, name, stereotype, and pre and post conditions. All of the information regarding the first level of abstraction appears compiled in Table 4.1. Figure 4.3 to Figure 4.6 illustrate all the elements from the second level of abstraction. Table 4.2 to Table 4.6 detail all the elements of such level.

First Abstraction Level

First Abstraction Level			
Name	**Instance of**	**Composite**	**Tags**
Notepad	*WindowInf*	True	exit = false; fileName = “”; findWhat = ””; text = “”; dirty = false; replaceWith= “”;
Find	*WindowInf*	True	-
Replace	*WindowInf*	True	-

Open	*WindowInf*	True	-		
SaveAs	*WindowInf*	True	-		
SaveChanges?	*QueryMsgBox*	False	-		
Connectors					
Source	**Destination**	**Name**	**Stereotype**	**Pre**	**Post**
Notepad	Replace	Replace	*EWIE*(*)	IsEnabled(“Replace”) && !IsOpen("Find")	target.findWhat == source.findWhat && target.replaceWith == source.replaceWith && IsEnabled(“Replace”)
Notepad	Find	Find	*EWIE*(*)	IsEnabled("Notepad") && MyNotepad.text != “” && !IsOpen("Replace")	target.findWhat == source.findWhat && IsEnabled(“Find”)
Notepad	Open	Open	*EWIE*(*)	IsEnabled("Notepad") && !dirty	target.fileName == source.fileName && IsEnabled(“Open”)
Notepad	SaveAs	SaveAs	*EWIE*(*)	IsEnabled("Notepad")	target.fileName == source.fileName
Notepad	SaveChanges?	Open	*EWIE*(*)	IsEnabled("Notepad") && dirty	IsEnabled(“SaveChanges?”)
Notepad	SaveChanges?	Exit	*EWIE*(*)	IsEnabled("Notepad") && source.exit	IsEnabled(“SaveChanges?”)
SaveChanges?	SaveAs	Yes	*Enabling*	IsEnabled("SaveChanges") && fileName == ""	!IsOpen("SaveChanges")

Table 4.1 – First level of notepad application’s navigation map. (*) the EWIE word refers to the element named *EnablingWithInfoExchange* from VAN4GUIM CTT connectors’ profile.

Second Abstraction Level

Find

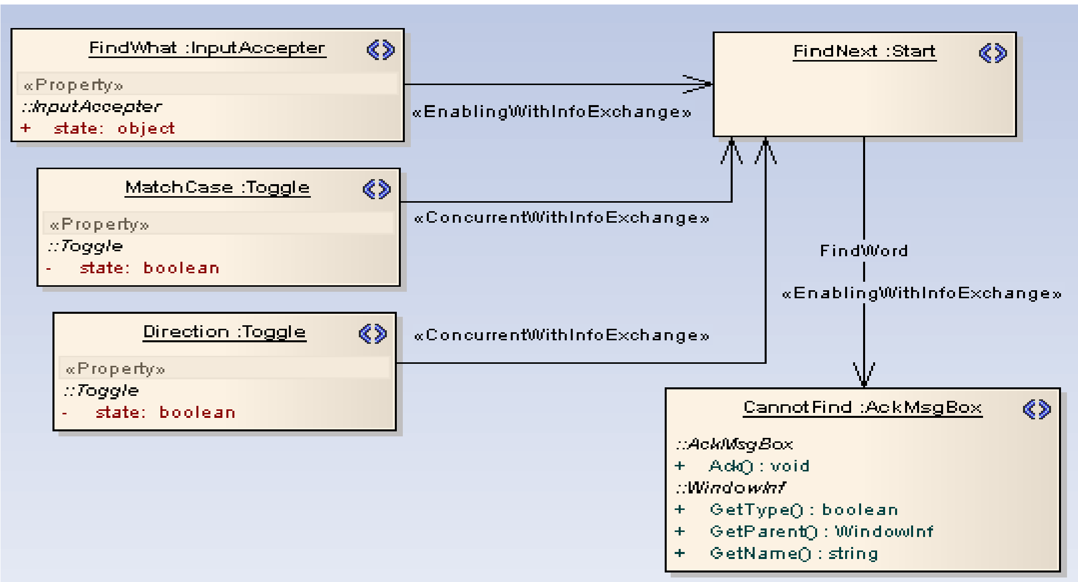


Figure 4.3 – Second abstraction level for the *Find* functionality

Second Abstraction Level (Find)			
Name	**Instance of**	**Composite**	**Property**
FindWhat	*InputAccepter*	False	object state
MatchCase	*Toggle*	False	boolean state
Direction	*Toggle*	False	boolean state
FindNext	*Start*	False	-
CannotFind	*AckMsgBox*	False	-

Connectors					
Source	**Destination**	**Name**	**Stereotype**	**Pre**	**Post**
FindNext	CannotFind	FindWord	*EWIE*(*)	IsEnabled("Find") && findWhat!= ""	!MyNotepad.FindWord(findWhat, matchCase, direction) => IsEnabled("CannotFind")
Find	Notepad	Cancel	*Concurrent*	IsEnabled("Find")	!IsOpen("Find")

Table 4.2 – Second level of the notepad application's navigation map regarding the *Find* functionality. (*) the EWIE word refers to the element named *EnablingWithInfoExchange* from VAN4GUIM CTT connectors' profile.

Replace

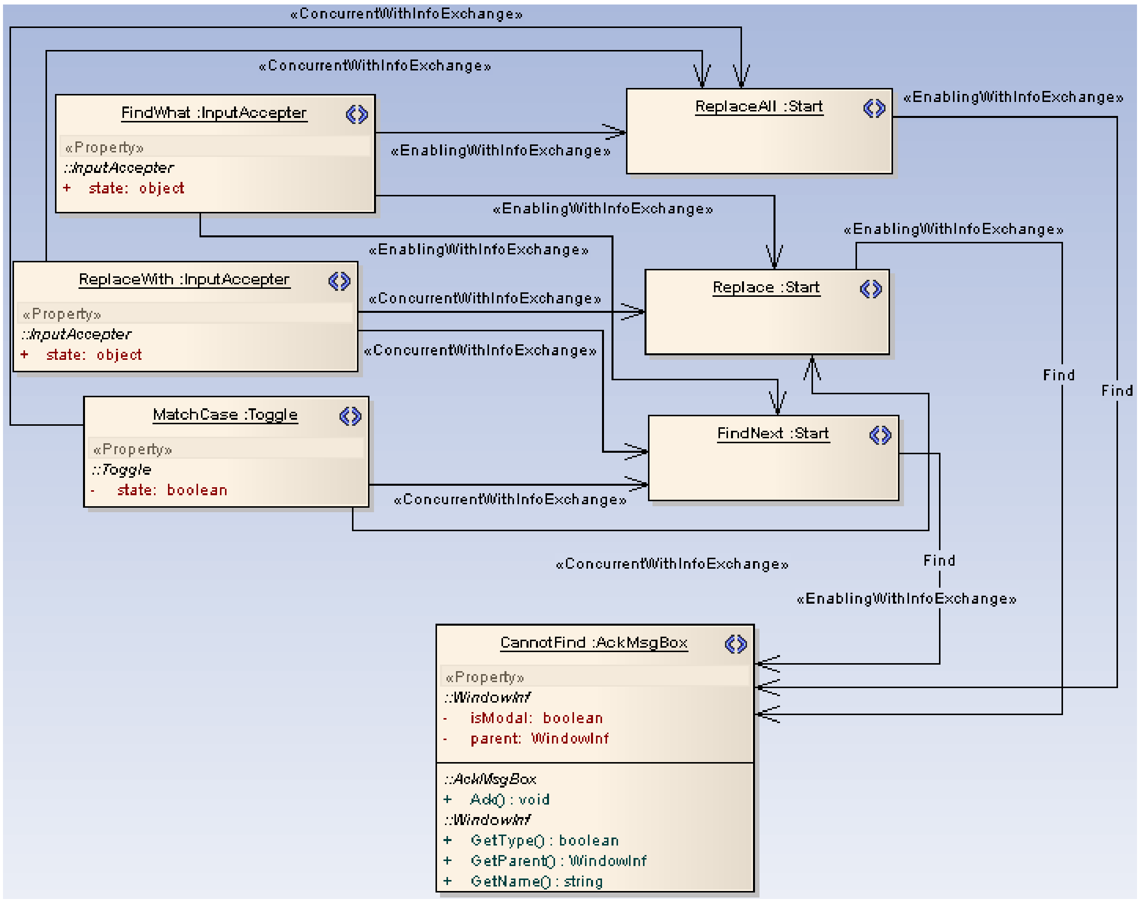


Figure 4.4 – Second abstraction level for the *Replace* functionality.

Second Abstraction Level (Replace)			
Name	**Instance of**	**Composite**	**Property**
FindWhat	*InputAccepter*	False	object state
ReplaceWith	*InputAccepter*	False	object state
MatchCase	*Toggle*	False	boolean state
FindNext	*Start*	False	-
Replace	*Start*	False	-
ReplaceAll	*Start*	False	-
CannotFind	*AckMsgBox*	False	-

Connectors					
Source	**Destination**	**Name**	**Stereotype**	**Pre**	**Post**
Replace	Notepad	Cancel	*Concurrent*	IsEnabled("Replace")	!IsOpen("Replace")
ReplaceAll	CannotFind	Find	*EWIE*	!MyNotepad.FindWord(findWhat, matchCaseState) && findWhat != null	IsEnabled("CannotFind")
FindNext	CannotFind	Find	*EWIE*	!MyNotepad.FindWord(findWhat, matchCaseState) && findWhat != null	IsEnabled("CannotFind")
Replace	CannotFind	Find	*EWIE*	!MyNotepad.FindWord(findWhat, matchCaseState) && findWhat != null	IsEnabled("CannotFind")

Table 4.3 - Second level of the notepad application's navigation map regarding the *Replace* functionality.

SaveAs

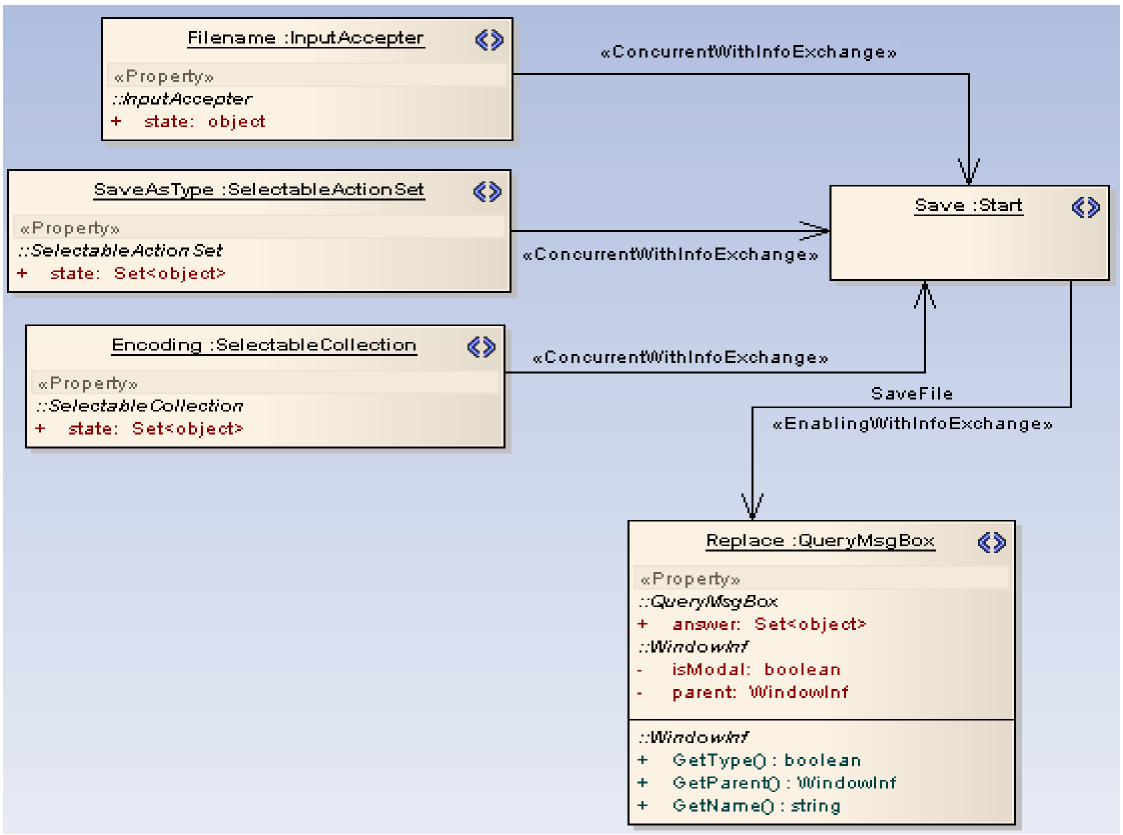


Figure 4.5 – Second abstraction level for the *SaveAs* functionality

Second Abstraction Level (SaveAs)					
Name	**Instance of**	**Composite**	**Property**		
Filename	*InputAccepter*	False	object state		
SaveAsType	*SelectableCollection*	False	*Set<object>* state		
Encoding	*SelectableCollection*	False	*Set<object>* state		
Save	*Start*	False	-		
Replace	*QueryMsgBox*	False	Set<object> answer		
Connectors					
Source	**Destination**	**Name**	**Stereotype**	**Pre**	**Post**
Save	Replace	SaveFile	*EWIE*	IsEnabled(“SaveAs”) && MyNotepad.Exists(filename)	IsOpen("Replace")
SaveAs	Notepad	Save	*EWIE*	IsEnabled("SaveAs")	MyNotepad.fileName == source.fileName && !IsOpen("SaveAs")

Table 4.4 - Second level of the notepad application’s navigation map regarding the *Save As* functionality.

Open

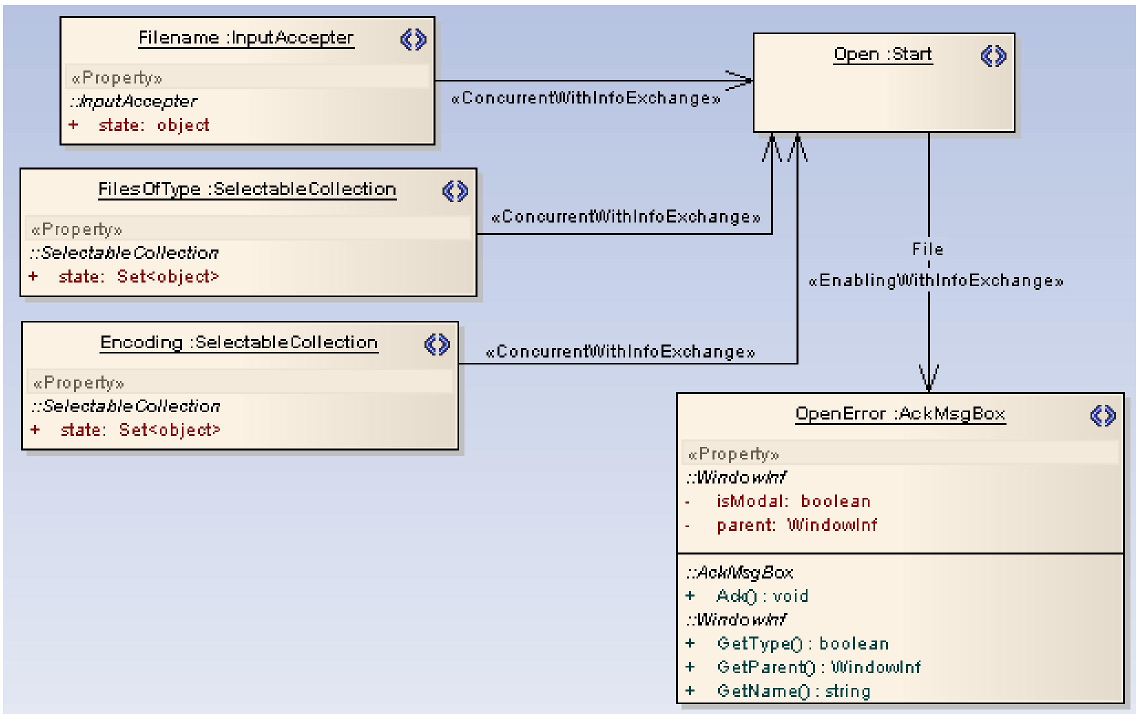


Figure 4.6 – Second abstraction level for the *Open* functionality

Second Abstraction Level (Open)			
Name	**Instance of**	**Composite**	**Property**
Filename	*InputAccepter*	False	object state
FilesOfType	*SelectableCollection*	False	*Set<object>* state
Encoding	*SelectableCollection*	False	*Set<object>* state
Open	*Start*	False	-
OpenError	*AckMsgBox*	False	-

Connectors					
Source	**Destination**	**Name**	**Stereotype**	**Pre**	**Post**
Open	OpenError	File	*EWIE*	IsEnabled("Open") && MyNotepad.Exists(filename)	IsOpen("Open")
Open	Notepad	Cancel	*Enabling*	IsEnabled("Open")	!IsOpen("Open")
Open	Notepad	Open	*EWIE*(*)	IsEnabled("Open")	MyNotepad.Exists(filename) => (fileName == source.fileName && !IsOpen("Open"))

Table 4.5 - Second level of the notepad application's navigation map regarding the *Open* functionality.

Notepad

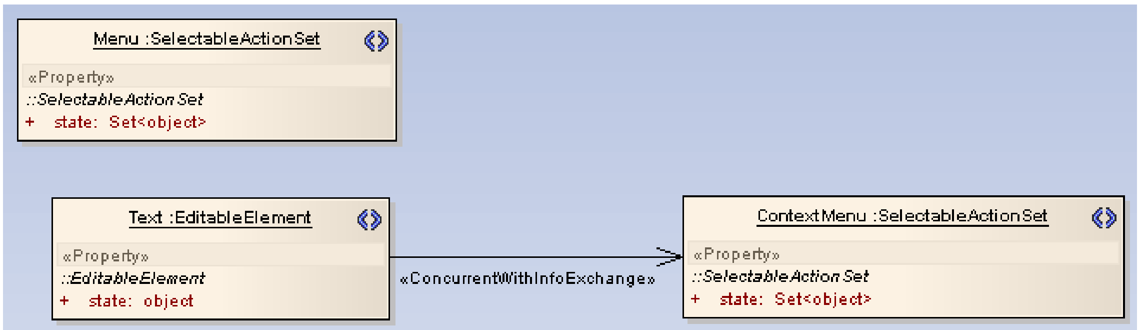


Fig 4.6 – Second abstraction level for the *Notepad*

Second Abstraction Level (Notepad)			
Name	**Instance of**	**Composite**	**Property**
Menu	*SelectableActionSet*	False	Set<object> state
Text	*EditableElement*	False	object state
ContextMenu	*SelectableCollection*	False	Set<object> state

Connectors				
Source	**Destination**	**Stereotype**	**Pre**	**Post**
-	-	-	-	-

Table 4.6 - Second level of the notepad application's navigation map regarding the *Notepad* functionality.

4.3 Spec# Model Obtained

This section presents a piece of the generated Spec# model. As explained in section 3.2, step number 3 will perform a translation from the visual model to the corresponding Spec#. This step was performed by means of a software tool (XMI2Spec#), which was built during the course of this dissertation, following the translation rules presented in section 3.5. The Spec# model shown in this section is related to both abstraction levels of the notepad GUI model. In fact, regarding the second level, it will only demonstrate the appropriate specification for the Find element of the navigation map. The specification of the other elements will be presented in Appendix B.

Listing 4.1 displays the navigation map and Listing 4.2 presents the second abstraction level of the navigation map for the Find window.

First Level

```
using WindowManager;
namespace MyNotepad;

//state variables

boolean exit = false;
string fileName = "";
string findWhat = "";
string replaceWith = "";
string selectedText = "";
string text = "";
boolean dirty = false;

//actions

[Action]
public void Notepad_Find_Find()
requires IsEnabled("Notepad") && text != "" && !IsOpen("Replace");
ensures Find.findWhat == findWhat && IsEnabled("Find");
{
   //TODO
}

[Action]
public void Notepad_Open_Open()
requires IsEnabled("Notepad") && !dirty;
ensures Open.fileName == fileName && IsEnabled("Open");
{
  //TODO
}

[Action]
public void Notepad_SaveAs_SaveAs()
requires IsEnabled("Notepad");
ensures SaveAs.fileName == fileName;
{
  //TODO
}

[Action]
public void Notepad_Open_SaveChanges()
```

```
requires IsEnabled("Notepad") && dirty ;
ensures IsEnabled("SaveChanges?");
{
  //TODO
  AddWindow("SaveChanges? ",  "Notepad", true);
}


[Action]
public void Notepad_Exit_SaveChanges()
requires IsEnabled("Notepad") && exit;
ensures IsEnabled("SaveChanges?");
{
  //TODO
}

[Action]
public void Notepad_Exit ()
requires IsEnabled("Notepad") && !dirty;
ensures !IsOpen("Notepad");
{
  //TODO
}

[Action]
public void Notepad_Replace_Replace()
requires IsEnabled("Notepad") && !IsOpen("Find");
ensures Replace.findWhat == findWhat && Replace.replaceWith == replacewith &&
IsEnabled("Replace");
{
  //TODO
}

[Action]
public void ChooseOption(string op)
requires IsOpen("SaveChanges?") ;
{
  // TODO
}

[Action]
public void SaveChanges_Yes_SaveAs()
requires  IsEnabled("SaveChanges?") && fileName == "";
ensures !IsOpen("SaveChanges") && IsEnabled("SaveAs");
{
  //TODO
}

[Action]
public void SaveChanges_No_Notepad()
requires  IsEnabled("SaveChanges?") && fileName == "";
ensures !IsOpen("SaveChanges") && IsEnabled("Notepad");
{
  //TODO
}
```

Listing 4.1 – Spec# model for the first level of the notepad application’s navigation map

Second Level (Find)

```
using WindowManager;
namespace Find;

var boolean directionState = false; //down <- false; up <- true
var object findWhat = null;
var boolean matchCaseState = false;

//properties

public boolean DirectionState
{
  [Action(kind=Probe)] get
  requires IsEnabled("Find");
  { return directionState;}
  [Action] set
  requires IsEnabled("Find");
  { directionState = value;}
}

public object FindWhat
{
  [Action(kind=Probe)] get
  requires IsEnabled("Find");
  { return findWhat.state;}
  [Action] set
  requires IsEnabled("Find");
  { findWhat.state = value;}
}

public boolean MatchCaseState
{
  [Action(kind=Probe)] get
  requires IsEnabled("Find");
  { return matchCaseState;}
  [Action] set
  requires IsEnabled("Find");
  { matchCaseState = value;}
}

//actions

[Action]
public void FindNext (object obj)
requires IsEnabled("Find") && findWhat!= "";
ensures  MyNotepad.AuxFind(findWhat.state, matchCase, direction) =>
MyNotepad.selectedText == findWhat;
{
  //TODO
}

[Action]
public void FindNext_FindWord_CannotFind (object obj)
requires IsEnabled("Find") && findWhat!= "";
ensures  !MyNotepad.FindWord(findWhat, matchCase, direction) =>
IsEnabled("CannotFind");
{
    AddWindow("CannotFind", "Find", true);
}

[Action]
public void CannotFindAckMsgBox()
requires IsEnabled("CannotFind");
```

```
ensures !IsOpen("CannotFind");
{
  Ack(); //TODO
  RemoveWindow("CannotFind");
}

[Action]
public void Find_Cancel_Notepad()
requires IsEnabled("Find");
ensures !IsOpen("Find");
{
  //TODO
}
```

Listing 4.2 – Spec# model for the second level of the notepad application's navigation map for the Find window

The word "//TODO" which appears repeatedly in Listing 4.2, represent multiple lines of specification which were added manually. The average time required to perform such task was about half an hour.

4.4 Metrics

Several experiments were performed in order to measure the time and effort required to model a GUI using VAN4GUIM UML profiles and also to evaluate the overall testing approach proposed in this dissertation. These experiments were based on the Microsoft Notepad.

The visual modelling notation (VAN4GUIM) took ten days to be defined. It included the analysis of the existing notations, selection of the notations to be used as basis of the new language, the description of the behaviour extensions to the previous selected languages and the definition of the UML Profiles.

For each experiment several metrics were taken into account: GUI modelling effort (time needed); time required for adding behaviour to the preliminary automatic constructed Spec# model; time needed to validate the obtained model in Spec#.

The Notepad visual model using VAN4GUIM UML profiles was constructed in 5 hours; It took half an hour to add additional behaviour (mainly related to the find functionality); in order to validate the obtained model, several views (groups of states) were defined. These views were constructed inside Spec Explorer and took one hour to be completed and checked against the VAN4GUIM model.

Both quality and correctness of the Notepad visual model were accessed according to coverage criteria defined in section 3.4. The quality of the FSM was measured according to the coverage criteria defined in section 3.6 [3].

This dissertation emerges on the initial phase of a GUI testing process (MBT). It represents an input to the approach followed by [3]. At the end phase of the process, all errors founded are exactly the same as the ones detected in the approach carried out by [3], but with less effort required in the modelling activity (about 40% savings). This is due to the deep knowledge of the UML notation on which this language is based on.

Chapter 5

Conclusions and future work

This chapter presents a summary of the main contributions of the work reported in this dissertation in the fields of GUI visual modelling, GUI testing, and test adequacy analysis.

This research work started with an overall analysis of the state of the art on GUI modelling topic. Such analysis provided a means of understanding the work that has been done towards GUI modelling. In addition, it also gave birth to new ideas that were taken into account in order to fulfil the goals proposed. Some of those ideas have involved and therefore led to the design of a new original visual notation – the VAN4GUIM notation.

The **VAN4GUIM** is a visual notation developed as five different UML profiles and is based on three notations/concepts: Canonical Abstract Prototyping notation; ConcurTaskTrees (CTT) notation; and the Window Manager concept. This new notation has helped to accomplish the initial proposed goal of hiding formalism details inherent to models used in model-based testing (MBT) approaches diminishing the GUI modelling effort. In addition, a software tool was developed in order to automatically translate the GUI models, developed with VAN4GUIM notation, to Spec#. The translation process is based on a previously defined **set of rules** which describe how to perform such translation, as can be seen in section 3.5. Despite the fact that this tool (**XMI2Spec#**) has facilitated the translation to Spec#, the Spec# model may still need to be completed manually with additional behaviour, which is not included in the visual model (developed in a higher abstraction level). A scientific paper was submitted to an international conference called "ICSOFT 2008 – 3rd International Conference on Software and Data Technologies" and has been accepted.

As stated in section 4.2.1, the effort required to model a GUI using VAN4GUIM is lower than the effort required to model the same GUI directly in Spec# (around 40% less). Furthermore, VAN4GUIM is a powerful and intuitive notation, since it is based on ConcurTaskTrees, which is a widely accepted notation, and on Canonical Abstract Prototypes, which offer a significant amount of capabilities/resources to model GUIs.

As future work, it is expected to improve the VAN4GUIM notation by adding visual descriptive icons to each UML Profile. This would increase the expressiveness of the notation for testers and modellers (end users of the notation).

In summary, all of the proposed goals have been successfully achieved.

References

[1] M. U. a. B. Legeard, *Practical model-based testing*, 2007.

[2] A. C. Paiva, J. C. P. Faria, and R. M. Vidal, "Towards the Integration of Visual and Formal Models for GUI Testing," in *Third Workshop on Model Based Testing (MBT'07)*, Braga, Portugal, 2007.

[3] A. C. R. Paiva, "Automated Specification-Based Testing of Graphical User Interfaces," in *Department of Electrical and Computer Engineering* Porto: Engineering Faculty of Porto University (Ph.D thesis), 2007, p. 228.

[4] P. P. a. R. Bastide, "UML for Interactive Systems: What is Missing."

[5] J. S. Harald Fecher, Marcel Kyas, and Willem-Paul de Roever, "New Unclarities in the Semantics of UML 2.0 State Machines."

[6] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview," in *CASSIS'04 - International workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, Marseille, 2004.

[7] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object-oriented Systems*. New York: Springer-Verlag Telos, 2005.

[8] E. Schlungbaum, "Model-based User Interface Software Tools Current state of declarative models," Graphics, Visualization & Usability Center GIT-GVU-96-30, 1996.

[9] W. W. Kai Blankenhorn, "Extending UML to GUI Modeling," 2004.

[10] P. T. Lethbridge, "Advanced Software Design and Reengineering - UML Extension Mechanisms," 2008.

[11] P. P. d. S. a. N. W. Paton, "User Interface Modelling with UML," 2000.

[12] P. P. d. Silva and N. W. Paton, "UMLi: The Unified Modeling Language for Interactive Applications," in *Third International Conference UML2000 - The Unified Modeling Language. Advancing the Standard*, 2000, pp. 117-132.

[13] N. J. Nunes and J. F. e. Cunha, "Towards a UML profile for interaction design: the Wisdom approach," in *Third International Conference*, York, UK, 2000, pp. 101-116.

[14] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan, and M. Florins, "USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces," in *WMI'04: Proceedings of the W3C Workshop on Multimodal Web Applications*, 2004.

[15] S.-C. R. I. Project, "CAMELEON Project Homepage."

[16] B. E. John and D. E. Kieras, "The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast," *ACM ToCHI,* 1996.

[17] H. R. Hartson, A. C. Siochi, and D. Hix, "The UAN: a user-oriented representation for direct manipulation interface designs," in *ACM Transactions on Information Systems (TOIS)*: ACM Press, 1990.

[18] F. Paternò, C. Mancini, and S. Meniconi, "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," in *Interact'97*, 1997.

[19] N. S. Dan Diaper, *The Handbook of Task Analysis for Human-Computer Interaction*, 2004.

[20] N. J. N. a. H. C. Leonel Nóbrega, "Mapping ConcurTaskTrees into UML 2.0," 2005.

[21] F. Paternò, "Towards a UML for Interactive Systems," in *Proceedings of Human-Computer Interaction Conference HCI'2001*, 2001.

[22] G. Mori, F. Paternò, and C. Santoro, "CTTE: Support for Developing and Analyzing Task Models for Interactive System Design," *IEEE Transactions on Software Engineering,* vol. 28, 2002.

[23] L. L. Constantine, "Canonical Abstract Prototypes for Visual and Interaction Design," in *Interactive Systems. Design, Specification and Verification DSV-IS'03*, 2003.

[24] Wikipedia, "Notepad - Wikipedia, the free encyclopedia."

Appendix A

VISUAL ABSTRACT NOTATION FOR GUI MODELLING AND TESTING
VAN4GUIM

Rodrigo Moreira, Ana C. R. Paiva
Faculty of Engineering of the University of Porto, Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
ei04093@fe.up.pt, apaiva@fe.up.pt

Keywords: GUI modelling; Visual GUI modelling; GUI testing; Test coverage criteria; UML Profiles.

Abstract: This paper presents a new Visual Notation for GUI Modelling and testing (VAN4GUIM) which aims to hide, as much as possible, formalism details inherent to models used in model-based testing (MBT) approaches and to promote the use of MBT in industrial environments providing a visual front-end for modelling which is more attractive to testers than textual notation. This visual notation is developed as five different UML profiles and based on three notations/concepts: Canonical Abstract Prototyping notation; ConcurTaskTrees (CTT) notation; and the Window Manager concept. A set of translation rules was defined in order to automatically perform conversion from VAN4GUIM to Spec#. GUI models are developed with VAN4GUIM notation then translated automatically to Spec# that can be then completed manually with additional behaviour not included in the visual model. As soon as a Spec# model is completed, it can be used as input to Spec Explorer (model-based testing tool) which generates test cases and executes those tests automatically.

1 INTRODUCTION

GUI testing is an area of increasing importance, where the tests are performed from the end users point of view. Software companies have the best of interests on finding defects on their products before their costumers' do, not only to meet user demands and therefore increase confidence in relation to their software, but also to induce correctness and commitment with them. For these reasons, GUI testing is extremely necessary. It is particularly time consuming, labour-intensive, expensive and difficult. Presently used GUI testing methods are almost ad hoc and require test engineers to manually develop the necessary scripting to perform test execution, and though evaluate if the GUI is effectively tested. However, there are some tools that can help improving GUI testing process. Some of these tools exploit a broadly accepted method that generates GUI test scripts which relies on the capture/playback technique. Such technique requires testers to perform labour-intensive interaction with the GUI via mouse events and keystrokes. During interaction user events are recorded into scripts which and can be automatically played later for GUI testing. However, when different inputs are required to conduct the test or even if the GUI changes, it is then required to re-generate the test scripts. In addition, it is hard to cover all possible test cases for all GUI components and capture/playback method often records redundant data (Utting and Legeard, 2007).

The use of a model to describe the behaviour of a system is an established and key advantage regarding testing. Models can be used in numerous ways, for instance, to improve quality of software documentation, code generation and test case generation. Model-based testing represents the automation of the design of black-box tests. The usage of a model to describe the behaviour of a GUI in combination with an automated test tool to generate test cases, execute those tests and report errors found, can dramatically reduce the time required meant for testing software.

In recent times, model-based testing has been receiving attention due to the potential to automate test generation and increasing model driven software engineering practices. Nevertheless, the usage of uncommon and unfeasible modelling notations, the lack of integrated tool environments and support, the difficulties inherent to the constructions of models, the test case explosion problem, the gap between the model and the implementation, remain as obstacles regarding the adoption of model-based GUI testing approaches. In addition, the models used are often textual models and usually testers and modellers prefer working with visual/graphical notations.

The goal of this research work is to

- Develop a visual modelling front-end hiding as much as possible the formal details from modellers and testers.
- Define a set of rules to translate the visual notation into Spec# (Barnett et al., 2005) (an extension of C# with contracts).
- Develop a tool to automate the translation from visual model to Spec# and ensure consistency between both models.

2 STATE OF THE ART

In current times, GUIs play an important role in most of software systems, as they represent the fore-front of systems. UML is a natural candidate for GUI modelling since it represents a standard notation for object-oriented modelling of applications. GUIs can be decomposed in two main groups: a dynamic or **behaviour** group and a static or **layout** group. While the dynamic group can be modelled using existing UML diagrams and elements, GUI layout cannot, due to the fact that all existing UML diagrams are not layout-aware (Blankenhorn and Walter, 2004). In addition, it is not clear and simple to identify how UI elements, such as user tasks and display, are supported by UML. As such, it is necessary to make use of UML extension mechanisms, like **constraints**, **tagged values**, and **stereotypes**, in order to provide more flexibility to the existing UML notation. With these extension mechanisms it becomes possible to style several **UML profiles** for GUI modelling.

2.1 UML Profile for GUI Layout

Several researchers have recognized the lack of support for layout information in UML and thus have taken different approaches. Kai Blankenhorn and Wilhelm Walter (Blankenhorn, 2004) have developed an UML Profile for GUI Layout, which is a UML 2.0 profile that uses Diagram Interchange to store layout information while staying fully conform to standards. The diagram-interchange specification originates XMI from the XML metadata interchange format, which is used for storing information about the elements of a UML diagram. The profile's meta-model makes use of stereotyped classes that are linked by constrained associations, taking benefit from UML 2.0 extension mechanisms. In order to improve the usefulness of the graphical language and to transfer the general look of designer sketches to models, the authors have developed a set of stereotype icons. They claim that their approach yields benefits for those involved in the design process of GUIs. Designers are their main audience. The profile is best suited for creating an initial model of the layout and navigational concept of the application. However it does not model the behaviour of the GUI.

2.2 UMLi

The UMLi notation (Silva and Paton, 2000) aims to be a light-weight extension to the UML notation with the purpose to provide greater support for UI design, becoming possible to model both behaviour and structure of a system. However, modelling the behaviour of a system via UMLi is not indeed straightforward due to its complexity. UMLi notation has been influenced by model-based user interface development environment (MB-UIDE) technology. In addition, the authors of UMLi believe that the MB-UIDE technology offers many insights into the abstract description of user interfaces that can be adapted for use with the UML technology, such as techniques for specifying **static** and **dynamic** aspects of user interfaces using declarative models. The notation defines three distinct types of models: **presentation** model, **domain** model and **behaviour** model. The presentation model represents the visual part of the user interfaces that can be modelled using object diagrams composed of interaction objects. Domain models specify classes and objects that represent the system entities, the domain elements. Behaviour models describe object collaboration and common interaction behaviour, such as tasks, actions and events.

2.3 Wisdom Profile

The Wisdom Profile is proposed by Nunes and Cunha (Nunes and Cunha, 2000), for the documentation, specification and design of interactive systems. They propose a minimal set of extensions for a UML profile for interactive systems development taking advantages of human-computer interaction domain knowledge under the notation and semantics of the UML. The Wisdom approach suggests two important models: the **analysis model** and the **interaction model**. The latter includes the information, dialogue and presentation dimensions, mapping the conceptual architectural models for interactive systems, while maintaining the desired separation of concerns. The analysis model encompasses the UML profile architecture and shared information. During the design phase, the interaction model embraces two other models: the **dialogue model** and the **presentation model**. The former specifies the dialogue structure of the application, using an UML based approach of the ConcurTaskTrees (CTT) notation. The latter defines the physical realization of the interactive system, centring on the structure of the different presentation entities in order to realize the physical interaction with the user. The authors propose a set of UML extensions to support the design model.

2.4 usiXML

UsiXML (User Interface eXtensible Markup Language) is a XML-compliant markup language that describes the User Interface (UI) for multiple contexts of use such as Character User Interfaces (CUIs), Graphical User Interfaces (GUIs) and Multimodal User Interfaces (Vanderdonckt et al., 2004). With usiXML it becomes possible to specify a user interface at different levels of abstraction while maintaining the mappings between those levels, whenever required. This notation is based on five main concepts: expressiveness of UI (depends on the context of use), central storage of models, transformational approach (each model may be subject to several transformations supporting various development keys), multiple development paths, and flexible development approaches (top-down, bottom-up, wide-spreading). The main audience for usiXML are analysts, modellers, designers, and others.

2.5 Canonical Abstract Components

The concept of abstract user interface prototypes offers designers a form of representation for specifying and exploring visual and interaction design ideas that are between abstract task models and realistic or representational prototypes. They represent an intermediate form that can speed the user interface design process and improve the quality of the result. As abstractions, they can serve as an intermediate bridge between task models and realistic designs, smoothing, simplifying, and systematizing the design process. Canonical Abstract Prototypes (CAP) are an extension to usage-centred design which provides a formal vocabulary for expressing visual and interaction designs without concern with details of appearance and behaviour. CAPs embody a model specifically created to support a smooth progression from abstraction toward realization in user interface design. Each Canonical Abstract Component is comprised by a symbolic graphical identifier and a descriptive name. The graphical symbols aim to serve as learned shorthand for the various functions available. The notation is quite simple, since it is built on two basic symbols: a **generic tool** or **action** and a **generic material** or **container**. Materials are the containers, content, information or data. Tools are the actions, operators, mechanisms, or controls that can be used to create, manipulate, transform or operate upon materials. The combination of a container and an action form a generic hybrid component.

2.6 ConcurTaskTrees

The ConcurTaskTrees (CTT) is one of the most widely used notations for task modelling, specifically tailored for UI model-based design. This notation has been developed taking into account the previous experience in task modelling and adding new features to better obtain an easy-to-use powerful notation, to describe the dialogue in interactive systems. In fact, CTT provides the concept of hierarchical structure, exposing a wide range of granularity allowing large and small structures to be reused and, enables reusable task structures to be defined at both low and high semantic level. CTT introduces a rich set of graphical temporal operators, with a higher expressiveness than those offered by concurrent notations. In a model-based GUI testing approach, task models can be used to define the behaviour of user interfaces (Silva et al., 2007).

2.7 Spec#

The Spec# programming system represents an attempt to develop a more cost effective way to maintain software in high standards, and has been developed at Microsoft Research lab, in Redmond, USA. The Spec# system consists of three components: the Spec# programming language, the Spec# compiler, and the Spec# static program verifier (Barnett et al., 2005). The Spec# programming language extends the existing object-oriented .NET programming language C# and expands the type system to include non-null types and checked exceptions. It also provides method contracts in the form of pre- and post-conditions, and also invariants. Since all of the specifications written in Spec# may be executable, it is possible to specify invariants, pre- and post-conditions, and executable method bodies in a high-level action language, with primitives to change the value of state variables, and even call external methods defined in .NET assemblies. Spec# provides the ability to build a formal specification of an interactive application, describing the actions that a user may perform when interacting with the system, in the terms of changes to the state of the application. Using Spec#, one can build a formal specification of an interactive application, describing the actions a user can perform at each moment, and the expected effect of each user action, in terms of changes to the state of the application (according to a model of the application state as perceived by the user) and possible effects to the environment (Paiva, 2007). The effect of user actions may depend not only on the current state of the application, but also on environment conditions. The state of the application is described by means of state variables.

3 VAN4GUIM OVERVIEW

The VAN4GUIM (Visual Abstract Notation for GUI Modelling) was developed based on UML extension mechanisms, UML Profiles. An UML Profile can be useful for building UML models for particular domains. They are based on stereotypes and tagged values that are applied to elements, attributes, methods, links, and link ends. Those extensions together with added restrictions define UML meta-models that can be used to construct models for such particular domains.

The VAN4GUIM UML Profiles are based on three notations/concepts:

- **Canonical Abstract Prototyping** which is a notation introduced by Larry Constantine (Constantine, 2003). The prototypes are an extension to usage-centred design that provide a formal vocabulary for expressing visual and interaction designs without concern for details of shape and behaviour.
- A commonly accepted and widely applied notation, **ConcurTaskTrees** (CTT) (Paternò et al., 1997), which initial goal was to support designers of interactive systems. The CTT is a notation for task modelling being able to graphically represent a hierarchical structure, with a set of temporal operators capable of describing concurrent behaviour.
- **Window Manager** Concept is useful to describe the common behaviour of windows showing up and disappearing during the execution of a window application.

The Canonical Abstract Prototyping notation was extended with behaviour (state, properties with associated set and get methods (Figure 1), methods and restrictions such as pre- and post-conditions) and the CTT notation was extended with restrictions over the operators which define how to use them correctly (from the VAN4GUIM point of view).

VAN4GUIM is composed of five different UML Profiles:

- **Containers** – Is a subset of Canonical Abstract Components which act as holders of user interface objects (generically called *DataStores*). A *Container* extends *DataStore* and can hold an object (*Element*) or a set of objects (*Collection*) (Figure 1).

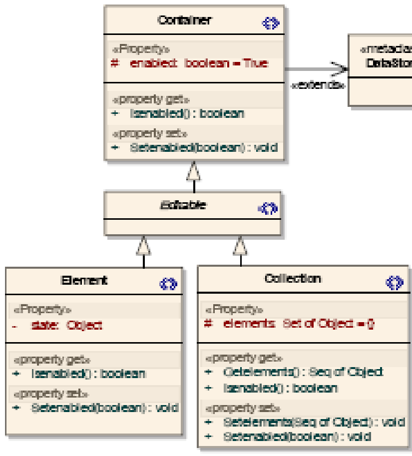


Figure 1: Containers UML Profile.

- **User Actions** – Is a subset of Canonical Abstract Components which represent tools (actions, operators, mechanisms, or controls) that can be applied upon containers (generically called *InteractionFunctions*).

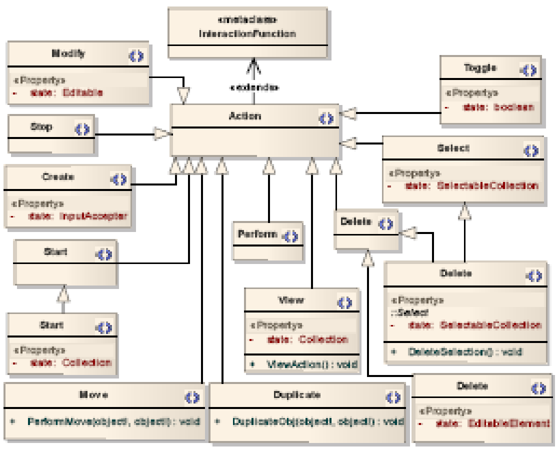


Figure 2: User Actions UML Profile.

An *Action* extends *InteractionFunction* and can model several different user actions, such as, *Modify*, which updates an *Editable* container, and *Move*, which moves an object from a source to a target object (Figure 2). *Move* has additional restrictions to model behaviour, such as, its parameters cannot be null (represented by an exclamation mark "!" at the of the parameter's type) and a pre-condition stating that the source should be different from the target (*source != target*).

- **Hybrids** – Are combinations of *DataStores* and *InteractionFunction* (Figure 3). They are used to model user actions that take place over specific containers. For instance, a *SelectableCollection* is the combination between a *Collection Container* and a *Select Action* (Figure 2) which is *per si* a restriction on its behaviour.

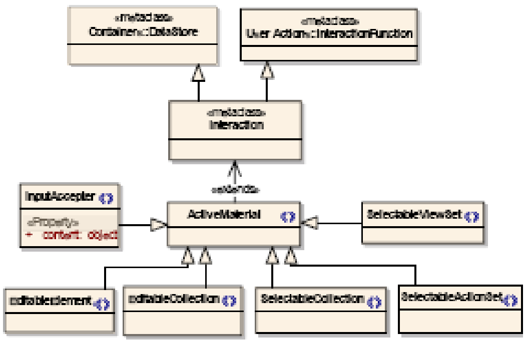


Figure 3: Hybrids UML Profile.

- **CTT Connectors** – Is based on CTT notation and describes relationships between two elements of the VAN4GUIM (Figure 4). E.g., the *EnablingWithInfoExchange* connector transfers information between its source and its target and, at the same time, sets the *enabled* property of the target to true. This is described by the following post-condition *source.enabled == true*.

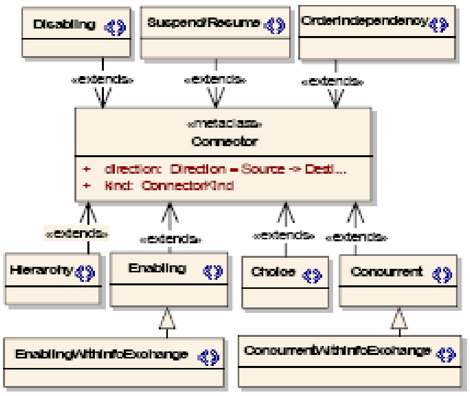


Figure 4: CTT Connectors UML Profile.

- **Window Manager** – To describe the windows' behaviour (Figure 5).

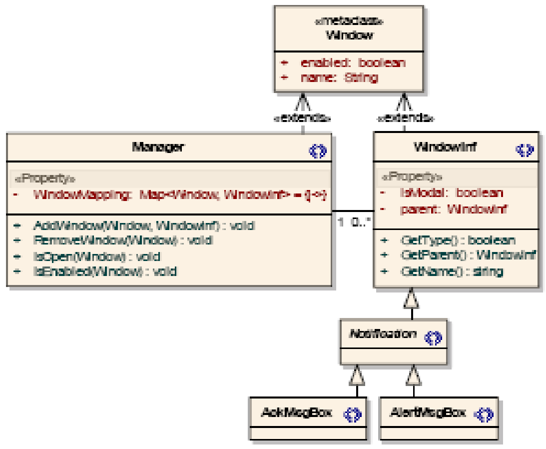


Figure 5: Window Manager UML Profile.

The behaviour added to the VAN4GUIM notations is taken into account by the translation process to Spec#.

4 GUI MODEL

The GUI model constructed in VAN4GUIM is a state machine diagram in which states can be instances of any element within Containers, User Actions, Hybrids and Window Manager Profiles. Transitions between states are elements within CTT Connectors Profile.

A GUI model constructed in VAN4GUIM notation will have, at least, two levels:

- **A Navigation map diagram** – this diagram shows the set windows of the GUI and the possible transitions between them which represent the possible actions the users can perform to open/close a specific window of the GUI.

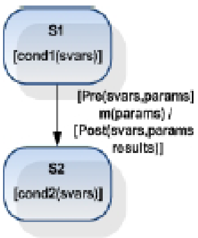
- **Behaviour of each Window** – this diagram describes the behaviour of each window, e.g., the containers and the set of actions the user can perform and the relationship between elements of the diagram. At this level of abstraction, it is possible to have *AckMsgBox* and *AlertMsgBox* but it is not possible to have other kind of windows from the Window Manager Profile.

However, situations may occur where more than two model levels can be useful. It is responsibility of the modeller to decide how many levels the GUI model should have.

5 VISUAL TO TEXTUAL TRANSLATION RULES

GUI models constructed based on VAN4GUIM are translated to Spec# textual notation according to some rules that are presented next. The behaviour within the GUI model and GUI Profiles are taken into account.

5.1 Simple Transition

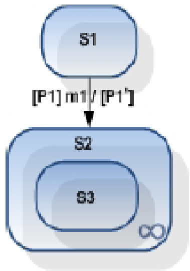



```
[Action] m(params)
requires cond1(svars)
&& Pre(svars,params);

ensures
Post(svars,params,results)
&& cond2(svars);

{ //TODO }
```

S1 and *S2* represent an instance of a stereotype available from any of the Profiles defined.

In order to simplify the state machines and expressions, from now on, it is assumed that *Si* is a condition over state variables in state *i*; whenever states represent windows, *Ni* is the name of the window *i* and *[Pi]mi/[Pi']* are transitions between states in which *[Pi]* is a pre-condition over state variables and parameters, *mi* is a function with (omitted) parameters and *[Pi']* is a post-condition over state variables, parameters and result of the executed function.

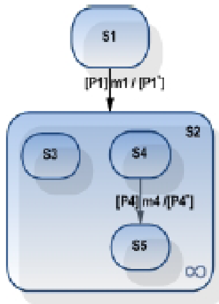
5.2 Transition to a composite state




```
namespace N1;
using WindowManager;

[Action] m1
requires S1 && [P1];
ensures [P1'] && S2
        && S3; {
 AddWindow(N2,(true,S1));
 //TODO
}
```

S1 and *S2* are instances of the *windowInf* stereotype; *S3* can be any instance of any stereotype of any profile (except a *windowInf*).

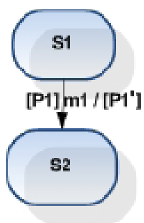
5.3 Transition to a composite state with two or more possible initial states




```
namespace N1;
using WindowManager;
[Action] m1
requires S1 && [P1];
ensures [P1'] && S2
        && S3 && S4;
{ //TODO }

namespace N2;
using WindowManager;
[Action] m4
requires S4 && [P4]
ensures [P4'] && S5
{ //TODO  }
```

S1 and *S2* represent instances of *windowInf* stereotype; *S3*, *S4* and *S5* can be instances of any stereotype of any profile (except *windowInf*).

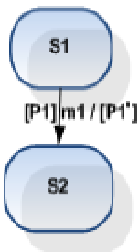
5.4 Transition to a acknowledge message state




```
[Action] m1
requires S1 && [P1];
ensures [P1'] && S2 &&
        IsOpen(N2); {
 AddWindow(S2,(true,S1));
 //TODO
}

[Action] Ok
requires IsOpen(N2)
ensures [P1'] &&
        !IsOpen(N2){
  RemoveWindow(N2);
  //TODO
}
```

S1 represent an instance of any stereotype of any profile; *S2* is an instance of the stereotype *AckMsgBox* (modal window).

5.5 Transition to an alert message state




```
[Action] m1
requires S1 && [P1];
ensures [P1'] && S2 &&
              IsOpen(N2);  {
  AddWindow(N2,(true,S1));
  //TODO
}
[Action]ChooseOp(string op)
requires IsOpen(N2) && S2;
{  //TODO   }
```

S1 is an instance of any stereotype of any profile (except *windowInf*). *S2* is an instance of the *AlertMsgBox* stereotype (modal window).

6 CASE STUDY

The Microsoft Notepad text editor is used to illustrate the approach.

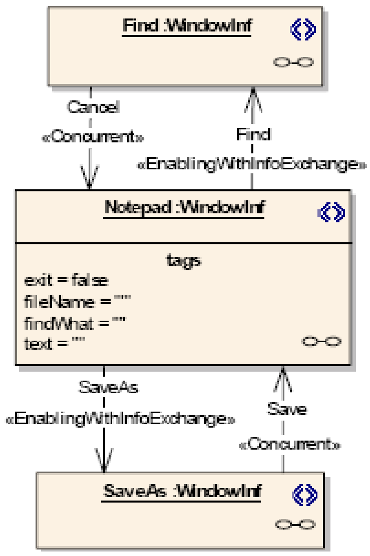


Figure 6: Part of the Navigation map.

Tagged values are translated to instance variables (e.g., *fileName*, and *text* in Notepad window).

The Spec# specification generated automatically from the diagram in Figure 6 is listed below.

```
namespace Notepad;
using WindowManager;
//state variables
string fileName = "";
string text = "";

// Actions
[Action]
public void FindCancelNotepad()
ensures !IsOpen("Find"); {
  //TODO
}
[Action]
public void NotepadFindFind()
requires Notepad.Text.Length > 0 &&
         !IsOpen("Replace");
ensures Find.findWhat==Notepad.findWhat;
{ // TODO }
```

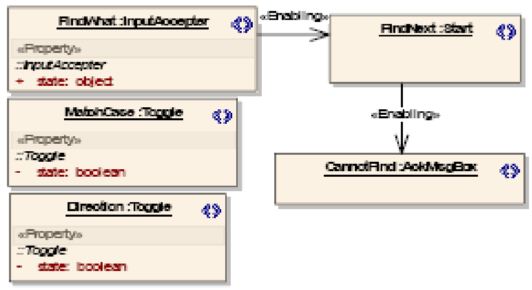


Figure 7: Find window behaviour.

Part of the Spec# specification generated automatically from the diagram in Figure 7 is listed below.

```
namespace Find;
using WindowManager;

//state variables
private boolean directionState = false;
private string findWhat = null;
private boolean matchCaseState;

// Properties
public string DirectionState {
  [Action(kind=Probe)] get
  requires IsEnabled("Find");
  { return directionState;}
  [Action] set
  requires IsEnabled("Find");
  { directionState = value;}
}
// similar properties for FindWhat and
// MatchCase states

//Actions
[Action] public void
FindNextStart(object obj)
requires FindWhat.state!= "";
ensures IsEnabled("CannotFind"); {
  AddWindow("CannotFind",(true,"FindNext"));
}
```

```
[Action] public void
CannotFindAckMsgBox() {
requires !IsEnabled("CannotFind");
ensures !IsOpen("CannotFind"); {
  RemoveWindow("CannotFind");
}
```

6 CONCLUSIONS

This paper has presented a new visual modelling language for GUI modelling called VAN4GUIM. It extends previous notations found in the literature by defining five different UML Profiles. It extends Canonical Abstract Components and ConcurTaskTree, and also the Window Manager concept by defining such UML Profiles with attributes, properties, restrictions (invariants, pre- and post-conditions) and operations.

The VAN4GUIM notation took ten (part time) days to be defined which included the analysis of existing notations, selection of the notations to be used as basis of this work and the description of behaviour as additional restrictions.

The Notepad model took 5 hours to be constructed and half an hour to add behaviour to the automatically translated Spec# model (mainly related to the find functionality).

It is our strong believe that such a notation will increase the acceptance of Model-Based GUI testing in industry since it is more pleasant and based on the widely used and known UML modelling language.

The VAN4GUIM together with the automatic translations mechanism provides savings in the time spent with the modelling activity around 40% when compared with the GUI modelling directly in Spec#.

REFERENCES

Barnett, M., Deline, R., Jacobs, B., Fähndrich, M., Leino, K. R. M., Schulte, W. & Venter, H. (2005) The Spec# Programming System: Challenges and Directions. *VSTTE2005*.

Blankenhorn, K. (2004) A UML Profile for GUI Layout. *Department of Digital Media*. University of Applied Sciences Furtwangen.

Blankenhorn, K. & Walter, W. (2004) Extending UML to GUI Modeling. IN R. KEIL-SLAWIK, H. S., G. SZWILLUS (Ed.) *Mensch & Computer 2004*. Allgegenwärtige Interaktion, München, Oldenbourg Verlag.

Constantine, L. L. (2003) Canonical Abstract Prototypes for Visual and Interaction Design. *Interactive Systems. Design, Specification and Verification DSV-IS'03*. Springer LNCS 2844.

Nunes, N. J. & Cunha, J. F. E. (2000) Towards a UML profile for interaction design: the Wisdom approach. IN SELIC, A. E. A. S. K. A. B. (Ed.) *Third International Conference*. York, UK, Springer.

Paiva, A. C. R. (2007) Automated Specification-Based Testing of Graphical User Interfaces. *Department of Electrical and Computer Engineering*. Porto, Engineering Faculty of Porto University (Ph.D thesis).

Paternò, F., Mancini, C. & Meniconi, S. (1997) ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. *Interact'97*.

Silva, J. L., Campos, J. C. & Paiva, A. C. R. (2007) Model-based user interface testing with Spec Explorer and ConcurTaskTrees. *2nd International Workshop on Formal Methods for Interactive Systems*. Lancaster, UK.

Silva, P. P. D. & Paton, N. W. (2000) UMLi: The Unified Modeling Language for Interactive Applications. *Third International Conference UML2000 - The Unified Modeling Language. Advancing the Standard.*

Utting, M. & Legeard, B. (2007) *Practical Model-Based Testing: A Tools Approach,* San Francisco, Morgan Kaufmann Publisher.

Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D. & Florins, M. (2004) USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces. *WMI'04: Proceedings of the W3C Workshop on Multimodal Web Applications.*

Appendix B

Second Level (Replace)

```
using WindowManager;
namespace Replace;

object findWhat;
object replaceWith;
boolean matchCaseState;


public object FindWhat
{
  [Action(kind=Probe)] get
  requires IsEnabled("Replace");
  { return findWhat.state;}
  [Action] set
  Requires IsEnabled("Replace");
  { findWhat.state = value;}
}

public object ReplaceWith
{
  [Action(kind=Probe)] get
  requires IsEnabled("Replace");
  { return replaceWith.state;}
  [Action] set
  requires IsEnabled("Replace");
  { replaceWith.state = value;}
}

public boolean MatchCaseState
{
  [Action(kind=Probe)] get
  requires IsEnabled("Replace");
  { return matchCase;}
  [Action] set
  requires IsEnabled("Replace");
  { matchCase = value;}
}

//actions

[Action]
public void FindNext (object obj)
requires IsEnabled("Replace") && findWhat != null;
{
  //TODO
}

[Action]
public void FindNext_Find_CannotFind(object obj)
requires !MyNotepad.FindWord(findWhat, matchCaseState)  && findWhat != null;
ensures IsEnabled(“CannotFind”);
{
   AddWindow(“CannotFind”, “Replace”, true);
}
```

```
[Action]
public void Replace_Find_CannotFind(object obj)
requires !MyNotepad.FindWord(findWhat, matchCaseState)  && findWhat != null;
ensures IsEnabled("CannotFind");
{
   AddWindow("CannotFind", "Replace", true);
}

[Action]
public void ReplaceAll_Find_CannotFind(object obj)
requires !MyNotepad.FindWord(findWhat, matchCaseState) && findWhat != null;
ensures IsEnabled("CannotFind");
{
   AddWindow("CannotFind", "Replace", true);
}

[Action]
public void CannotFindAckMsgBox()
requires IsEnabled("CannotFind")
ensures !IsOpen("CannotFind")
{
  //TODO
  RemoveWindow("CannotFind");
}

[Action]
public void Replace (object obj)
requires IsEnabled("Replace")  && findWhat!= null;
{
  //TODO
}

[Action]
public void ReplaceAll (object obj)
requires IsEnabled("Replace") && findWhat!= null;
{
  // TODO
}

[Action]
public void Replace_Cancel_Notepad()
requires IsEnabled("Replace");
ensures !IsOpen("Replace");
{
  //TODO
}
```

Second Level (SaveAs)

```
using WindowManager;
namespace SaveAs;

object filename;
Set<object> saveAsType;
Set<object> encoding;

//properties

public object Filename
{
  [Action(kind=Probe)] get
  requires IsEnabled("SaveAs");
```

```
  { return filename.State;}
  [Action] set
  requires IsEnabled("SaveAs");
  { Filename.State = value;  }
}

public Set<object> SaveAsType
{
  [Action(kind=Probe)] get
  requires IsEnabled("SaveAs");
  { return saveAsType.selection;}
  [Action] set
  requires IsEnabled("SaveAs");
  { saveAsType.selection = value;}
}

public Set<object> Encoding
{
  [Action(kind=Probe)] get
  requires IsEnabled("SaveAs");
  { return encoding.selection;}
  [Action] set
  requires IsEnabled("SaveAs");
  { encoding.selection = value;}
}

//actions

[Action]
public void Save (object obj)
requires IsEnabled("SaveAs") && !MyNotepad.Exists(filename);
ensures MyNotepad.fileName == filename && !IsOpen("SaveAs");
{
   //TODO
}

[Action]
public void Save_SaveFile_Replace(object obj)
requires IsEnabled("SaveAs") &&  MyNotepad.Exists(filename);
ensures IsOpen("Replace") ;
{
    AddWindow("Replace", "SaveAs", true);
}

[Action]
public void Replace(string op)
requires IsEnabled("Replace?") && op in set {"yes","no"};
{
  //TODO
  RemoveWindow("Replace?");
}

[Action]
public void SaveAs_Save_Notepad()
requires IsEnabled("SaveAs");
ensures MyNotepad.fileName == fileName && !IsOpen("SaveAs");
{
  //TODO
}
```

Second Level (Open)

```
using WindowManager;
namespace Open;

object filename;
Set<object> saveAsType;
Set<object> encoding;

//properties

public object Filename
{
  [Action(kind=Probe)] get
  requires IsEnabled("Open");
  { return filename;}
  [Action] set
  requires IsEnabled("Open");
  { filename = value;}
}

public Set<object> SaveAsType
{
  [Action(kind=Probe)] get
  requires IsEnabled("Open");
  { return saveAsType.selection;}
  [Action] set
  requires IsEnabled("Open");
  { saveAsType.selection = value;}
}

public Set<object> Encoding
{
  [Action(kind=Probe)] get
  requires IsEnabled("Open");
  { return encoding.selection;}
  [Action] set
  requires IsEnabled("Open");
  { encoding.selection = value;}
}

//actions

[Action]
public void Open (object obj)
requires IsEnabled("Open") && MyNotepad.Exists(filename);
ensures MyNotepad.fileName == filename && !IsOpen("Open");
{
  //TODO
}

[Action]
public void Open_File_OpenError(object obj)
requires IsEnabled("Open") && MyNotepad.Exists(filename);
ensures IsOpen("Open");
{
   AddWindow("OpenError", "Open", true);
}
```

```
[Action]
public void OpenErrorMsgBox()
requires IsEnabled("OpenError");
ensures !IsOpen("OpenError");
{
  Ack(); //TODO
  RemoveWindow("OpenError");
}

[Action]
public void Open_Cancel_Notepad()
requires IsEnabled("Open");
ensures !IsOpen("Open");
{
  //TODO
}

[Action]
public void Open_Open_Notepad()
requires IsEnabled("Open");
ensures
MyNotepad.Exists(filename) => (fileName == MyNotepad.fileName &&
!IsOpen("Open"));
{
  //TODO
}
```

Second Level (Notepad)

```
using WindowManager;
namespace Notepad;

object text;
Set<object> menu;
Set<object> contextmenu;

//properties

public object Text
{
  [Action(kind=Probe)] get
  requires IsEnabled("Notepad");
  { return text;}
  [Action] set
  requires IsEnabled("Notepad");
  { text = value;}
}

public Set<object> Menu
{
  [Action(kind=Probe)] get
  requires IsEnabled("Notepad");
  { return menu.selection;}
  [Action] set
  requires IsEnabled("Notepad");
  {
    CallActionMenu(value);
    menu.selection = value;
  }
}
```

```
public Set<object> ContextMenu
{
  [Action(kind=Probe)] get
  requires IsEnabled("Notepad");
  { return contextmenu.selection;}
  [Action] set
  requires IsEnabled("Notepad")
  { CallActionContextMenu(value);
    contextmenu.selection = value;
  }
}
```