Universidade Vigo

Departamento de Informática

Complex Components Abstraction to Simplify a Visual Interface Specification

Thesis

For obtaining the academic degree of Doctor

Pedro Miguel Teixeira Faria

Supervisor: Prof. Dr. Javier Rodeiro Iglesias

October 2014

Complex Components Abstraction to Simplify a Visual Interface Specification

This work was partially funded by the Portuguese Government, through PROTEC program, supervised by FCT – Foundation for Science and Technology, under the grant with reference SFRH/PROTEC/49496/2009.

À Isabel

À Filipa, ao Dinis e ao João

Agradecimentos

Quero agradecer a todos aqueles que contribuíram para a realização deste trabalho.

Começo por agradecer a imensa disponibilidade do orientador desta investigação, o Professor Javier Rodeiro Iglesias, pelo incansável apoio, incentivo e acompanhamento próximo que me deu, sempre disponível para discutir qualquer dúvida que surgisse. Um reconhecido agradecimento pelas suas ideias e sugestões, as quais contribuíram grandemente para o êxito alcançado pela realização deste trabalho.

À Universidade de Vigo e aos professores do Departamento de Informática, pela disponibilidade que demonstraram logo no primeiro ano de estudos. Em particular, agradeço ao Professor Jacinto Dacosta, o apoio e acompanhamento que prestou.

À Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Viana do Castelo, pelas condições e recursos disponibilizados para que esta investigação se concretizasse.

Aos meus colegas de trabalho, Pedro, Luís, Paula e João, por todo o apoio ao longo desta etapa e pela sensação de segurança transmitida aquando das minhas ausências. Em especial ao Pedro, pela amizade e pela disponibilidade em alturas de maior aperto.

À minha família e amigos pelo constante incentivo e por compreenderem a menor disponibilidade de tempo que tive para convivermos.

À Filipa, ao Dinis e ao João, pelo tempo em que não estive presente.

À Isabel!

Abstract

The term user interface (UI) is most of the times associated to what a user see and interact with. Nowadays, there are a huge variety of user interface types and interaction styles and, therefore it emerges a demand of specific tools and methodologies to support the design process.

Along the years, several researchers have proposed a multitude of specification methods enabling the representation of user interfaces. However, there is a lack in specification methods that consider at once: to allow specifying not only the interface components (usually known as *widgets*) but concomitantly use those components to represent the interface as a whole.

This dissertation addresses the specification and representation of interactive GUI (Graphical User Interface) prototypes, based on direct manipulation interaction style, using interactive visual components. Specifically, this thesis contributes to the formalization process of interface complex components, from non-functional prototypes with free design issues.

The conducted research has a UI designer centred perspective, since the interface designer can freely establish the interface functionality, including the visual elements to be used, independently of any platform or programming environment. To overcome the identified limitation of current methods, a new *complex component* concept is introduced, allowing increasing the amount of abstraction in the user interfaces construction process.

In order to exemplify, compare and validate the presented methods, a test bed user interface of a simple 2D game is presented. This interface is originally represented by simple components using a well-established methodology. From there, it was possible to verify the *complex components* abstraction, obtained from simple components and thus, to contribute to reduce the complexity of the interface design process. An algorithm that allows the identification of *complex components* from state diagrams, representing the UI functionality, is introduced. As a result of this algorithm the interface representation becomes simpler as it reduces the number of represented states and transitions. *Complex components* reusability is also demonstrated to contribute to simplify the design of other new UIs.

This thesis also introduces a new specification language and an automated prototyping system, supporting the new *complex component* concept, and consequently enabling the specification of complete and interactive visual user interfaces.

Comparing to existent approaches, the proposed *complex component*, together with the developed specification language, have a potentially wider application as they enable a more expeditious specification of complete visual interfaces. Furthermore, they enable to use interface components independently of platforms and/or predefined widgets toolkits (which usually merely allows the presentation manipulation).

Sumário

O termo interface de utilizador é na maioria das vezes associado com aquilo que um utilizador vê e interage. Actualmente existe uma enorme variedade de tipos de interfaces e estilos de interacção e, como tal, uma procura de ferramentas específicas e metodologias que suportem o processo de design.

Ao longo do tempo, diversos investigadores propuseram uma variedade de métodos de especificação que permitem a representação de interfaces de utilizador. No entanto, existe uma falta de métodos que considerem de uma só vez a possibilidade de: especificar não apenas os componentes de uma interface (normalmente conhecidos como *widgets*) mas simultaneamente utilizar esses componentes para representar a interface como um todo.

Esta dissertação aborda a especificação e representação de protótipos de GUIs (Graphical User Interfaces), baseados no estilo de interação de manipulação directa, utilizando componentes visuais interactivos. Especificamente, esta tese contribui para o processo de formalização de componentes complexos de interface, a partir de protótipos não-funcionais suportando desenho livre.

A investigação realizada tem uma perspectiva centrada no designer da interface, uma vez que este poderá estabelecer o seu funcionamento de modo livre, incluindo os elementos visuais a utilizar, independentemente de qualquer plataforma ou ambiente de programação. De modo a ultrapassar a limitação identificada nos métodos actuais, foi introduzido um novo conceito de *componente complexo*, permitindo aumentar o nível de abstracção no processo de construção de interfaces de utilizador.

De modo a exemplificar, comparar e validar métodos apresentados, foi estabelecida uma interface de um jogo simples em 2D. Esta interface é originalmente representada através de componentes simples, usando uma metodologia devidamente estabelecida. Foi possível verificar a abstracção de *componentes complexos*, obtida a partir de componentes simples e desse modo, contribuir para reduzir a complexidade do processo de design da interface. Foi introduzido um algoritmo que permite a identificação de componentes complexos, a partir de diagramas de estado, representando a funcionalidade da interface. Como resultado deste algoritmo, a representação da interface torna-se mais simples, uma vez que o algoritmo reduz o número de estados e transições representados. É também demonstrada a contribuição para simplificar o desenho de novas interfaces, através da propriedade de reutilização dos *componentes complexos*.

Esta tese introduz também uma nova linguagem de especificação e um sistema automático de prototipagem, suportando o novo conceito de *componente complexo*, e consequentemente permitindo a especificação de interfaces visuais interactivas completas.

Comparativamente com abordagens existentes, o *componente complexo* proposto, juntamente com a linguagem de especificação desenvolvida, têm um amplo potencial de aplicação, uma vez que permitem uma especificação mais expedita de interfaces visuais completas. Adicionalmente, permitem usar componentes de interface independentes de plataformas e/ou *toolkits* de *widgets* pré-definidos (os quais normalmente apenas permitam a manipulação da apresentação).

Resumo

El 90% de la información que recibe el cerebro humano es visual y el término User Interface (UI) se refiere la mayoría de la veces a lo que el usuario ve y aquello con lo que interactúa. Negroponte en 1994 propuso una definición simple para el interfaz : “interfaz es donde las personas y los bits se encuentran”. Sin embargo, la simplicidad de esta definición no tiene en consideración otro elemento : la aplicación. Una descripción aún simple pero mas completa la propuso Silva en 2000: “un interfaz de usuario transmite la salida de la aplicación hacia el usuario y la entrada del usuario a la aplicación”. También se puede ver la interacción como un diálogo entre el ordenador y el usuario, y la elección del estilo de interacción a utilizar puede tener un profundo efecto en la naturaleza de ese diálogo. Shneiderman en 1998 clasificó cinco estilos primarios de interacción: Manipulación Directa, Menús de Selección, Formularios, Lenguaje de Comandos y Lenguaje Natural. Dix en 2004 definió estilos de interacción completando o solapando algunos de los anteriores e introduciendo otros cuatro estilos en su clasificación: Question/answer and query dialogs, WIMP (Windows, Icons, Menus and Pointers), Point-and-click e Interfaces 3D. De acuerdo con este autor, está creciendo el uso de técnicas de Realidad Virtual y Sistemas de Visualización de Información, donde el usuario puede moverse en un entorno simulado en tres dimensiones.

Un interfaz de usuario puede combinar uno o más estilos de interacción. Rogers en 2011 determinó un conjunto de términos para clasificar e identificar el interfaz de usuario con el cual el usuario interactúa para controlar una aplicación. Algunas veces, la mayor dificultad es decidir que tipo de interfaz es el más adecuado para una aplicación concreta. Hoy en día, la existencia de múltiples plataformas (MS-Windows, X-Windows, Mac OS, Windows Phone, Android, iOs) ha elevado el interés sobre los modelos de interfaces de usuario que están orientados a la creación de interfaces de usuario multiplataforma, multicontexto y multimodales.

Habitualmente, un interfaz gráfico de usuario (GUI en inglés) es un interfaz de usuario que combina un estilo de interacción WIMP con algún otro. Una de las principales razones por los que los GUIs han llegado a ser tan populares es que son usados por aplicaciones independientes de la plataforma. Como consecuencia de ello, los desarrolladores pueden contruir aplicaciones en la capa mas alta de una arquitectura basada en eventos consistente, mediante librerías de widgets de uso común, proporcionando a los usuarios finales interfaces de usuario con curvas de aprendizaje sencillas, fáciles de usar y manteniendo el

conocimiento obtenido utilizando una aplicación al pasar a otra utilizando la ventaja de la consistencia de los look and feel. El look and feel es un término que abarca aspectos del diseño del interfaz, como por ejemplo: colores, formas, composición y fuentes que representan el look. La estética del interfaz de usuario ha sido estudiada por autores como Bodar en 1994, Tractinsky en 1997 y por Ngo en 2001 y 2002. El feel se considera como el comportamiento de los elementos dinámicos de la interfaz como los botones, contenedores y menús.

Usualmente, las grandes compañias de software crean sus propias guías de estilo o estándares de diseño. Smith en 1996, introdujó que "un estándar es un requerimiento, regla o recomendación basada en prácticas y principios probados. Representa un consenso de un grupo de profesionales acreditados oficialmente, ya sea de ámbito local, nacional o internacional". De la misma forma, van Dam en 1997 introdujo el concepto de interfaces post-WIMP (interfaces de usuario, en su mayoría GUIs, que intentan ir más allá del paradigma WIMP). Consideró que un interfaz post-WIMP contiene al menos un estilo de interacción no dependiente de widgets clásicos 2D, y que tiende a involucrar todos los sentidos humanos, comunicación en lenguaje natural y múltiples usuarios. Ofreció el ejemplo de un interfaz gestual utilizando lápiz, que a la vez, con mayor o menos éxito mezcla técnicas WIMP con post-WIMP para tareas en 2D.

Se establece cada vez más una relación muy cercana entre el concepto de interfaz de usuario gráfico y el concepto de sistema interactivo. Sin embargo, los usuarios no usan sistemas interactivos, sino que usan los interfaces de los que estos disponen. Los sistemas interactivos son responsables de establecer la comunicación entre el usuario y los dispositivos físicos. Por ello, el éxito de una aplicación interactiva depende de la aceptación de los interfaces por parte de los usuarios.

El proceso de creación de prototipos de interfaces de usuario es el resultado, en la mayoría de las ocasiones, del trabajo de muchas personas. Un analista centra su trabajo en las etapas más tempranas de la creación del interfaz de usuario, centrándose en el dominio del cliente. Un diseñador gráfico crea la parte gráfica del interfaz de usuario. Un diseñador de interfaces ha de considerar la interacción del usuario con el interfaz, además de unir la interacción con el diseño visual del interfaz. El desarrollador crea y gestiona el diálogo de la aplicación, que no es otra cosa que la interacción definida anteriormente. Finalmente, el interfaz visual final (el prototipo del interfaz completo) es analizado por el ingeniero de usabilidad, que está interesado en el comportamiento global del prototipo de interfaz, por que tiene que analizar todos los estados visuales resultantes de los tres elementos

fundamentales de un interfaz de usuario: la presentación visual, la composición de componentes de interfaz y el comportamiento de la interfaz.

Una representación del interfaz de usuario puede llegar a ser difícil de implementar, en muchos casos debido a la dependencia de librería de componentes de intefaz predefinidos, que no permiten la personalización del interfaz de usuario de la forma que el diseñador de interfaces desearía. Es importante para un diseñador de interfaces tener la posibilidad de diseñar libremente componentes de manera que pueda usarlos para implementar prototipos de interfaz de usuario completos de forma sencilla.

Esta tesis se centra en la especificación de prototipos de interfaces graficos interactivos, basados en un estilo de interacción de manipulación directa y utilizando componentes gráficos interactivos. Especificamente, esta tesis busca la formalización de componentes complejos de interfaz, desde prototipos no funcionales que tengan un diseño libre, no basado en componentes predefinidos desde librerias o widgets. Esta propuesta está centrada en la visión del diseñador de interfaces debido a que la funcionalidad de la interfaz debe poder ser establecida libremente por el diseñador de interfaces, haciendo uso de los elementos visuales que desee, de forma independiente a cualquier entorno de programación o plataforma software.

Se han definido por ello un conjunto de restricciones que un método de especificación de prototipado de interfaces de usuario basado en componentes debería cumplir como son:

- Debe permitir no solo la especificación de los componentes del interfaz de forma individual e independiente de los otros componentes, sino la especificación del interfaz como un todo a través del uso combinado de los componentes que lo componen.
- Debe soportar tres ámbitos de especificación de los componentes del interfaz:
  - La presentación visual de cada componente (a nivel de los estados visuales que este componente puede tener en el caso de ser un componente complejo) que posibilite la presentación visual del interfaz de usuario global
  - La composición de componentes, entendido como la composición topológica de los componentes para conformar el interfaz de usuario y que está directamente relacionada con una composición jerárquica de los componentes del interfaz de usuario.
  - El comportamiento del interfaz de usuario como un todo, de manera que el diálogo que se establece para cada componente del interfaz como su relación

consigo mismo o con otros componentes del interfaz determinen el comportamiento del interfaz de usuario global.

- Debe ser independiente de plataforma software.

Se ha asignado un acrónimo para estas restricciones para una mejor legibilidad del texto y para un identificación más exacta de los criterios a los que se refiere. Este acrónimo es *CFFI* y su correspondencia con los criterios es la siguiente:

- Criterio 1 : Complete
- Criterio 2: Funtional
- Criterio 3: Free
- Criterio 4: Independent

Normalmente, un interfaz de usuario se genera partiendo de una especificación de alto nivel, seguido de una selección de objetos de interacción apropiados y la composición de los objetos seleccionados (basados principalmente en especificaciones del modelo de datos). Es posible representar un interfaz usando varios modelos de especificación que se pueden encontrar en la literatura (por ejemplo, modelos de especificación, herramientas, lenguajes). Sin embargo, no se encuentran métodos que permitan la especificación de componentes de interfaz (widgets) no dependientes de librerías al tiempo que consideren el interfaz como un todo. No se contempla la posibilidad de considerar el interfaz de usuario como un macrocomponente que a su vez tiene componentes. Para abordar una solución a esta limitación, se propone un nuevo método en esta tesis. Se ha propuesto el concepto de componente complejo. El componente complejo soporta presentación visual (todos los estados visuales que un componente puede tener estan presentes en la visualización de algún estado global del interfaz total), composición de componentes y diálogo entre componentes en reacción a la interfacción del usuario. Se propone además, unido a lo anterior, la definición de un método que permita el uso del componente complejo de manera que se pueda simplificar el proceso de especificación de un interfaz de usuario visual.

Para lograr ambos objetivos se han desglosado en subobjetivos que permitan un proceso metódico en la investigación. Primeramente se han identificado y comparado los métodos existentes en la literatura que permitan crear prototipos, intefaces o componentes centrandose principalmente en como soportan la presentación visual, la composición de componentes y el comportamiento del interfaz (tanto entre componentes como con la interfaz completa). Se ha analizado si son dependientes de librerias de uso común o

plataformas de desarrollo o por el contrario permiten el diseño libre de componentes o del interfaz. Se ha establecido que aunque existen métodos que permiten parte de las restricciones indicadas en el estudio no hay ninguno que cumpla todas las planteadas.

Con objeto de poder realizar verificaciones de resultados se ha especificado un interfaz con funcionalidades variadas basado en componentes simples. De esta manera este interfaz test sirve como elemento de prueba de los conceptos y procesos desarrollados en esta tesis y queda también como un test para futuras investigaciones en este campo tanto por parte de este equipo de investigación como de cualquier otro que desee testear los métodos propuestos como nuevos métodos desarrollados.

Se ha propuesto el concepto de componente complejo como un componente del interfaz de usuario que cumple las restricciones *CFFI* . Se ha propuesto este concepto debido a que los actuales métodos de especificación de interfaces basado en componentes simples son muy complejos en su definición. La complejidad de la especificación aumenta exponencialmente según aumenta el número de componentes de la interfaz. Por ello, se ha planteado una aproximación a la especificación de componentes del interfaz de usuario intentando aprovechar las ventajas que el paradigma de orientación a objetos posee. El uso de componentes complejos que mantienen intrinsecamente en su especificación la definición de sus estados internos, los elementos de interacción hacia otros componentes y la interacción que reciben desde otros componentes, hace mucho mas simple la especificación de interfaces de usuario completos al reaprovechar toda o parte de la lógica de componentes previamente creados. Se ha realizado una caracterización del componente complejo de manera que se han establecido las características que este debe poseer para poder servir como mecanismo de especificación de un componente de interfaz. Estas características se han obtenido como primera fuente de toda la revisión bibliográfica realizada, identificando características comunes en los métodos de especificación estudiados y que eran concondantes con las restricciones *CFFI*. Se han identificado también características de los Abstract Interaction Objects que son métodos de especificación mas centrados en los componentes del interfaz de usuario. Y por último se han identificado características provenientes del paradigma de orientación a objetos para poder aprovechar las ventajas de estos, que estan plenamente probadas y verificadas en la Ingeniería de Software tanto a nivel de diseño como de implementación. Las características mas relevantes identificadas corresponden con la especificación de eventos propios del componentes complejo, que determinan su comportamiento interno y su comportamiento con otro componentes, y que vienen derivados de los eventos del usuario sobre los

componentes del interfaz de usuario. Se han identificado los *Self Events* (*SEs*) que corresponden con eventos que estan encapsulados dentro del componente complejo y que se activan como resultado de un evento de usuario. Los *Self Events* solo se aplican sobre elementos que están contenidos en el componente complejo y que varían el estado global del componente complejo por la modificación de sus elementos internos. Se han identificado también los *Delegate Events* (*DEs*) que tienen una funcionalidad similar a los *SEs*, pero son activados desde otro componente complejo. Otra caracteristica asociada a eventos son las *Delegate Actions* (*DAs*) que corresponden dentro de un componente complejo con la invocación de un *DE* de otro componente complejo del interfaz de usuario. Por último se han identificado los *Application Events* (*AEs*) que son eventos del componente complejo que se relacionan con el core de la aplicación que gestiona el interfaz de usuario. Relacionados con los eventos del componente complejo estan también los estados visuales resultado de la ocurrencia de un evento propio de cualquier tipo del componente complejo. La ocurrencia de los eventos propios de un componente complejo dan lugar a Transciones Visuales y pueden ser representados mediante diagramas de estados propios del componente complejo.

Se han propuesto dos mecanismos de identificación semiautomáticos de componentes complejos. Uno de ellos a partir de especificaciones de interfaces de usuario con componentes simples y otro a partir del diagrama de estados del interfaz de usuario. En el primer de los métodos se introduce su utilización mediante un proceso iterativo en cual se parte de componentes simples, los eventos de usuario y las condiciones que cambian los componentes simples en el interfaz de usuario. El proceso se basa en un agrupamiento progresivo de componentes simples/complejos en componentes complejos en cada iteración que se realiza. Una iteración para abstraer un componente complejo tiene tres criterios: 1) un componente complejo debe tener internamente componentes simples/complejos y los *Self Events* consecuencia de un evento de usuario; 2) un componente complejo identificado tiene relaciones con otros componentes complejos a través de sus *DE* o *DA* que no está totalmente identificados; y 3) si el interfaz no es totalmente funcional y alguno de los componentes complejos aún tienen *DE*/*DA* o alguna otra condición no satisfecha, el nivel de abstracción en componentes complejos puede ser incrementado si se siguen agrupando componentes. Se ha realizado un test de este proceso sobre el intefaz test previamente definido. Como consecuencia del estudio y test de este proceso se ha descubierto que no es posible representar un interfaz de usuario completo y funcional con un componente complejo, como podría ser el resultado óptimo de este

proceso, a consecuencia del tercer criterio de iteración. Por ello, se ha introducido el concepto de Componente Complejo Completo, que representa un componente complejo de interfaz de usuario totalmente independiente de cualquier otro componente del interfaz de usuario.

El segundo proceso de identificación de componentes complejos se basa en un proceso iterativo en el cual, a partir del diagrama de estados y transiciones de estados visuales de un interfaz de usuario, se hace un refinamiento progresivo utilizando los componentes y los eventos utilizados sobre ellos para las transiciones de manera que se obtiene en cada iteración una agrupación de componentes junto con sus transiciones internas. Se repite este proceso hasta que no pueden agruparse mas componentes. La solución básica para el agrupamiento está en el concepto de Estado Equivalente, que está basado en el concepto más extendido de Clase de Equivalencia. Otro concepto necesario para este proceso es el de estado estable, un estado estable es aquel que una vez que ha sido alcanzado, no existe ningún evento que lo lleve a un estado previo. Un estado equivalente es una agrupación de estados que tienen un comportamiento consistente. Para crear un estado equivalente debe asumirse que existe una relación cíclica entre los estados que lo componen y además al menos uno de ellos es un estado estable. Se ha ejecutado este proceso sobre el interfaz test como mecanismo de validación del proceso. La simplificación por la utilización de componentes se pone de manifiesto en este proceso dado que por cada iteración de agrupamiento de estados del diagrama de estados de la interfaz, este diagrama se simplifica al quedar los estados intermedios no estables internos al contexto de funcionamiento del componente complejo y solo quedan visibles aquellas estados y transiciones que se relacionan con otros componentes del interfaz.

Se ha establecido la posibilidad de utilizar como en el caso de las clases en el paradigma de orientación a objetos la característica de reutilización de componentes complejos. Para ello, y dada la característica de presentación visual de los componentes complejos que los dota de semántica, se ha establecido el concepto de reutilización de componentes complejos de interfaz de usuario a nivel semántico y nivel funcional. La reutilización de componentes complejos a nivel semántico se puede obtener por la modificación de la presentación visual sin modificar de ninguna manera la funcionalidad del componente complejo. La aproximación de reutilización a nivel semántico está asociada a la propiedad de Polimorfismo del paradigma de Orientación a Objetos. La reutilización de componentes complejos a nivel funcional se realiza por la modificación de la funcionalidad del mismo, al variar los *Self Events*, *Delegate Events*/*Actions* y los *Application Events* pero manteniendo

una parte de la estructura del componente complejo inicial. La aproximación de reutilización a nivel funcional está asociada a la propiedad de Herencia del paradigma de Orientación a Objetos. Se han realizado también demostraciones del uso de las propiedades de Asociación y Agregación del paradigma de Orientación a Objetos sobre los componentes complejos de interfaz que proporcionan nuevos componentes complejos con nuevas funcionalidades con un mecanismo probado y validado previamente.

Como resultado de todas las aportaciones anteriormente mencionadas se ha realizado una especificación XML de componente complejo y una herramienta de implementación de prototipado de intefaz de usuario a partir de la especificación XML denominada UIFD Specification. La especificación XML tiene tres niveles de agrupamiento superior que corresponden con un repositorio donde estan las definiciones canónicas de los componentes complejos y simples que puedan existir de especificación de interfaces de usuario previas, una librería de componentes complejos preparados para ser usados en un interfaz y por último un interfaz en donde estan los componentes complejos de la librería y que permiten construir el prototipo de la interfaz de usuario y donde se puede encontrar los componentes complejos completos. Se ha desarrollado y detallado la sintaxis XML de especificación de todos los niveles de abstracción de componentes de interfaz y se han realizado ejemplos de los mismos. Se ha desarrollado una herramienta que permite generar un prototipo de interfaz de usuario a partir de la especificación de la interfaz basada en componentes complejos del interfaz de usuario. Se han especificado sus módulos, su arquitectura, su estructura de datos y la personalización de componentes de interfaz.

Como conclusiones de la tesis se puede indicar que se han introducido un conjunto de conceptos y procesos útiles en la simplificación del proceso de especificación de prototipos de interfaces de usuario, como son el componente complejo de interfaz, el componentes complejo completo de interfaz, los eventos propios y delegados, las acciones delegadas y los eventos de aplicación del componente complejo, mecanismos para la reutilización de componentes desde su especificación, procesos de identificación a diferentes niveles de abstracción de componentes complejos desde los componentes simples y desde los diagramas de estados de interfaz, una especificación XML de componentes complejos y una herramienta de prototipado a partir de especificación UIFD de interfaces de usuario mediante componentes complejos de interfaz. Resultados parciales de esta tesis han sido presentados y sometidos a procesos de revisión siendo aceptados para su difusión.

Acronyms

3DML	3D Markup Language
AAIML	Alternate Abstract Interface Markup Language
ADO	Abstract Data Object
ADV	Abstract Data View
AIO	Abstract Interaction Object
AUI	Abstract User Interface
ACM	Abstract Composition Model
APM	Abstract Presentation Model
AUIL	Abstract User Interface Language
AUIML	Abstract User Interface Markup Language
CC	Complex Component
CCC	Complete Complex Component
CCM	Concrete Composition Model
CCXML	Call Control eXtensible Markup Language
CFFI	Complete Functional Free Independent
CPM	Concrete Presentation Model
CUI	Concrete User Interface
D3ML	Device-Independent Multimodal Markup Language
DA	Delegate Action
DE	Delegate Event
DGAIU	Definición Gráfica Abstracta de la Interfaz de Usuario
DISL	Dialog and Interface Specification Language
DM	Dialog Model
EMMA	Extensible Multimodal Annotation Markup language
FUI	Final User Interface
GIML	Generalized Interface Markup Language
GladeXML	Gnome Project XML Language
GUI	Graphical User Interface
IMML	Interactive Message Modelling Language
InkML	Ink Markup Language
InTml	Interaction Techniques Markup Language
IOG	Interaction Objects Graphs

ISML	Interface Specification Meta-Language
Luxor	XML UI Language Toolkit
MariaXML	Modelbased lAnguage foR Interactive Applications
MDML	Multiple Device Markup Language
MPML	Multimodal Presentation Markup Language
MVC	Model-View-Controller
MXML	Macromedia Flex Markup Language
PlasticML	Abstract Interaction Description Language
RIML	Rendering Independent Markup Language
SC	Simple Component
SE	Self Event
SeescoaXML	Software Engineering for Embedded Systems
SISL	Several Interfaces Single Logic
SSIML	Scene Structure and Integration Modelling Language
SunML	Simple Unified Natural Markup Language
TeresaXML	Transformation Environment for inteRactivE Systems representAtions
UI	User Interface
UIFD	User Interface Free Design
UIML	User Interface Markup Language
UsiXML	User Interface Extensible Markup Language
VHML	Virtual Human Markup Language
VoiceXML	Voice Extensible Markup Language
XAML	Microsoft Extensible Application Markup Language
XDL	XML Interface Description Language
XForms	The neXt generation of web FORMS
XICL	eXtensible User Interface Components Language
XIML	eXtensible Interface Markup Language
XISL	eXtensible Interaction Scenario Language
XML-UIDL	User Interface Description Languages Based in XML
XMMVR	eXtensible Markup Language for Multimodal Interaction – VR Worlds
XUL	XML-based User Interface Language

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scope

The 90% of the transmitted information in the human brain is visual and the term *user interface* (UI) is most of the times associated to what a user see and interact with. Negroponte (1994) proposed an interesting but simple interface definition: "*interface is where people and bits meet*". However, the simplicity of this definition lacks on the consideration of another element beyond the user: the application. Thus, a still simple but more complete definition can be found in (Silva 2000): "*a user interface convey the output of applications and the input from application users*". Also, interaction can be seen as a dialog between the computer and the user and the choice of an interaction style to be used can have a profound effect on the nature of that dialog. Shneiderman (1998) has classified five primary styles: *Direct Manipulation* (Hutchins et al. 1985)*; Menu Selection; Form Fill-In; Command Language; and Natural Language*. Dix et al. (2004) refer interaction styles overlapping/completing some of the previous ones and introduce four other common styles in their classification: *Question/answer and query dialog, WIMP (Windows, Icons, Menus and Pointers); Point-and-click; and 3D Interfaces*. Still according with this author, the use of techniques of *virtual reality* and *information visualization systems* where the user can move about within a simulated 3D world is also growing. A user interface can combine one or more interaction styles and, presently, a huge variety of terms is used to classify/identify the user interface with which a user may interact to control an application as can be observed in

Table 1 (Rogers et al. 2011). Sometimes the main difficulty is to decide which interface type(s) is(are) better suited to support a given application. Today, the existence of multiple platforms (e.g. MS-Windows, X-Windows, Mac OS, Windows Phone, Android, iOS) has raised the interest in user interface models that focuses on creating multi-platform, multi-context and multimodal user interfaces.

Command-based	Mobile	Shareable
WIMP and GUI	Speech	Tangible
Multimedia	Pen	Augmented and Mixed Reality
Virtual Reality	Touch	Wearable
Information Visualization	Air-based Gesture	Robotic
Web	Haptic	Brain-computer
Consumer Electronics and Appliances	Multimodal	–

Table 1: Several user interface types referred by (Rogers et al. 2011).

Usually, a called *Graphical User Interface* (GUI) is an UI that combines the WIMP interaction style with any other. One main reason why GUIs became so popular is because they were introduced as application-independent platforms. As a major consequence, developers could build applications on top of a consistent event-based architecture, using a common toolkit of *widgets* (*wi*ndows ga*dgets*), providing the final users with ease of learning, ease of use, and ease of transfer of knowledge gained from using one application to another, because of consistencies in *look and feel*. *Look and feel* is a term that comprises interface design aspects, such as: colours, shapes, layout and typefaces which represent the *look*. Previously the aesthetic issue has been considered by several authors (Bodart et al. 1994; Ngo & Byrne 2001; Ngo et al. 2002; Tractinsky 1997). The *feel* is represented by the behaviour of dynamic elements such as buttons, boxes and menus. Typically, large software companies create their own style guides or standards. Smith (1996) refers that "*a standard is a requirement, rule or recommendation based on proven principles and practice. Represents a consensus of a group of officially licensed professionals locally, nationally or internationally*". Also, several years ago, van Dam (1997) introduced the concept of *post-WIMP* interfaces (user interfaces, mostly GUIs, which attempt to go beyond the WIMP paradigm). He considered that a post-WIMP interface is one containing at least one interaction style not dependent on classical 2D widgets, but ultimately involving all human senses in parallel, natural language communication and multiple users. He gave the example of a pen-based gesture recognizer which, at that time, more or less successfully melded together WIMP and post-WIMP techniques for 2D tasks. Usually, a close relation between the GUI concept

and the interactive system concept is noted. However, users don't use interactive systems, but use the interfaces provided by them. Interactive systems are responsible for establishing communication between the user and the physical devices. Hence, the success of an interactive application depends on the interfaces at user's disposal.

The process of creating a user interface prototype results, most of the times, from the involvement of several persons. The *system analyst* focuses his work on the early stages of user interface conception (centred on client domain). A *graphical designer* creates the graphical part of a user interface (does not need to consider the *dialog*). The *complete interface designer* has to consider the interaction of the user with the interface (the *dialog* part). He also joins the interaction with the interface visual design. The *developer* manages the user interface dialog. And, the final visual interface (the complete interface prototype) is analysed by the *usability engineer*, who is interested in the global interface prototype, because he has to analyze the global visual states resulting from the three elements (visual presentation, component composition and interface behaviour).

This thesis addresses the specification and representation of interactive GUI prototypes, based on a direct manipulation interaction style, and making use of interactive visual components. Specifically, the focus of this thesis is in the formalization process of interface complex components, from non-functional prototypes with free design issues. The proposed approach has a UI designer centred focus, because interface functionality can be freely established by the interface designer, including the visual elements to be used, which are independent of any platform or programming environment.

Usually, a user interface is generated starting from high-level specifications, followed by automatic selection of appropriate interaction objects and layout of the selected objects (based on data model specifications). It is possible to represent an interface using various specification methods presented in literature (e.g. specification models, tools, languages). However, there is a lack in methods which allow the specification of interface components (e.g. widgets) while considering the user interface as a whole. To overcome this limitation, a new method is proposed in this thesis. For that propose a new concept was introduced: the *complex component*. This new concept supports: visual presentation (all visual states that each component can have, used to render each global interface screen); components composition; and dialog between components in reaction to user interaction.

1.2 Research Goals

The research herein presented follows from the following thesis statement:

The process to design a complete and functional user interface can be simplified through the specification of *complex components*, and at the same time use those components to represent the interface as a whole.

From this thesis statement two main goals of the research work emerge:

- To establish a new *complex component* abstraction concept, and;
- To develop and test a new method that makes use of the novel *complex component* concept in order to simplify the specification process of complete user interfaces.

From these two general goals, more specific goals are intended to achieve, namely:

- To identify and compare methods to create complete functional user interface prototypes. In particular, it will be assessed those methods that support visual presentation, component composition and behaviour of the interface as a whole. Also, it will be verified the support for free design of components and independency of any software platform;
- To build a test bed user interface including a vast set of functional features in order to allow to validate and compare specification methods;
- To fully characterize the new *complex component* concept and to establish a correspondence between the new *complex component* concept with the fully validated object-oriented paradigm;
- From the test bed user interface, to abstract *complex components* in order to validate the proposed method, regarding the simplification of the user interface design process;
- To develop a method to enable the extraction of *complex components* from state diagram representation of a user interface;
- To verify the reusability of *complex components* changing the user interface purpose;
- To define a new XML-based specification language that supports the *complex component* concept and simultaneously allows to specify a complete functional user interface;

– To implement a user interface prototyper able to generate the prototype of a complete and functional user interface from the proposed specification language.

1.3 Structure of the Thesis

The rest of the dissertation is organized as follows:

– **Chapter 2:** This chapter presents key concepts about interactive systems with special focus in visual user interfaces. Several methods to specify user interfaces are surveyed and compared. The surveyed methods were selected according with a established set of criteria: *CFFI* criteria (*Complete*, *Functional*, *Free* and *Independent*). A deepen analysis is presented regarding User Interface Description Languages Based on XML (XML-UIDL) and a more abstract concept such as Abstract Interaction Objects (AIOs).

– **Chapter 3:** This chapter presents the specification of a user interface using one of the previously analysed methods. The user interface represents a 2D game for younger children, which serves as a test bed for validation and comparison purposes.

– **Chapter 4:** The proposed *complex component* concept is introduced in this chapter. The concept is fully characterized concerning visual appearance, components composition and interface behaviour. Based on the *complex component*, a new method to abstract the type of components is presented. An abstraction example is detailed in order to explain the concept. Additionally, the new concept is compared with the OO paradigm.

– **Chapter 5:** This chapter illustrates the utilization of the proposed method to abstract *complex components*. For that purpose, the test bed user interface described in Chapter 3 is used. Additionally, an analysis is described stressing the components, the events, the visual states and the global visual transitions involved.

– **Chapter 6:** This chapter addresses the identification and reuse of *complex components*. In the first part of the chapter, a method to identify complex components state diagrams representations is described and illustrated. The second part is focused on the reuse features provided by complex components usage. Both approaches have a focus on simplifying the design of visual user interfaces.

- **Chapter 7:** Two main contributions of this thesis are presented in this chapter. A new user interface specification method supported in XML technology is proposed. Additionally, the implementation of a visual user interface prototyper is described. The protoyper takes the XML-based specification as input and generates the complete user interface.
- **Chapter 8:** In this chapter a summary of developed work and obtained contributions is presented. Open issues for future research are also envisaged.

Chapter 2

User Interface Specification Methods

2.1 Introduction

A complete user interface representation becomes difficult to implement, in many cases due to the dependency on predesigned user interface components libraries (toolkits) that not allow user interface customization, as a user interface designer would like. It is important to an interface designer to have possibility to freely design components (using a hand-made or a computer technique, allowing to represent a set of drawings/visual elements showing what could be the final visual interface look) in order to use them to implement complete interfaces prototypes. The main focus of this dissertation is concerned in representing interactive GUI prototypes, based on direct manipulation interaction style, using interactive visual components. These components are complex components, meaning to have several possible visual states. With the objective of being possible to specify complete functional user interface prototypes, using complex components, a comprehensive literature survey was conducted in order to identify methods (specification models, tools and languages) which meet the following criteria:

- To allow specifying not only the interface components (e.g. widgets) but the user interface as a whole, through using complex components;
- Supporting three characteristics:
  - *Visual Presentation* of each component (all component visual states) used to render each global interface screen;

  - *Components Composition* (topological composition);
  - *Interface Behaviour* as a whole. The dialog that each component may have (action/reaction to other components or the application) is related to other interface components which have dialog.
- To support Free Design of components. Whether the method is capable of creating figures and exchange them on the visual interface to be created (e.g. by using own primitives) and allow to customize the components' behaviour;
- Be independent of any software platform.

In order to simplify the subsequent reference to introduced criteria, the following acronym (*CFFI*) will be used throughout this dissertation. *CFFI* stands for:

1. Criterion 1 (**C**omplete);
2. Criterion 2 (**F**unctional);
3. Criterion 3 (**F**ree);
4. Criterion 4 (**I**ndependent).

Following, a well established approach to a user interface design process is presented. The chapter follows with a comprehensive analysis of visual user interface specification methods.

2.2 Interface Design Steps

According with the ideal nature of an interactive system, this can and should be represented through a visual formalism: *visual* because is generated, communicated and understood by humans, and *formal* because it will be manipulated, maintained and analysed by computers (Schlungbaum & Elwert 1996). The term *representation* refers to the *visual appearance* decided to be used to represent components on the screen (e.g. what looks like a button, a text box or a video player). The term *specification* usually refers to a declarative model, with well defined and consistent rules to describe the interface appearance and functionality. Concerning an interface design process, it is composed by several steps, which can be represented as models. Quentin & Vanderdonckt (2004) say that "*A model captures some facets of the problem and translates them into specifications*". This idea is reinforced by (Navarre et al. 2009) "*The advantage of using models … it provides designers with abstract constructs, making possible to address user interface design issues at different levels of abstraction*". Different approaches can be used in the interface design

process. Rodeiro (2001) presents one approach describing 8 possible models to be used in the referred process.

Mental Model

Norman (1988) refers that *mental models* are "*models that people have of themselves, others, the environment and the things with which they interact. People create mental models from experience, from training and instruction*". Usually, users' *mental models* results from how a system is designed and implemented based on designers' conceptual models. Preece et al. (1994) suggests that software interfaces must be designed to help users building productive *mental models* of a system. There are common design principles employed to support and influence users' *mental models*, such as: simplicity, familiarity, availability, flexibility, feedback, safety, and affordance (Rogers et al. 2011).

Task Model

The user interfaces' developers recognize that the correct identification of future users and their usual tasks are the most important factors to the success of a future user interface (Meixner & Thiels 2008). A user-centred design (UCD) is a process in which the needs, wants and limitations of end users of a product are given extensive attention. Usually, starts with user tasks analysis, as the process to identify and understand user goals and tasks, the strategies he uses to perform the tasks, the tools he currently uses, any problems he experiences and the changes he would like to see on their tasks and tools. Several approaches can be used to collect that information like surveys, questionnaires, interviews, field usability tests and many other techniques. Then, these tasks could be modelled as a *Task Model* using some notation like (*UML*) or *ConcurTaskTrees* (*CTT*) (Paternò et al. 1997). Subsequently, is possible to use different approaches to generate the *Presentation Model* (explained below). A *User Interface Description Language* (*UIDL*) is one of these approaches and nowadays a growing number of *UIDLs* are based upon XML (as presented later on this chapter) with many success results.

Application Model

According with (Rodeiro 2001) the *Application Model* is available in all interface models and acts as a distinction element between the user interface and the underlying application. This model uses some *agent* concept (sometimes called *interactive object*) which communicates directly with the user, understanding an interactive system as a set of

independent but cooperatives agents. This model is important because it generates a high dependence on the components identification, at the *Abstract Presentation Model* (*APM*) and at the system final implementation. The components definition, available on the *Abstract Presentation Model*, should always be coherent with the components established in the *Application Model*.

Dialog Model

This model is focused in the correct specification of the dialog structure. Sometimes is referred as the *Abstract Dialog Model*. It represents the tasks performed by the user through the interface. Jacob (1986) presents a specification language for direct-manipulation interfaces and is the basis for the *Dialog Model (DM)* proposed by (Rodeiro 2001). It represents the user tasks (performed through the interface) using states[1] and commands[2] that transform states. The system starts by an initial state s_0 and reaches new states $s \in S$ as the result of states transitions produced by the commands. In order to achieve its goals, each task is implemented by all possible commands sequences available, always starting by an initial state s_0. Because of that sequence, it is considered that systems are always sequential.

Presentation Model

The *Presentation Model* (sometimes called the *Architectonic Model*) represents the interactive system structure/architecture, responsible for the appearance, the events, the communication channels *activation* (based on events) and *rendering* (based on states) (Rodeiro 2001). This model main characteristic is the responsibility for the system architecture description, through definition of components which establish the interface and its behaviour. Usually, the *Presentation Model* uses simple components, which with its modular description simplifies complex components composition. It is possible to divide this model in two.

- *Abstract Presentation Model (APM)*
  - The semantics of components does not include implementation details. There is no reference to the visual aspect that each component must have and is totally independent of the future environment where the interface will be implemented.

[1] Each state is composed by several attributes like mouse, window dimensions, etc.
[2] Concrete entries sequences correspondent to user actions.

- *Concrete Presentation Model (CPM)*
  - It contains details about each component implementation and is dependent on the implementation environment.

Rodeiro (2001) notes that the specification and design of an interactive system must provide some way to specify the rendering (the dialog rendering with the user[3] and the semantic rendering between the system objects).

Composition Model

The *Composition Model* allows representing the components spatial position (Rodeiro 2001). It is possible to divide this model in two groups.

- *Abstract Composition Model (ACM)*
  - This subdivided model allows doing an abstract representation of components available in *CPM*, at placing level in display device. With this model, it is possible to obtain the components representation, independently of the platform, and to establish the spatial rules between the interface concrete components (obtained previously from *CPM*) and its behaviour. It represents the components topological placing inside the final user interface (on an abstract way).
- *Concrete Composition Model (CCM)*
  - This model includes concrete values needed to place each component in the final interface. For example, if a component is centred (has the abstract centred property established in the *ACM*) it is possible to calculate its actual position, from the size of its container component. This model represents the spatial placing of components, described in *CPM*, over the display device.

3 Rendering available to inform the user about the evolution of interaction that is happening.

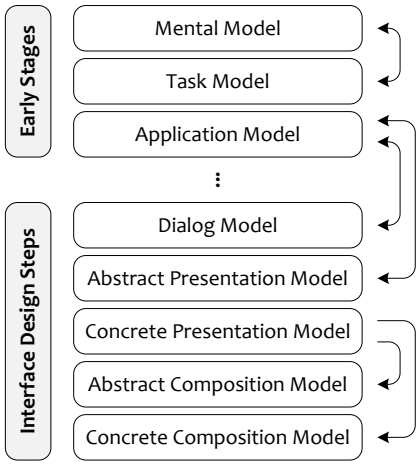

Figure 1: Interface Design Steps proposed by (Rodeiro 2001).

In Figure 1, the arrows represented between the *Early Stages* and the *Interface Design Steps* have significance. It is possible to observe that the *Task Model* is at a level belonging to the user *Mental Model*. The *Dialog Model* represents the tasks that user performs at the *Application Model* level. The *Abstract Presentation Model* should always be connected with the *Application Model*. It is also noted the relation established between *Abstract Composition Model* and the *Concrete Presentation Model* due to be possible to establish the spatial rules between interface concrete components and its behaviour. The specific placement of components on a display device is represented in the *Concrete Composition Model*. These components are represented in the *Concrete Presentation Model*.

After the *Interface Design Steps* have been characterized, it were divided in two groups, depicted in Figure 1. The three steps indicated as "*early stages*" correspond to the requirements analysis stage while the other 5 *interface design steps* are those directly related with the process of user interface specification.

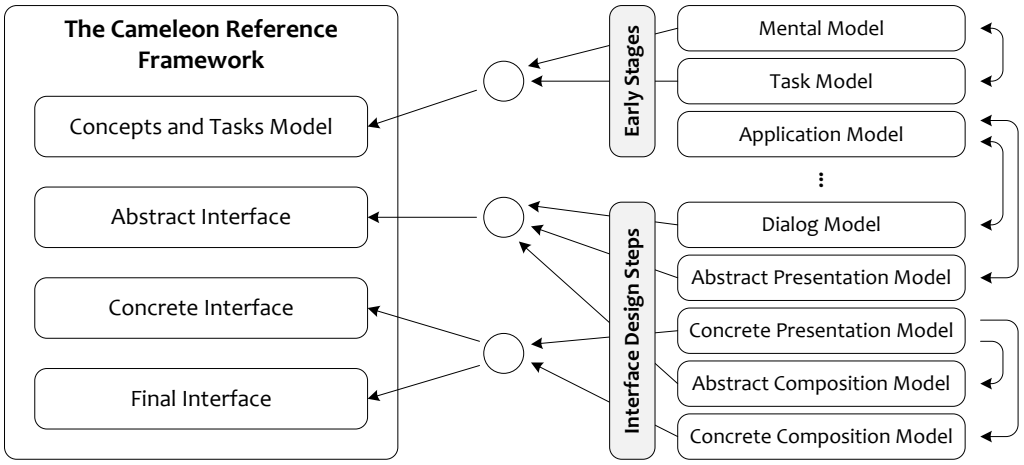

Figure 2: Relation between *Cameleon Reference Framework* and *Interface Design Steps*.

Meanwhile, the HCI software engineering community has developed a refinement process that is used as a reference framework for many tools and methods, with four abstraction levels: from a *Task Model*, an *Abstract UI* (*AUI*) is derived, and from there, the *Concrete UI* (*CUI*) and the *Final UI* (*FUI*) are produced for a particular targeted context of use. That reference is the *Cameleon* (***C**ontext **A**ware **M**odelling for **E**nabling and **L**everaging **E**ffective interacti**ON***) represented in left side of Figure 2: the *Cameleon Reference Framework* (Calvary et al. 2005). This framework is the result of *Unifying Reference Framework* evolution (Calvary et al. 2003) which attempted to characterize the models (at both design and run-time phases), the methods and the process involved in developing user interfaces for multiple contexts of use (multi-target user interfaces, sometimes called *Plastic User Interfaces* (Calvary et al. 2003)). It is possible to identify a relationship between the *Cameleon* framework and the *Interface Design Steps* previously introduced (Figure 2). At the highest level, the *Concepts and Tasks Model* brings together the concepts and the tasks descriptions produced by the designers for a particular interactive system, on a particular context of use (considering also the *Mental Model*). The *Application Model* is implicitly incorporated in the object architecture of the system (Calvary et al. 2003). The *Abstract Interface* results from combination of *Presentation* (abstract), *Composition* (abstract), and *Dialog* models. Although a *Concrete Interface* makes explicit the *final interface look and feel*, it stills a mockup that runs only within the multi-target development environment.

2.3 User Interface Specifications

Some bibliographic research on actual tools, models and languages, used to specify visual user interfaces, is presented in following sections. The objective is to identify which methods may support the previously established *CFFI* criteria. Next section starts by identifying some of the most earlier and referred models to specify user interfaces.

2.3.1 Specification Models

It is possible to represent an interface using various specification models presented in literature. Usually, a user interface is generated starting from high-level specifications, followed by automatic selection of appropriate interaction objects and layout of the selected objects (based on data model specifications) (Lauridsen 1995). A UI model represents all

relevant aspects of a user interface, using some type of interface modelling language or notation. Dix et al. (2004) refers to be possible to distinguish three levels of specification detail:

- *Lexical level*: corresponds to the allowed interaction objects (or "words") to be used in the interaction (e.g. icon's shape and keys pressed (word's sound and spelling));
- *Syntactic level*: structures (layout) of interaction objects and order of inputs/outputs allowed;
- *Semantic level*: meaning of the conversation (dialog) concerning its effect/mapping to the core system's functionality (internal data structures).

The Figure 3 represents an overview of several projects concerning interface specification models and the year in which the first version of the work was published. Five of these user interface specification models were selected and a brief analysis is described below.

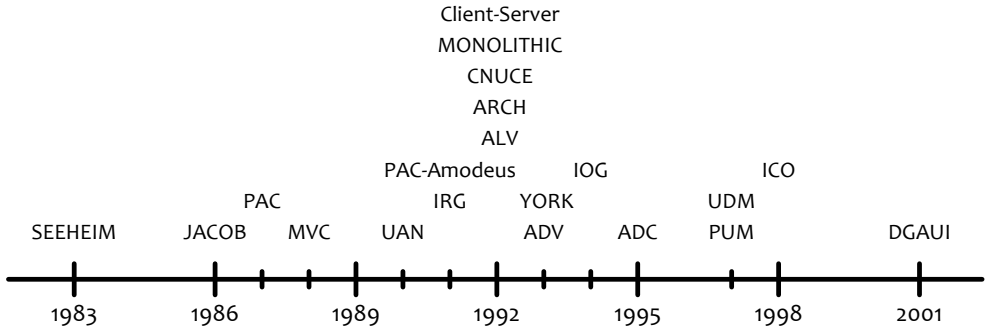

Figure 3: Interface Specification Models evolution.

Seeheim Model

The *Seeheim* model (Figure 4) is one of the earlier user interface architectures, resulted from the first workshop on user interfaces software tools (Seeheim, Germany, 1983). It was proposed in order to separate an application, having a graphical user interface, into three distinct layers: the Presentation Layer, the Dialog Layer, and the Application Layer, as shown in (Figure 4).

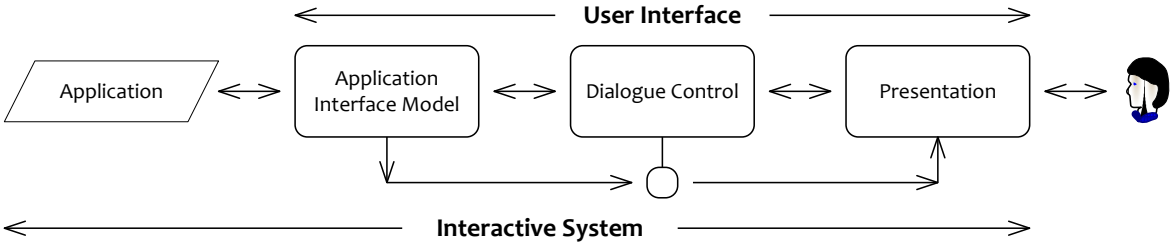

Figure 4: The *Seeheim* model.

According with (Phanouriou 2000) the *Seeheim* model was developed when most application logics were accessed through either *command language* or *form fill-in* interaction styles. Its functional partitioning of the interface assumes synchronous interaction between the user and the interface and does not support asynchronous modes of interaction, such as *direct manipulation*, where system feedback is interleaved with user's input (e.g. as *direct manipulation* computes output values, there are asynchronous callbacks to objects that are implemented in the application, in order to inform it about changes occurred in the visual interface).

Arch/Slinky Model

The *Arch* model (also known as the *Slinky* model) is an evolution of the *Seeheim* model, raising the level of abstraction with which it describes the user interface. The *Arch/Slinky* model separates its components better than *Seeheim*, by dividing one of them in two abstraction levels (Rodeiro 2001). The five components of the model are (Figure 5):

- *Domain Specific Component*: corresponds to the "Application Logic" in the *Seeheim* model;
- *Domain Adapter Component*: corresponds to the "Application Interface Model" in the *Seeheim* model;
- *Dialog Component*: corresponds to the "Dialogue Control" in the *Seeheim* Model;
- The "Presentation" in *Seeheim* model can be subdivided into 2 abstraction levels:
  - *Presentation Component*: provides a set of objects (called *Presentation Objects*) independent of final visual tools. This component acts as a mediator between the *Dialog* and the *Interaction Toolkit* components;
  - *Interaction Toolkit Component*: implements the physical interaction with the user, establishing the so-called *Interaction Objects* (e.g. widgets). These objects are mapped using some formalism, supported by the interaction tools (toolkits).

The *Arch/Slinky* model is a major improvement over the *Seeheim* model. It separates the interface from the application and presentation, and provides the proper abstractions to handle with *direct manipulation* interfaces. However, the *Arch/Slinky* model describes the interface using a particular toolkit (e.g. graphical toolkit). This makes impossible to map the same interface to two devices that are part of different families of devices (e.g. graphical and voice). The entire interface must be redesigned for platforms that have different physical requirements (e.g. small screen or voice-only input). However, as the *Presentation*

Component isolates the toolkit from the interface, the same interface description can be reused across similar platforms (e.g. MS-Windows and X-Windows).

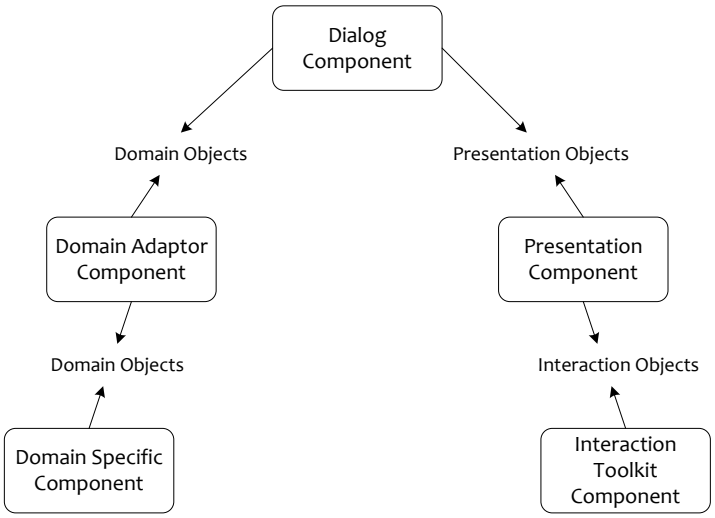

Figure 5: The *Arch/Slinky* model.

Model-View-Controller (MVC) Model

The Model-View-Controller (*MVC*) (Figure 6) was presented with Smalltalk (about 1988) (Krasner & Pope 1988). It is a software architectural pattern for implementing user interfaces and the main idea is to separate the presentation (*View*), the user input handling (*Controller*) and the "semantic" (*Model*). *MVC* divides the responsibilities for a user interface into those three layers. The *Model* layer represents the data structure of the application. The *View* layer accesses the data from the *Model* and draws it on the screen. The *Controller* layer acts as the interface between the *Model*/*View* and the user input. Generically, the standard interaction is:

- The user operates the input device;
- The *Controller* notifies the *Model* to change;
- The *Model* broadcasts change notification to its dependent *Views*;
- The *Views* update the screen (it can query the model).

This model contributed in many aspects to the user interface evolution, since many GUI libraries and application frameworks have adopted the *MVC* as a fundamental design pattern (e.g. Java/JFC, C, C++). According with (Dicker & Cowan 2008) the best modern systems use the event-driven GUI toolkits to satisfy the *MVC* architecture. At the abstract level, *MVC* provides a convenient user interface division.

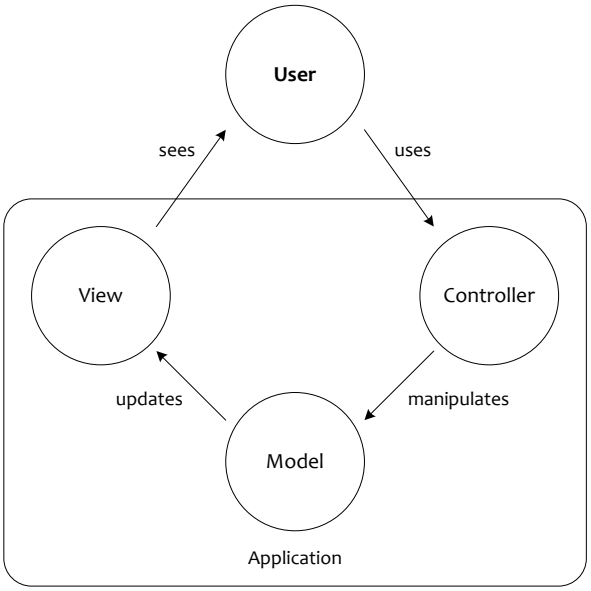

Figure 6: The *MVC* model.

Interaction Objects Graphs (IOG) Model

Typically, a user interface specification model assumes the existence of a set of primitives (interaction objects or widgets) in order that only the presentation (specific to visual interfaces) is able to be managed. The Interaction Objects Graphs (*IOG*), introduced by (Carr 1994), is an approach to the problem of creating new widgets, allowing to define not only its behaviour but also its position on a graphical interface and its concrete spatial relations with other objects.

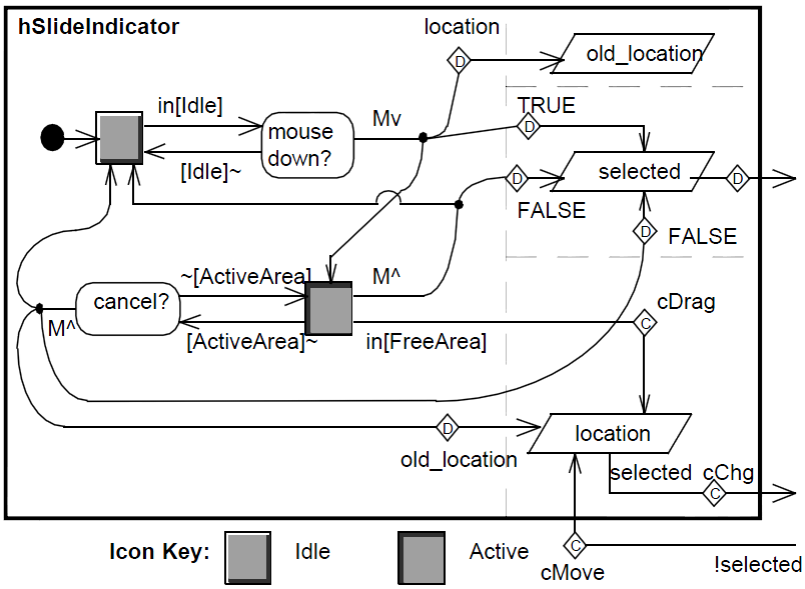

Figure 7: Example of a horizontal slider indicator represented with *IOG* model (Carr 1994).

The *IOG* model is one of the most complete user interface specifications. The IOGs are based on Interface Representation Graphs (IRGs) (Figure 7), and allow to do low level object specification. However, two limitations indicated by the author, concerning the use of this model are: the level of detail required to specify an interactive object grows the specification considerably, becoming it impractical to use. A first solution to this problem could be to specify the attributes affecting only the dynamic behaviour of the IOG and to define the other attributes on a list. Other problem is the need of codification, which could be solved by a tool to edit an IOG and to execute it directly.

DGAIU Model

The *DGAIU* model (*Definición Gráfica Abstracta de la Interfaz de Usuario*) (Rodeiro 2001) (Figure 8) is the one which supports more *Interface Design Steps*.

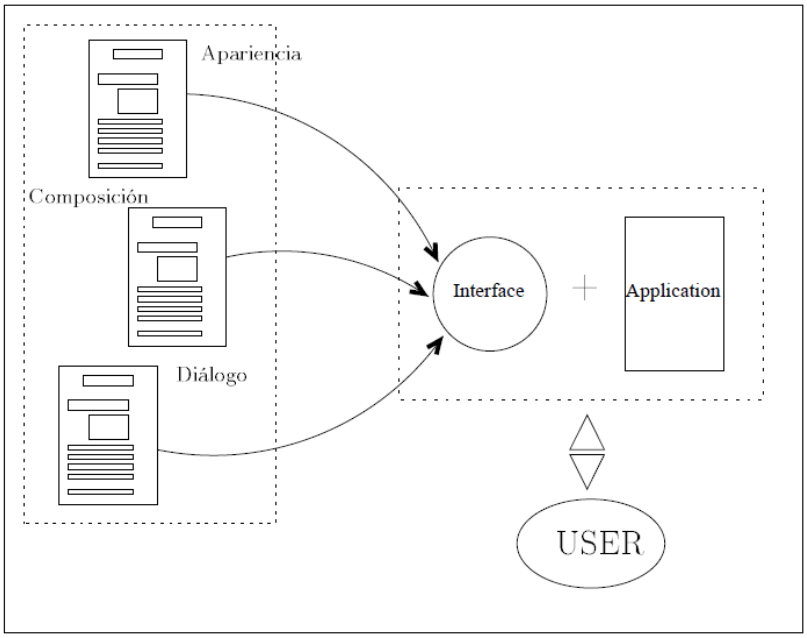

Figure 8: The DGAIU model.

DGAIU considers that a visual user interface is not a continuous structure but is composed by discrete finite elements. Those individual elements are called components and because they follow a hierarchical and topological structure they can be contained inside of each other. In order to specify a user interface, the designer has a representation system with which he may:

- To create new components with its visual appearance;
- To indicate individual components composition to establish the concrete user interface;

– To represent the dialog between components inside the user interface, indicating the events, which each component responds to, and how the user interface responds to them (modifying the interface components or doing application calls).

Comparative Analysis

From (Rodeiro 2001) is possible to verify that most user interface specification models, that support interface abstract representation, do it at behavioural level, and only a few perform an abstraction of the interface concrete representation (at visualization level). Almost all models which incorporate interface concrete representations do it because they incorporate tools to automatically generate interface code. From the list of about twenty models studied by Rodeiro (2001) it was decided to briefly introduce here 5 of them. Two of these models (*Seeheim* and *MVC*) are the most earlier and mentioned in the literature. The other three models were here introduced, due they support more than the APM (Abstract Presentation Model): *Arch*, *DGAIU* and *IOG* models support the CPM (Concrete Presentation Model) and, *DGAIU* and *IOG* support also the CCM (Concrete Composition Model) (Table 2).

Acronym	**Description**	**DM**	**APM**	**CPM**	**ACM**	**CCM**
Seeheim	Seeheim Model	✓	✓			
Arch/Slinky	Arch/Slinky Model	✓	✓	✓		
MVC	Model-View-Controller	✓	✓			
IOG	Interaction Objects Graphs	✓	✓	✓		✓
DGAIU	Definición Gráfica Abstracta de la Interfaz de Usuario	✓	✓	✓	✓	✓

Table 2: Specification models represented by 5 *Interface Design Steps* (Rodeiro 2001).

Still considering the 5 specification models, the information available in Table 2 can be disposed under a second perspective of analysis: the *CFFI* criteria.

		Functional				
	Complete	Visual presentation	Components composition	Interface behaviour	***Free Design***	***Platform Independent***
Seeheim		(✓)				✓
Arch/Slinky		✓				✓
MVC		(✓)				✓
IOG	(✓)	✓	(✓)	✓	✓	✓
DGAIU	(✓)	✓	✓	✓	✓	✓

Table 3: Specification models analysed according with the *CFFI* criteria.

Two of the identified models are indicated as the more complete (DGAIU and IOG). However, the first criterion (**C**omplete) is not totally accomplished because the interface

specification is obtained through the use of simple components. As a consequence, the level of detail required to specify an interface, grows the specification considerably, making it impractical to use. Additionally, considering the second criterion (**F**unctional), IOG doesn't support the ACM (Abstract Composition Model) resulting infeasible the hierarchy of components. The other three models (Seeheim, Arch/Slinky and MVC) don't consider several of the *CFFI* criteria to generate the final interface structure.

The following sections are dedicated to continue identifying methods used to specify user interfaces. The next one is focused in specification tools particularly used with graphical user interfaces.

2.3.2 Specification Tools

During the last decades several tools had appeared, concerning the development of graphical user interfaces (Righetti 2006). Those tools may be classified under one of the following types:

- Prototyping tools:
  - Paper and pencil;
  - Sketching tools/Facade tools;
- Development tools:
  - User interface builders;
  - Model-Based User Interface Development;
  - XML-Based UI Languages.

Prototyping tools

Prototyping is an important technique to reduce the cost and risk involved in developing complex software systems (Szekely 1995). It essentially involves building a small scale version of a complex system in order to acquire critical knowledge required to build the system. Interface prototypes can be built using a large variety of tools, ranging from paper and pencil to draw mockups of displays, to other more complete and sophisticated interface construction toolkits. Following, two categories of tools that can be used to prototype user interfaces will be presented.

Paper and Pencil

Paper and pencil are perhaps the most popular tools used to describe interface designs (Righetti 2006). Under this category is possible also to include electronic versions of these

tools such as drawing and painting tools (sometimes called mockup tools). Paper and pencil are prototyping tools with several strengths such as:

- Easy to use: most people can draw boxes with buttons, menus and scribbles representing the objects in an application domain;
- Allows extensive control over design details: control is not limited by the ability of one person to draw, because he can always draw a scribble and tell others what it means;
- Encourage team design: many people can draw at the same time, especially when drawing on blackboards and large sheets of paper;
- Useful for capturing different kinds of information: whatever is not captured in the drawings can easily be expressed with textual annotations.

The main weaknesses of using paper and pencil are to be very awkward to capture behaviour, and also the interface prototypes not be executable. Behaviour is often represented using two drawings showing the interface before and after an action, together with an annotation of what the action is (e.g. clicking the mouse). However, despite their weaknesses, paper and pencil (or their electronic versions) are invaluable complements to all the other prototyping tools.

Sketching tools/Facade tools

Facade tools are essentially drawing editors with an ability to specify input behaviour. They allow developers to build screens that look and behave like the screens of the real application, except that there is no "application" behind them. One of the earliest examples of these tools is Hypercard (Hypercard 1992). Meanwhile, new tools emerged from academic area (e.g. SILK (Landay 1996) and DENIM (Lin et al. 2002)) as such from commercial area (e.g. ForeUI (ForeUI 2012)). Tools in this category differ mostly in the quality of the drawings (e.g. 3D shapes), the sophistication of the input behaviours that can be specified and also the intended platforms (desktop, web, mobile). The main weakness of sketching/facade tools appears to be that they do not produce reusable code that can be used to build the real application, so the implementation effort in building the prototype is lost. Additionally, since the prototyping and the implementation tools are different, the prototype and the application might be built by different teams of people, and many of the lessons learned while building the prototype remain in the minds of the prototyping team, and may not carry over effectively to the implementation of the system.

Development tools

Other type of tools is concerned with the process of creating concrete user interfaces and is disposed under three main categories.

User interface builders

The user interface builders are popular interface construction tools, which provide to developers a drawing-like interface to specify the interface, and generate executable code that can be linked into an application to produce a strength implementation. These interface builders use a general purpose programming language, such as Java or C++, to specify behaviour, the same language that is often used to implement the application functionality. There are several interface builder tools in the market (e.g. Sun NetBeans (NetBeans 2012), Microsoft Visual Studio (Microsoft Corp 2012)) widely used and reported to greatly facilitate the interface construction process. They provide fast turnaround to changes because of their drawing-like interface and their ability to quickly switch between "build" mode where developers specify the elements of displays and "run" mode where developers can test the interface as if they were end-users. However, a shortcoming for who is designing the visual interface is the obligation to use concrete components (widgets) with the characteristics they own and having some limitations to do a free customization, concerning the components visual appearance and behaviour.

Model-Based User Interface Development

The Model-Based User Interface Development (MBUID) results from the evolution of the UIMS paradigm (Meixner et al. 2011). The term UIMS (User Interface Management System) was used when referring to an interface specification model, and should not be understood as a system, but rather a software architecture (also called a User Interface Architecture) in which the implementation of an application's user interface was clearly separated from that of the application's underlying functionality (Rosenberg et al. 1988). Meixner et al. (2011) indicates several examples of these tools such as ADEPT, TRIDENT or MASTERMIND. It is considered that MBUID is currently in its fourth generation and it is focused on the development of context-sensitive UIs for a variety of different platforms, devices and modalities and the integration of web-applications. Central elements of most of the current approaches are models which are stored (mostly) as XML-based languages to enable easy import and export into authoring tools. Furthermore, one research focus is on the optimization of the automatically generated UIs by ensuring a higher degree of

usability. Today, MBUID approaches are often called “model-driven” and not “model-based” anymore. Model-driven UI development puts models at centre of the development process, which are then (semi-) automatically transformed into an executable code.

XML-Based UI Languages

During the last decade, new specification tools have emerged, with special focus on XML-Based UI Languages, usually known as: User Interface Description Languages Based in XML (*XML-UIDL*). These languages became very popular and it appears to decrease the abstraction level needed to specify user interfaces. Thus, a bibliographic research and a brief survey of several of those XML-UIDLs were made, and a succinct introduction to them is presented in the following section.

2.3.3 User Interface Description Languages Based in XML

To specify user interfaces using XML is considered to be one solution for the standardization and interoperability between applications (Carnero 2007). Also, XML becomes one of the main description languages for all kinds of metadata on the internet. These are the main reasons for the constant new emergence of XML-UIDLs. This generation tools comprise of authoring environments and XML-based languages tools, due to the enormous maintenance versatility, extension capabilities and refinement possibilities of XML-based languages. As referred by (Muller et al. 2001), the XML technology as a common representation standard, allows:

- To specify the abstract interface model;
- To describe specific characteristics for different devices;
- To specify the transformation process from abstract to concrete interaction objects.

A list of XML-UIDLs and its development environments has grown strongly in the last decade and each of them came up with a specific application purpose. In Figure 9, is possible to observe the release year of first versions of several XML-Compliant Languages (drafts in some cases).

With the fast growing number of XML-UIDLs available (Table 4), is possible to verify some kind of new paradigm of use, because of the adoption of these languages to specify user interfaces.

Acronym	**Description**
3DML	3D Markup Language
AAIML	Alternate Abstract Interface Markup Language
AUIL	Abstract User Interface Language
AUIML	Abstract User Interface Markup Language
CCXML	Call Control eXtensible Markup Language
D3ML	Device-Independent Multimodal Markup Language
DISL	Dialog and Interface Specification Language
EMMA	Extensible Multimodal Annotation Markup language
GIML	Generalized Interface Markup Language
GladeXML	Gnome Project XML Language
IMML	Interactive Message Modelling Language
InkML	Ink Markup Language
InTml	Interaction Techniques Markup Language
ISML	Interface Specification Meta-Language
Luxor	XML UI Language Toolkit
MariaXML	Modelbased lAnguage foR Interactive Applications
MDML	Multiple Device Markup Language
MPML	Multimodal Presentation Markup Language
MXML	Macromedia Flex Markup Language
PlasticML	Abstract Interaction Description Language
RIML	Rendering Independent Markup Language
SeescoaXML	Software Engineering for Embedded Systems using a Component-Oriented Approach
SISL	Several Interfaces Single Logic
SSIML	Scene Structure and Integration Modelling Language
SunML	Simple Unified Natural Markup Language
TeresaXML	Transformation Environment for inteRactivE Systems representAtions
UIML	User Interface Markup Language
UsiXML	User Interface Extensible Markup Language
VHML	Virtual Human Markup Language
VoiceXML	Voice Extensible Markup Language
XAML	Microsoft Extensible Application Markup Language
XDL	XML Interface Description Language
XForms	The neXt generation of web FORMS
XICL	eXtensible User Interface Components Language
XIML	eXtensible Interface Markup Language
XISL	eXtensible Interaction Scenario Language
XMMVR	eXtensible Markup Language for Multimodal Interaction with VR Worlds
XUL	XML-based User Interface Language

Table 4: List of some User Interface Description Languages Based in XML (*XML-UIDLs*).

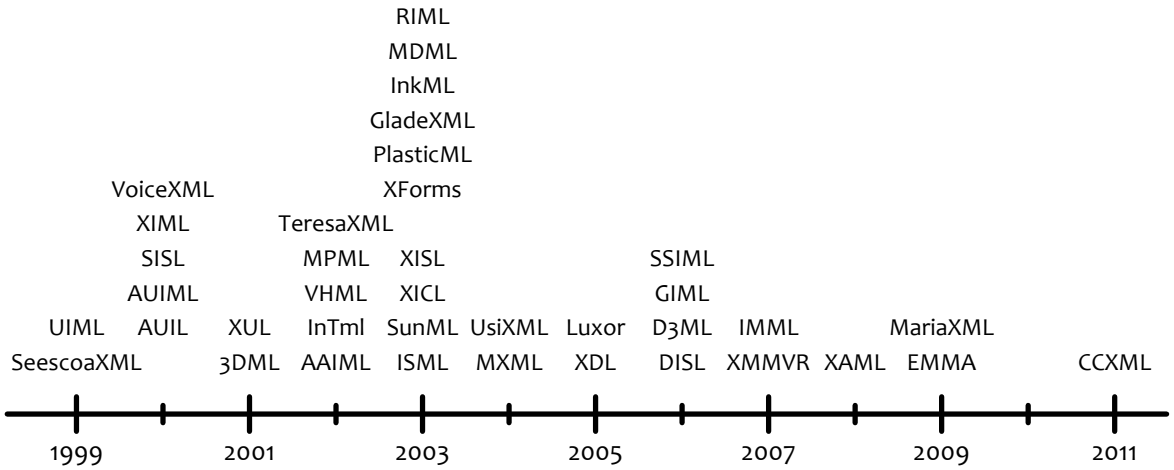

Figure 9: XML-UIDLs evolution (drafts in some cases).

Following, a simple characterization of the identified XML-UIDLs is made, focusing in features that each one owns and the application possibilities they offer.

3DML

This is a markup language that describes desktop-based 3D presentations, virtual reality (VR) and augmented reality (AR) applications (Figueroa et al. 2001a; Figueroa et al. 2001b) (the latest version is from spring 2005). The main objective was to support different types of input and output devices, and 3D interaction techniques. It establishes a way to describe classes of 3D interaction techniques and devices, and it allows developers to combine instances of such classes with VR objects, in order to create applications. 3DML doesn't describe the visual (i.e. geometry and texture), or sound capabilities of VR objects (third-party languages have to be used for such details). It is not executable by its own and also relies on an implementation in a third-party toolkit (e.g. Java3D, VR-Juggler). Later, this language evolved to a new one called InTml (Figueroa et al. 2002).

AAIML

AAIML language is a XML-based, device-independent markup language, sufficiently abstract to accommodate devices that comply with the Universal Remote Console (URC) specification (Gottfried et al. 2002). Wireless technologies are already present for a long time at home, at work, in public places, and on the move. By employing network based technologies, people can remotely control connected services and electronic devices from anywhere, using a wide variety of remote console devices, such as cell phones, tablets, car radios, wrist watches, wearable and other computers. Therefore, a way was sought to

accommodate the diverse user interface needs of access devices, without having to implement a separate UI for each of them. For example, a wrist watch cannot accommodate the same graphically rich UI as a desktop computer, and a UI written for a handheld computer will be different from an audio based UI of a car radio. The AAIML language establishes a variety of interactors for input and output operations. An interactor is mapped to a concrete widget (or combination of widgets) available in the URC platform. For example, the interactor "string-selection" could be rendered as an array of radio buttons on a GUI, and as a voice menu on a voice-based UI platform. It is up to the URC device to decide what mapping patterns to use, and the target does not have to have any knowledge about it. AAIML will implement each code into a different user interface, depending on the final platform complying with the URC specification. All prototypes are implemented mainly in Java, and take advantage of third-party tools for middleware and wireless communication implementations.

AUIL

This XML-based language (Siebelink 2000) was designed with the purpose to describe the generic properties of a given interactive data service, which then automatically adapt the service content, behaviour and presentation towards various device-specific XML languages (e.g. XHTML, WML, VoiceXML). This service adaptation process is based in the user's preferences, his terminal capabilities, his current location or the service requirements. AUIL was designed for authoring a "skeleton service", which mainly concentrates on the behavioural aspect of the service. In particular, a service is divided into several "units", which describe separate dialogs between the user and the service (forms, menus…). AUIL documents may contain explicit content as well, but in view of the desired separation of behaviour, content and presentation, the explicit content should be kept minimal. AUIL contains minimal support for presentation (which is dealt by applying style sheets). The main purpose of this language is to build the application behaviour.

AUIML

AUIML is a XML-UIDL in result of a project (previously called *DRUID*) developed by IBM. It is part of the IBM's internal processes and allows defining the intent (or purpose) of an interaction with a user, instead of focusing on the appearance (Souchon & Vanderdonckt 2003). This means that the designers have to concentrate mainly on the semantics of the interactions. Indeed, AUIML is intended to be independent of any client platform, any

implementation language, and any UI implementation technology. A single intent should run on many devices. A UI is described in terms of manipulated elements (a data model that structures the information required to support a particular interaction), of interaction elements (a presentation model that specifies the look of the UI – choice, group, table, tree) and of actions which allow to describe a micro-dialogue to manage events between the interface and the data. AUIML is comparable to UIML (Souchon & Vanderdonckt 2003) in such that only supports some predefined features of presentation and dialog models and has support to languages like: HTML, DHTML, Java Swing, PalmOS and WML.

CCXML

The CCXML (W3C Recommendation 2011a) is a language designed to complement dialog systems such as VoiceXML (W3C Recommendation 2000). CCXML can provide a complete telephony service application, comprised of Web server CGI compliant application logic, one or more CCXML documents to declare and perform call control actions, and to control one or more dialog applications that perform user media interactions. The CCXML specification originated from the desire to handle call control requirements that were beyond the scope of the VoiceXML specification (e.g. support for multi-party conferencing, with advanced conference and audio control). CCXML and VoiceXML implementations are not mutually dependent. A CCXML implementation may or may not support voice dialogs, or may support dialog languages other than VoiceXML. However, CCXML is a XML-UIDL not designed to support visual interfaces representation.

D3ML

D3ML is an abstract, Web-based user interface description language, primarily intended for adaptation to various concrete markup languages, which can be directly rendered by browsers and comparable client applications (Gobël et al. 2006). D3ML consists of a set of integrated XHTML modules and it was designed as a domain-specific language for modelling device-independence and multimodality in Web-based user interface descriptions, allowing developers to model enough meta-information for adapting output to any useful combination of input and output modalities. This XML-UIDL was also developed in close relation with another specification language: RIML (Dermler et al. 2003).

DISL

The main focus of DISL (Dialog and Interface Specification Language) specification is on mobile interaction devices. The authors (Schaefer et al. 2006) proposed a Multimodal Interface Presentation and Interaction Model (MIPIM). DISL is considered as a modified subset of UIML that extends the language in order to enable generic and modality independent dialog descriptions. It is mainly concerned with generic widgets description and behavioural aspects improvements. Generic widgets are introduced in order to separate the presentation from the structure and behaviour. Further, a DISL rendering engine may use this information to create interface components appropriated to the interaction modality (e.g. graphical, vocal) in which the widget will operate.

EMMA

The W3C Multimodal Interaction Working Group (W3C Recommendation 2009) aims to develop specifications to enable access to the Web, using multimodal interaction. EMMA is a XML-based language that is used to contain and to annotate information, automatically extracted from users input, which manipulates multimodal UIs. The language is able to convey meaning for different types of single input (e.g. text, speech, handwriting) and combinations of any previous modalities. These combinations are compliant with other languages (e.g. InkML, VoiceXML). It was not found reference to user interface presentation or composition models.

GIML

GIML allows describing an interface from the functional point of view, which allows producing final interfaces for any domain. This includes interfaces for non human users (an application that interacts with the developed GITK application through e.g. a generated web service interface) (Kost 2006). The GIML language has been designed to support an abstract interface definition on one side, as well as a continuous reification process towards the final user interface. Some main characteristics are:

- Low number of tags;
- Separation of concerns (structure, style, content and behaviour);
- Model logical relation of interface objects. The hierarchical nature of the language is used to model containment-relations, such as for e.g. label belongs-to widget and widgets are contained in a group. These relations are important for two tasks: the layout generation process and for providing a navigation scheme;

– No validation constraints (as opposed to XForms specification (W3C Recommendation 2012) which considers input validation).

Also, a toolkit was developed: GITK (Generalized Interface ToolKit). The difference to other toolkits (such as GTK+ and QT) is that GITK is a meta-toolkit, a toolkit that maps to other toolkits, meaning that GITK provides abstraction at dialog level. The developer segments the application into tasks. For each task that needs user interaction, a dialog is started and an event-callback mechanism is used to react onto user input. The main difference relies in the way the dialog is specified. GITK can be seen as a "black box" that receives the dialog description and handles all details to produce and run an adapted instance of the dialog. The application is totally decoupled from the resulting interface. The referred "black box" can transform an abstract dialog description into an adapted representation. This in turn will be the source for the final interface. GIML can also be used too with Coin3D library for 3D purposes and GTK+ for 2D purposes.

GladeXML

This language allows to dynamically load user interfaces and is used by Glade Interface Designer, which is an independent GUI builder programming-language (GladeXML 2003). The user interfaces designed in Glade are saved in a XML document, and by using the GtkBuilder GTK+ these can be dynamically loaded by the applications. By using GtkBuilder, Glade XML files can be used in numerous programming languages including C, C++, C#, Java, Perl, Python and others. An object represents the 'instantiation' of an XML interface description. When one of these objects is created, the XML file is read, and the interface is created.

IMML

The author (Leite 2006) refers that software engineering and human-computer interaction communities have approached the development of interactive systems differently, using methods, techniques and tools that are not easily integrated. For instance, it is difficult to construct a functional specification or software architecture from task models, scenarios and user interface prototypes. It is also difficult to generate a user interface prototype directly from software engineering specification. The author considers that the design process could be improved by providing the designer with models, languages and tools which allows integration of both areas. In order to achieve that goal, a language which allows interactive systems specification was proposed: the Interactive

Message Modelling Language (IMML). This language has a vocabulary for reasoning about functional, interactive and communicative purposes of UI elements. The Figure 10 represents an overview how the language is related with different development methods and techniques.

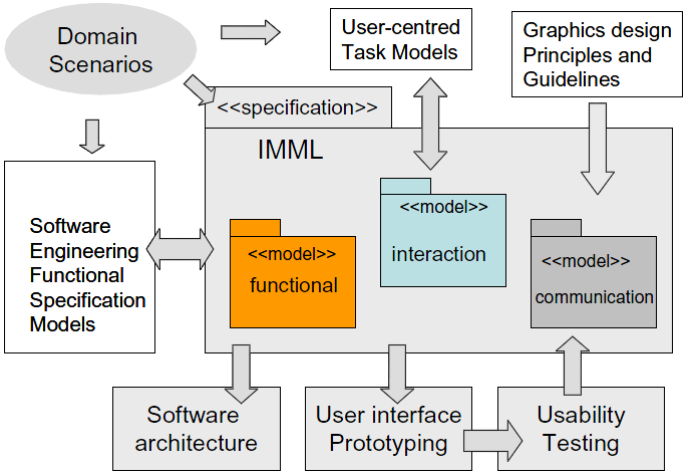

Figure 10: A development process using IMML.

IMML helps to integrate functional, task, dialog and presentation models, emphasizing the communicability aspects of interactive systems without losing control of the interactivity and functionality. In order to help a designer to specify a user interface using IMML, a tool called Visual IMML was developed, which provides a graphical vocabulary to construct the abstract description of a user interface. From that, then it is possible to translate the UI to DHTML code, supporting a browser-based UI for web.

InkML

A pen-based interface is enabled by a device that allows movements of the pen to be captured as digital ink (W3C Recommendation 2003). Digital ink can be passed on to recognition software that will convert the pen input into appropriate computer actions. Alternatively, the handwritten input can be organized into ink documents, notes or messages that can be stored for later retrieval or exchanged through telecommunications means. Hardware and software vendors have typically stored and represented digital ink using proprietary or restrictive formats. The lack of a public and comprehensive digital ink format has severely limited the capture, transmission, processing, and presentation of digital ink across heterogeneous devices developed by multiple vendors. In response to this need, the Ink Markup Language (InkML) provides a simple and platform-neutral data format to promote the interchange of digital ink between software applications. InkML

supports a complete and accurate representation of digital ink. In addition to the pen position over time, InkML allows recording of information about device characteristics and detailed dynamic behaviour to support applications such as handwriting recognition and authentication. InkML also provides features to support rendering of digital ink captured optically to approximate the original appearance. It is not the purpose of InkML to describe and store semantic information, such as the plain text of ink recognized as handwriting. Nor it is a goal of InkML to store the contextual information about the ink, such as what kind of field in a form where ink was written. However, InkML provides means for extension. InkML can include XML from other schemas and embed within other XML documents. A vast number of applications can use the pen as a very convenient and natural form of input, such as: Ink Messaging, Ink and SMIL, Electronic Form-Filling; Pen Input and Multimodal Systems. In its simplest form, an InkML file with its enclosed traces looks like this:

```
Simple InkML specification example.
<ink xmlns="http://www.w3.org/2003/InkML">
   <trace>
      10 0, 9 14, 8 28, 7 42, 6 56, 6 70, 8 84, 8 98, 8 112, 9 126, 10 140,
      13 154, 14 168, 17 182, 18 188, 23 174, 30 160, 38 147, 49 135,
      58 124, 72 121, 77 135, 80 149, 82 163, 84 177, 87 191, 93 205
   </trace>
   <trace>
      130 155, 144 159, 158 160, 170 154, 179 143, 179 129, 166 125,
      152 128, 140 136, 131 149, 126 163, 124 177, 128 190, 137 200,
      150 208, 163 210, 178 208, 192 201, 205 192, 214 180
   </trace>
   <trace>
      227 50, 226 64, 225 78, 227 92, 228 106, 228 120, 229 134,
      230 148, 234 162, 235 176, 238 190, 241 204
   </trace>
   <trace>
      282 45, 281 59, 284 73, 285 87, 287 101, 288 115, 290 129,
      291 143, 294 157, 294 171, 294 185, 296 199, 300 213
   </trace>
   <trace>
      366 130, 359 143, 354 157, 349 171, 352 185, 359 197,
      371 204, 385 205, 398 202, 408 191, 413 177, 413 163,
      405 150, 392 143, 378 141, 365 150
   </trace>
</ink>
```

These traces consist simply of X and Y value pairs, and may look like Figure 11 when rendered:

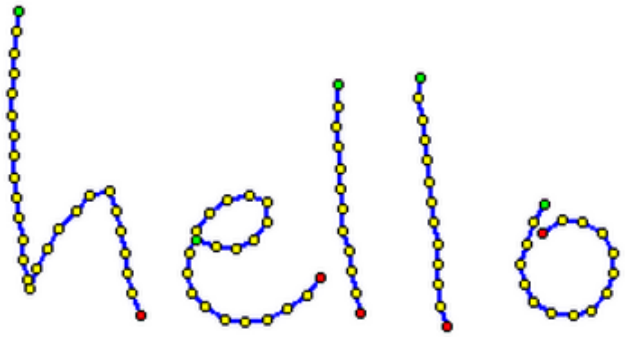

Figure 11: Example of trace rendering.

InTML

This language (Figueroa et al. 2002; Figueroa 2009) is an evolution of a previous language: 3DML (Figueroa et al. 2001a). It allows describing input/output devices, interaction techniques and applications – a combination of all previous elements plus geometry in X3D (Web3D 2011). The InTml language provides an instrument for comparison of 3D interaction techniques (InTs), for the study of those alternatives in an application, and for rapid prototyping. InTml can accelerate the production of VR applications, because its concepts are at a higher level than the ones in languages traditionally used in this area, such as C++ and Java, without the restrictions of devices and interaction techniques of languages such as VRML. InTml also provides a way to hide unnecessary details from designers, such as device configuration and interaction technique implementation, so they can concentrate on the tasks that a VR application does. An IDE was developed in order to support two types of users. Novices can create VR applications by plugging existing filter classes (required functionality in a desired platform), and run them inside the *Eclipse* environment, once such an environment has been properly setup. Experts can define new filter classes in existent filter libraries, and generate code templates for the targeted platforms, in order to ease the development of new implementations. There were three preliminary implementations of InTml: Java, C++ and *ActionScript*.

ISML

ISML (Interface Specification Meta-Language) was developed with the objective that metaphors (indicated by authors as, shared concepts between the user and the computer) be made explicit in design (Crowle & Hole 2003). ISML decouples the metaphor model from any particular implementation, and expresses mappings of the concepts shared between the user and the system. In order such a model becomes useful, ISML provides a framework

that supports mappings between both user-oriented models (such as task descriptions) and software architecture concerns (interactor definitions). The ISML framework composites these concepts within five layers. Each of these layers is supported by computational formalisms, including: communicating objects, state modelling, abstract-to-concrete mappings, event modelling and task models.

Luxor

Luxor (Java XML User Interface Toolkit) is a free, open-source XML UI Language (XUL) toolkit. It was released under the GNU General Public License (GPL) that lets build UIs using XML. It includes an ultra-light weight, multi-threaded web server, a portal engine, a template engine (Velocity), a scripting interpreter (Python) and other tools (Bauer et al. 2005). Luxor is different of the Mozilla XUL (Mozilla Developer Center 2010) (which is presented below in this chapter). The main idea of Luxor is to provide building cross-platform user interfaces as easy as building web pages, with no hard-coded or specific operating system. That idea is focused in joining XUL, SVG, HTML, Velocity, Web Start, XSL/T, Python and other technologies.

MariaXML

The authors (Paternò et al. 2009) present in this paper a language for describing user interfaces at different abstraction levels, and also the associated tool that supports such language. The language lays its foundation from previous experiences mainly with TeresaXML (Mori et al. 2003). MariaXML (Paternò et al. 2009) inherits the TeresaXML modular approach with one language for the abstract description and then a number of platform-dependent languages that refine the abstraction, depending on the interaction resources considered. In the first version, the authors have considered the following platforms: graphical form-based, graphical mobile form-based, vocal, digital TV, graphical direct manipulation, multimodal (graphical and vocal) for desktop and mobile, advanced mobile (with support for multi-touch and accelerometers, e.g., iPhone). The language considers abstract and concrete user interface specification. An interface is composed of one data model and one or more presentations associated with a dialog model, which provides information about the events that can be triggered at a given time. The dynamic behaviour of events and its associated handlers are specified using CTT (Paternò et al. 1997) temporal operators. In the case of applications based on Web services, designers and developers often have to compose existing functionalities and corresponding user interface

specifications. An authoring tool, MARIA, has been developed in order to support editing user interface specifications, with the new language, in particular for interactive applications based on Web services. It provides the possibility to switch from a model to the final implementation. For instance, it is possible to specify general rules to associate concrete elements with abstract ones, or specifying how to render concrete elements using a specific implementation technology. When the composition step occurs: it can be either a static composition (occurring at design time) or a dynamic composition (occurring at runtime, namely during application execution). This latter composition is especially significant in ubiquitous applications, since such environments services can dynamically appear and disappear.

MDML

MDML (Multi-Device Markup Language) was born with the objective of proposing a rule-based approach to automate the process of GUI generation, based on a single XML GUI definition (Johnson & Parekh 2003). The GUI must be defined using a MDML schema and MDML is considered to be a swing-based schema. The reason for that is Java Swing be used, which includes a very rich set of widgets and layouts. A framework is available and is structured in four parts:

- Display Engine: processes the MDML input file;
- Rule Engine: parses and validates the language specific rule file;
- Handler: processes the Display Engine and the Rule Engine output, and generates a XML file which adheres to a code generator schema;
- Code Generator: produces the source code for the language.

In order to use this framework, it is required thinking about design first and code later, since the code is automatically generated. Basically, the authors consider extremely important the design in this framework, since the idea is to use this language across multiple platforms and devices, with varying degrees of richness and limitations.

MPML

The main goal of MPML (Multimodal Presentation Markup Language) research was to enable a presentation at anytime and anywhere, using lifelike character agents in place of human presenters (Okazaki et al. 2002). That was the motivation for the creators develop MPML-VR (MPML for Virtual Reality), which provides multimodal presentation in 3D

virtual space. Also a character agent system (MPML-VR Agent) was developed together with the specification, having some features like:

- Control of presentation space: includes background space change, to import an external object and to accept the interactions with the viewers;
- VRML based: it is employed as the 3D virtual space platform, because a lot of contents and authoring tools are available. Also, to port MPML-VR system to other Web3D platform (e.g. X3D) was planned;
- Animation: the creator of a MPML-VR Agent may model gestures just as a standard VRML animation;
- Locomotion: a MPML-VR Agent has also a locomotion engine, which enables a character to move freely around the space;
- Speech and balloon: a MPML-VR agent can communicate with viewers through synthesized speech;
- Emotional expression: the use of emotions is supported through facial expressions and speech emphasizing, previously established for an agent;
- Agent profile: an agent interprets parameters stored on an agent profile, supported on a XML database (e.g. URL of a VRML file supporting appearance, gender, voice pitch, moving speed, available behaviours, etc.).

The MPML language has been developed to distribute multimodal contents through the internet. Thus, a viewing system in cooperation with a web browser has been created. The viewers only have to access to the URL where a MPML-VR content is located. To achieve this, XSLT (XSL Transformations) are used in order to convert an MPML-VR document to HTML, JavaScript to be executed in the browser, and ActiveX/COM for controlling Cortona VRML Client and the Microsoft Speech API.

MXML

MXML was first introduced by Macromedia in 2004 (Adobe 2009). This interface markup language is used in combination with *ActionScript* (Adobe 2013a) to develop rich internet applications. It is used mainly to, declaratively, lay out the interface of applications, and can also be used to implement business logic and internet application behaviours. It can contain blocks of *ActionScript* code, either when creating the body of an event handler function, or with data binding. MXML is often used with Flex Server, which dynamically compiles it into standard binary SWF files. However, the Adobe Flash Builder IDE (formerly

Adobe Flex Builder) and free Flex SDK can also compile MXML into SWF files, without the use of a Flex Server. MXML is considered a proprietary standard due to its tight integration with Adobe technologies. It is like XAML in this respect. No published translators exist for converting an MXML document to another user interface language such as UIML, XUL, XForms, XAML, or SVG. However, third-party plugins for Flex Builder are capable of generating a result other than a SWF file from Flex applications, for instance native mobile applications.

PlasticML

It is high the constant growth of computers and devices in recent years. The availability of such a wide range of devices has become a challenge for designers of interactive software systems. Today, to reach `N` information through `M` peripheral is equal, for the developer, to write `N*M` programs. However, from the user point of view, the service offered should remain the same. Instead of coding `N*M` applications, researchers try to offer one model for many interfaces. This is what the author call *plasticity* (Rouillard 2003): it's the ability of a user interface to be reused on multiple platforms that have different capabilities. PlasticML was created in order to be used to describe internet UIs. With the PlasticML toolkit, programmers can graphically create some XML documents, on a high level interface. The use of XSL translations automatically provides targeted files, such as HTML, WML and VoiceXML (W3C Recommendation 2000).

RIML

RIML (Renderer Independent Markup Language) approach targets the authoring of device independent documents, which are similar to HTML pages (Dermler et al. 2003). RIML stresses the separation of content definition (i.e. what is to be presented) from the description of dynamic adaptations, which can be performed on the content in order to match varying capabilities of devices. Automated web pagination support was a main design goal for RIML (e.g. to distribute for several pages contents which don't fit on a single page). A RIML author requires a minimal knowledge of how a desired layout will be paginated by the RIML adaptation system. Authors familiar with XHTML/XFORMS (W3C Recommendation 2012) can easily create content using RIML, since it is based on such languages.

SeescoaXML

The Seescoa project proposes an architecture for runtime serialization of Java user interfaces into a XML description. This XML description provides an abstraction of the user interface, which is described as a hierarchy of abstract interaction objects (AIOs) (Bodart & Vanderdonckt 1996). Once a user interface has been serialized, and a XML description produced, the description has to move to another device, where it can be "de-serialised" into a user interface for the target device. This de-serialisation involves mapping the platform independent AIO, into platform specific concrete interaction object (CIO). It consists on a suite of models and a mechanism, which automatically produce different final user interfaces, at runtime, for different computing platforms, possibly equipped with different input/output devices, offering various interaction modalities (e.g. joystick). This system is context-sensitive as it is expressed, first in a modality-independent way, and then connected to a specialization for each specific platform (Souchon & Vanderdonckt 2003; Guerrero-Garcia et al. 2009). An abstract user interface is maintained containing specifications to different rendering mechanisms (presentation aspects) and its related behaviour (dialog aspects). These XML specifications are then transformed into platform-specific specifications, using XSLT transformations. Then, they are connected to a high-level input/output devices description.

SISL

Since a long time ago that interactive services such as information and e-commerce services are becoming increasingly more flexible in the types of user interfaces they support. These interfaces incorporate automatic speech recognition and natural language understanding. SISL (Several Interfaces Single Logic) is an architecture and domain-specific language for designing and implementing interactive services with multiple user interfaces (Ball et al. 2000). A key principle underlying SISL is that all user interfaces of a service share the same service logic. SISL provides a clean separation between the service logic and the software for a variety of interfaces, including Java applets, HTML pages, speech-based natural language dialogue, and telephone-based voice access. It is important to denote that SISL is focused on interactive services provided between different interfaces, and was not created to support interfaces representation.

SSIML

3D components can speed up the development of interactive 3D content and enhance the integration of 3D content into applications but, there is a lack of tools which support 3D component concepts above the implementation level (especially for components with complex inner structures and a large number of properties). To address this problem SSIML (Scene Structure and Integration Modelling Language) was introduced. It is a visual language to support the abstract specification of 3D components (Vitzthum 2006) and focuses on the (semi-)formal pre-implementation specification of 3D scenes and their integration into applications. SSIML is an extension of the Unified Modeling Language (UML) and it allows specifying integration relations between UML classes and 3D scene elements, and also the integration of behaviours into SSIML components. Connections between geometries and behaviours can be defined in a more structured manner and furthermore, 3D component architectures can help to facilitate the integration of interactive 3D content into applications. The integration of interactive 3D content can increase the attractiveness and enhance the usability of stand-alone and web applications. It is possible to generate code from it SSIML/Components into X3D (Web3D 2011). By generating code from SSIML models, is possible a seamless transition from the design level to the implementation level.

SunML

Simple Unified Natural Markup Language (SunML) (Picard et al. 2003; Guerrero-Garcia et al. 2009) is a XML-based language to specify concrete user interfaces that can be mapped to different devices (PC, PDA, voice). The innovation of this language is the ability to dynamic components specification. In SunML is also possible to encapsulate the style and the content of each widget, independently of the others. Another feature offered in SunML is widget composition. Some operators have been defined for that purpose: union (semantically-common widgets), intersection, subtraction, substitution, inclusion. Widgets Merging Language (WML) is the extension used for that purpose. SunML presents a reduced set of elements (widgets). Each widget can be differentiated into complex and simple categories, and composition of widgets is used to specify more complex widgets. SunML language is very general and is also applicable to non-graphical interfaces, as voice interfaces (VoiceXML). It supports Java Swing, HTML and UIML.

TeresaXML

TERESA is a project under which the TeresaXML language was developed. This project is able to support top-down transformations, from task models to abstract user interfaces and then to different platforms (such as mobile phones or desktop systems) (Mori et al. 2003). Meaning, the project has tools to support the design and generation of a concrete user interface to a specific platform. It is divided in 2 parts:

- A XML description of a task model language, CTT (Paternò et al. 1997));
- A language to describe abstract user interfaces (TeresaXML);

This project allows designers to keep a unitary view of the design of a given nomadic application (to be seen on a web-browser or a mobile device). The abstract user interface is mainly established by a number of presentations defining its static structure and a number of transitions, defining how the user interface evolves over the time. Each presentation is constituted by a set of interactors, composed through a number of different operators (e.g. grouping, ordering, relation and hierarchy).

UIML

User Interface Markup Language (UIML) (Abrams & Phanouriou 1999) is a XML-based language which allows to declare a user interface independently of the platform or programming language. Originally, UIML was not based in the model-based interface design concepts. However, a UIML document has parts that easily map to some of the existing models. For example, in the *Simple Skeleton UIML Document* code listed just below (UIML 2009), a presentation model is specified in the `<structure>` and `<style>` sections, and a dialog model is specified in the `<behavior>` section.

Simple Skeleton UIML Document.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 3.0 Draft//EN" "UIML3_0g.dtd">
<uiml>
      <head>...</head>
      <interface>
          <structure>...</structure>
          <style>...</style>
          <behavior>...</behavior>
          <content>...</content>
      </interface>
</uiml>
```

In UIML, a user interface is a hierarchy of XML elements and each one can be for e.g. a text, an image or other available widgets. Also a research prototype was developed, that provides an integrated development environment for UIML: *Transformation-based*

Integrated Development Environment (*TIDE*). With TIDE, the developer writes UIML code and the IDE generates the interface. LiquidApps (LiquidApps 2009) is the commercially available authoring environment that is based in UIML.

UsiXML

USer Interface eXtensible Markup Language (UsiXML) is a XML-Compliant markup language to describe the user interface for multiple contexts of use such as, Character User Interfaces, Graphical User Interfaces, Auditory User Interfaces and Multimodal User Interfaces (Vanderdonckt et al. 2004). UsiXML is structured according to different levels of abstraction established by the *Cameleon Reference Framework* (Calvary et al. 2005). The main idea is that anyone who uses UsiXML can design multicontext, multimodal and multiplatform interfaces. UsiXML, basically contains task models, represented using an extended version of CTT (Paternò et al. 1997), domain model analogous to class description in UML, and an abstract and concrete user interface description that includes presentation and dialog models. The authors consider that a single UIDL does not fit all requirements to successfully run a Model Driven Engineering (MDE) compliant approach. Therefore, several tools to support this language were developed, such as: in case of the *abstract interface* level the IdealXML was developed; and the *concrete interface* level is supported by several other tools like SketchiXML, GrafiXML, ComposiXML, PlastiXML and VisiXML.

VHML

VHML (Virtual Human Markup Language) is a language used to accommodate the various aspects of Human-Computer Interaction with regards to Facial Animation, Body Animation, Dialogue Manager Interaction, Text to Emotional Speech production, Emotional Representation, and Hyper and Multimedia information. It can be used to control the overall action of a virtual character (Beard & Reid 2002) and the purpose of it is to facilitate the realistic and natural interaction of a Talking Head/Virtual Human (human-like) with a user, via a web page or a standalone application. VHML is XML/XSL based, and has seven major sub languages important to the functionality of Virtual Humans:

- *DMML*: Dialogue Manager Markup Language;
- *FAML*: Facial Animation Markup Language;
- *BAML*: Body Animation Markup Language;
- *SML*: Speech Markup Language;
- *EML*: Emotion Markup Language;

- *GML*: Gesture Markup Language;
- *XHTML*: HyperText Markup Language.

A testing platform for VHML is available (MetaFace) and this framework is a combination of many technologies designed to bring anthropomorphic (human-like) interaction to websites. Basically, VHML is a language focused on to control the onscreen presence of virtual humans.

VoiceXML

VoiceXML was originally developed by four companies: AT&T, IBM, Lucent and Motorola. It is a W3C standard XML format for specifying interactive voice dialogues between a human and a computer (W3C Recommendation 2000). It allows to develop voice applications and to deploy it in an analogous way to HTML for visual applications. Just as HTML documents are interpreted by a visual web browser, VoiceXML documents are interpreted by a voice browser. VoiceXML has tags that instruct the voice browser to provide speech synthesis, automatic speech recognition, dialog management, and audio playback. Typically, HTTP is used as the transport protocol for fetching VoiceXML pages. Some applications may use static VoiceXML pages, while others rely on dynamic VoiceXML page generation using an application server like Tomcat, Weblogic, IIS, or WebSphere. The language describes the human-machine interaction provided by voice response systems, which includes:

- Output of synthesized speech (text-to-speech) and audio files;
- Recognition and recording of spoken input;
- Control of dialog flow.
- Telephony features such as call transfer and disconnect.

VoiceXML's main goal was to bring the full power of Web development and content delivery to voice response applications, and to free the authors of such applications from low level programming and resource management. It enables integration of voice services with data services using a client-server paradigm.

XAML

XAML is a markup language for declarative application programming for the *Windows Presentation Foundation* (Microsoft Corp 2011). The main idea is to describe rich user interfaces, similar to the interfaces created with *Adobe Flash Platform* (Adobe 2013b). It is possible to create visible user interface elements, using the declarative XAML language, and

then to separate the user interface definition from the runtime logic, by using code-behind files, joined to the markup through partial class definitions. This ability to mix code with XAML is considered to be important because XML by itself is declarative, and does not really suggest a model for flow control. An XML based declarative language is very intuitive for creating interfaces ranging from prototype to production, especially for people with a background in web design and technologies. The following XAML example demonstrates how simple is to create a button as part of a user interface.

```
<StackPanel>
  <Button Content="Click Me" />
</StackPanel>
```

The created button has default visual presentation through theme styles, and default behaviour through its class design.

XDL

XDL (XML Interface Description Language) is used to describe interfaces. It allows developers to create high-level descriptions of interface elements, leaving the exact rendering of the interfaces to the system at run-time (Hutchings & Pierce 2005). XDL supports the description of interface components, behaviours, and division points (across multiple devices), and provides abstractions to support user choices for interface configuration at run-time. Using XDL, a developer creates a single interface description that can be rendered and divided among multiple types of devices. Individual devices interpret the description and render an interface using a UI library (e.g. Swing or MIDP for J2ME), available for their platforms. The lowest level of description in XDL is the component or widget. XDL allows developers to specify a component's attributes and layout within an interface. For e.g. the button element specifies the text and layout attributes for the button within a panel. XDL provides support for most common interface components, including text fields, tables, menus, images and button groups. XDL was designed to be extensible, so that developers can extend the language to support specialized or custom components. Additionally to describe interface components, developers also describe the behaviours for those components in XDL. Behaviours allow developers to specify that an interface reacts to user actions by setting attributes of interface components, displaying new interfaces (windows), or returning information about the interface state to the application. XDL is supported by a framework called DIAMOND which supports a design space for divisible interfaces (for multiple devices).

XForms

XForms (known as the neXt generation of web FORMS) is a XML application that represents the next generation of forms for the Web (W3C Recommendation 2012). It separates the presentation from the data, keeping the principle of separation of concepts, allowing component reuse and device independence. The XForms is intended to be integrated into other markup languages, such as XHTML (W3C Recommendation 2002) or SVG (W3C Recommendation 2011b). The specification provides more rigorous form controls definitions and classifications, applied throughout the specification. The behavioural description (common to all form controls) indicates default layout styling and rendering requirements for the data. The output form control allows rendering non-text media types, particularly images, obtained from instance data. One important feature this specification contains is the ability to create wizard-like interfaces, with form controls dynamically available. In contrast with the original HTML forms, XForms uses a *model–view–controller* approach. The *model* consists in one or more XForms models describing form data, constraints upon that data and submissions. The *view* describes what controls appear in the form, how they are grouped together, and what data they are bound to. And, CSS can be used to describe the form's appearance.

XICL

XICL (eXtensible User Interface Components Language) is a markup language to UI and UI component development for browser-based software (Sousa & Leite 2004). Its syntax is based in XML, HTML and ECMAScript and also follows the Document Object Model (DOM). The intention was to provide a familiar syntax to Web system developer, to achieve the development of UI components to browser-based software, and the development of UI using HTML elements and XICL components. The UI development process using XICL requires a basic environment composed of an editor (XICL Studio), a library of components (XICL Lib) and an interpreter. The XICL Studio environment is browser-based software which integrates a basic editor, the XICL Lib and the interpreter in the same application. The developer executes it in a browser and it provides commands to store a component in the XICL Lib, to call the interpreter and to view the final DHTML code. Using the editor is possible to edit the specification of user interfaces or components. When editing components, the developer should store its source XICL code in the XICL lib, to be reused in future user interfaces. New UI components are described in XICL using HTML elements

and XICL components, by reuse and extension mechanisms. To develop a complete UI, it is necessary to write the specification in XICL code, using the editor and then call the interpreter to generate the DHTML final code. The code may then be executed in any common browser.

XIML

eXtensible Interface Markup Language (*XIML*) (Puerta & Einstein 2001) supports functionality to the complete user interface life cycle that allows, virtually, any element to be modelled and correspondingly any attribute or relationship to be associated with it. XIML representation is a collection of different models called components. In its first version, XIML predefines 5 basic interface components: task, domain, user, dialog, and presentation. The first three of these can be characterized as contextual and abstract while the last two can be described as implementational and concrete. XIML allows design models to be transformed in multi-device implementation solution. However, although this language has an interested purpose, it does not offer reusability and extensibility of components. In order to validate the expressiveness and usefulness of XIML (Puerta & Einstein 2001) undertook tests and projects including multi-platform development and intelligent interaction management.

XISL

The purpose of XISL (eXtensible Interaction Scenario/Sheet Language) is to provide a common language for web-based multimodal interaction systems on various types of terminals (e.g. touchscreen displays and mobile phones) (Katsurada et al. 2003). It enables to describe synchronization of multimodal inputs/outputs, dialog flow/transition, and some other descriptions required for multimodal interaction. XISL inherits a lot of tags from VoiceXML (W3C Recommendation 2000) and SMIL (Synchronized Multimedia Integration Language, whose W3C MM group has meanwhile been ended). These tags enable XISL to control dialog flow/transition, synchronization of multimodal inputs/outputs, and other descriptions required for multimodal interaction.

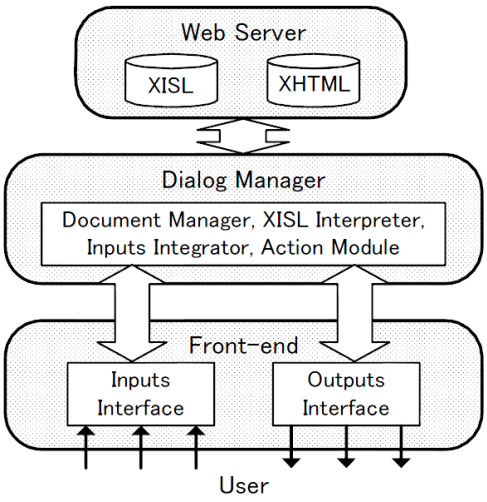

Figure 12: XISL execution system.

In Figure 12, it is shown the outline of a XISL execution system composed of two modules: a frontend module and a dialog manager module. The dialog manager interprets XISL documents, manages dialog flow, and controls inputs and outputs. It is independent of both application and terminal. On the other hand, the front-end is a user interface terminal that handles various modalities. It depends on applications and terminals, respectively. This division enables to reuse the dialog manager when introducing new terminals with different types of modalities.

XMMVR

The XMMVR (Extensible Markup Language for Multimodal Interaction with Virtual Reality worlds) is a markup language for defining scene, behaviour and interaction, which considers each world or interactive movie as a "xmmvr" element (Rodríguez et al. 2007; Rodríguez et al. 2010). It can be classified as a hybrid markup language because the idea is to use other languages such as *VoiceXML* and *X+V* for voice interaction, and *X3D* or *VRML* for scene description that would be embedded in it. Thus, processing the XML files (valid by a DTD) will allow XMMVR to link to programs and files required to operate the specified world. It is a system led by events and a timeline will not exist. Each element has a graphical appearance, specified by a VRML file, and a behaviour that allows interaction with the user. An event can be triggered by user due to a graphical interaction "GUI" or a vocal interaction "VUI". Hence, it is possible to develop multimodal applications through the specification of the scenes, the behaviour and the interaction in virtual worlds.

XUL

XUL stands for XML UI Language and was pioneered by Mozilla (Mozilla Developer Center 2010). XUL is superior to API-based UI toolkits such as Swing or WinForms, because it clearly separates the user interface into four parts:

- Content (structure and description of UI elements);
- Appearance (look & feel, skin, themes);
- Behaviour (scripting) and;
- Locale (localization information for internationalization).

XUL provides the user interface bases for a cross platform application. As DHTML was created for web page interface, XUL has been created also for interface of applications. It is used to build feature-rich platform applications that can run connected or disconnected from the internet. These applications are easily customized with alternative text, graphics and layout, so they can be readily branded or localized for various markets. The architecture is based on the use of packages that can approach the interface to an abstract or concrete perspective. The packages are composed of content, appearance, behaviour, location and platform. Each one uses different technologies. It is similar to HTML, and web developers, who are already familiar with DHTML, will not have difficulties to learn XUL. It is possible to design the interface using the markup language, to establish the appearance with CSS *style-sheets* and to use *JavaScript* to manipulate behaviour. Unlike HTML, XUL has an extensive set of graphical components used to create menus, toolbars, text boxes, among other components (Carnero 2007). The *presentation model* abstracts some platform-specific elements, but is typically fixed to one interface modality, with constraints on the form factor and input techniques (Bojanic 2007). It has the ability to separate the interface from the application logic, which simplify the interface maintenance, without changing application logic. A restriction of this XML representation is the excessive *JavaScript* dependence, which somehow limits the abstraction ability.

Comparative Analysis

Each XML-UIDL has been created with an objective and some evolved to be used with a specific purpose. XForms is an example of how the research has been incorporated into an industrial standard. It is a XML language for expressing the next generation of *web forms*, by splitting traditional forms into three parts (XForms model, instance data, and user interface). Other two languages with some relation between them are, for example, Luxor

and XUL. Although Luxor is based on Mozilla XUL, it doesn't strive to be a 100% Mozilla XUL clone (Bauer et al. 2005). Instead, Luxor tries to add more value to XUL through its tools. Also a comparison can be made between XUL and XAML. Both eases to create, to edit and to reuse graphical user interfaces for desktop and Web applications. However, XAML it is for Windows what XUL it is to Mozilla Firefox. Also, XUL is multi-platform and run natively, while XAML requires .NET or a compatible environment. HTML tags may be integrated directly into a XUL source, while XAML uses its own tags, equivalent to those of HTML, which allows concluding that coding XAML is simpler, but it does not run natively. Finally, another difference between them is that XUL is interpreted and supports bitmap images, while XAML is compiled and vector based. Also, it was possible to verify that some languages support other languages in its specification in order to improve/extend features that intend to provide access to the interface designer/developer. Typically, these languages have been created to support multimodal interfaces, such as XMMVR, which uses other languages such as VoiceXML and X+V for voice interaction and X3D or VRML.

The Table 5 indicates the XML-UIDL languages analysed concerning the *CFFI* criteria. The criteria which is not checked on the table means don't be supported by the language or the authors don't indicate/demonstrate it in the literature. Some languages were not created to support components having visual appearance (e.g. CCXML, VoiceXML). Other languages support the implementation of complete visual user interfaces for real applications. However, most of them are dependent of platforms and/or toolkits of predefined widgets (e.g. GIML, XAML). Also, only 3 of the analysed languages support free design of components on its specifications (e.g. InkML, MXML and XAML). It was possible to verify also, none of the languages completely supports the *CFFI* criteria. The two criteria which have a larger number of languages that do not support them are:

- Criteria 3: to allow free design of components;
- Criteria 4: to be independent of any software platform.

		Functional				
	Complete	Visual presentation	Components composition	Interface behaviour	***Free Design***	***Platform Independent***
3DML		✓				
AAIML		✓	✓	✓		✓
AUIL		✓		✓		
AUIML	✓	✓		✓		✓
CCXML				✓		
D3ML		✓		✓		
DISL		✓	✓	✓		
EMMA				✓		
GIML	✓	✓	✓	✓		
GladeXML	✓	✓	✓	✓		
IMML	✓	✓		✓		
InkML		✓			✓	
InTml		✓		✓		
ISML	✓	✓				
Luxor	✓	✓				
MariaXML	✓	✓	✓	✓		
MDML	✓	✓				
MPML		✓				
MXML	✓	✓	✓	✓	✓	
PlasticML	✓	✓				
RIML	✓	✓		✓		
SeescoaXML	✓	✓		✓		
SISL						✓
SSIML		✓		✓		
SunML		✓	✓	✓		
TeresaXML	✓	✓	✓	✓		
UIML	✓	✓	✓	✓		
UsiXML	✓	✓		✓		
VHML				✓		
VoiceXML				✓		
XAML	✓	✓	✓	✓	✓	
XDL	✓	✓	✓	✓		
XForms		✓	✓	✓		
XICL	✓	✓	✓	✓		
XIML	✓	✓		✓		
XISL		✓		✓		
XMMVR		✓		✓		
XUL	✓	✓				

Table 5: Analysis of XML-UIDLs, according with the *CFFI* criteria.

Hence, it was decided to increase the abstraction level, in order to continue to search for other methods which may contribute to support the *CFFI* criteria. Thus, a bibliographic research was made in order to identify *abstract interaction objects* (*AIOs*) which are

specification methods more focused in user interface components, and three of them are introduced in the following section.

2.4 Abstract Interaction Objects

Previous work in the field of interactive graphics have been done, by describing an interface in terms of a collection of *interaction objects* (Duke & Harrison 1993; Savidis 2004a). Additionally, (Duke et al. 1994) indicate the notion of interaction object does not need to be confined to graphics systems, since it represents a useful structure for thinking and reasoning about the behaviour of interactive systems in general. Usually, interaction objects are seen as independent entities with a local state that can engage events, using its global interface, possibly resulting in changes of its state. Interaction objects are not necessarily visual but, non visual interaction objects represent a low percentage of the total number of interaction objects. An *abstract interaction object* (*AIO*) represents the data structure of a user interface object without the graphical representation and is independent from the *environment* (Silva 2000). Generally, it is defined as a conceptual representation of an interface object. A *view* is sometimes referred as a collection of *AIOs* logically grouped in order to deal with the inputs and outputs of any task (Silva 2000). A *concrete interaction object* (*CIO*) represents any visible and manipulative user interface visual component, which can be used to input or output information regarding any user's interactive task. Each *human computer interaction* relies on three dimensions: the user, the task and the environment. Thus, during the interface designing process, three steps can be identified: (1) the selection of appropriate *AIOs*; (2) the transformation of *AIOs* into *CIOs* and (3) the placement of *CIOs* to obtain the final observable environment (Bodart et al. 1993). *CIOs* are sometimes called widgets or *physical interactors*, which often include simple objects (e.g. edit box, push button) and are also understood as the visual components of a UI. *AIOs* are the abstractions of these widgets. However, widgets are objects which including some restrictions such as (Bodart & Vanderdonckt 1996):

- *Lack of uniformity and standardization*: *CIOs* induces a generalization problem, once the same object can be found in different physical environments with different names, different graphical presentations, but still with the same behaviour;
- *Absence of abstract representation*: without such a representation, developers are submitted to specificities of several physical environments, designers are forced to

not ignore low level details, and human factors experts are mainly focused on presentation, rather than on behavioural aspects;

- *Lack of compatibility with OO programming*: in this programming paradigm most object classes' libraries encapsulate logically related classes with respect to inheritance relationship. Basic classes usually provide foundation classes, widget classes and graphical object classes. Any abstraction that is not compatible with OO dedicated mechanisms will be limited and useless;
- *Difficulty of reusability*: the reuse of existing objects leads to the acceptation of existing *CIO*'s constraints which can be considered as insufficient under some given circumstances. Creating a new *AIO* from existing *AIOs* will be less hazardous, as abstract properties (e.g., attributes) can be reused from one *AIO* to another.

Putting the focus on the user interface abstract representation, (Rodeiro 2001) refers that interface visual appearance and its behaviour can be abstractly represented, with independence on the development system, avoiding premature commitment to a concrete visual presentation. The representation of the interface visual appearance (before implementation) should consider the following details:

- *Components Visual Appearance*: abstract representation (geometry and attributes) of individual interface components;
- *Visual Composition*: abstract representation of visual components composition, in order to obtain a final visual user interface;
- *Dialog*: abstract representation of dialog established between interface components. Each component, which receives the dialog (action) of the user, is here represented together with the changes that occur in the other interface components.

Taking the above aspects in consideration, and knowing that the use of *AIOs* avoids premature commitment to specific presentations (leaving open the prospect of alternative visual presentations for different environments) it was decided to identify in literature some *AIOs* which may comply with the *CFFI* criteria. Thus, three *AIOs* have been analysed (*Interactor*, *Abstract Data View* and *Virtual Interaction Object*), focused in its features concerning: visual presentation, components composition and behaviour.

2.4.1 Interactor

Faconti & Paternò (1990) introduced the concept of *interactor* to represent an entity in interactive graphics capable of both input and output. The utility of *interactors* extends beyond the domain of graphics systems to the more general problem of developing interactive systems in a principled way. Here, input and output may involve the use of novel techniques such as speech recognition, motion sensors, sound, and even tactile stimulation. Interaction devices may also be soft, for example windows and icons, or novel structures such as stereoscopic images in a "virtual reality" setting. The challenge, facing system designers, is to select from this widening array of interaction techniques while taking into account human factors such as error recovery and persistence. The objective of the proposed *interactor* was to develop "building blocks" that may facilitate the rigorous development of interactive graphics systems. Thus, *interactors* are components (objects) used to describe an interactive system and do not indicate a specific language, but an adequate structure to model an interactive system. With *interactors* is possible to structure interactive systems models, by using the concept of an object being capable of displaying part of its state. Thus, each *interactor* has a state (defined by a set of attributes), a number of events (defined by a set of actions) and a rendering relation to specify which attributes/actions are perceivable and can be manipulated by the users. Two models have adopted this concept on their specifications: CNUCE and York (Duke et al. 1994; Harrison & Duke 1994). The formalisms used to express these models are based on different approaches. In the case of the CNUCE model, and at the highest abstraction level, an *interactor* is externally viewed as a "*black box*" that mediates between the "*user side*" and the "*application side*" (Duke et al. 1994). It can receive information from every side, process it and return it (Figure 13).

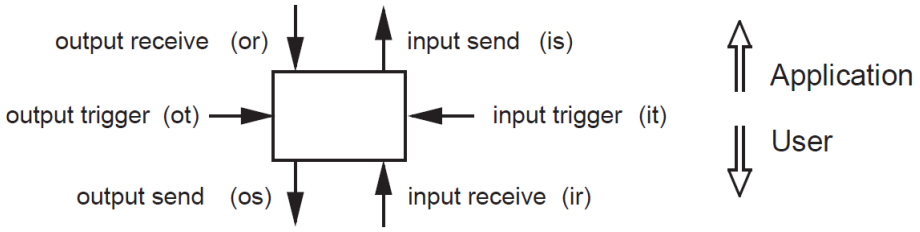

Figure 13: The CNUCE interaction object: external view (Harrison & Duke 1994).

In the CNUCE model an *interactor* can:

– receive (and accumulate) outputs from the application side (*or*);

- receive an output trigger (*ot*) (*interactor* then sends an output to the user side (*os*));
- receive (and accumulate) input from the user side (*ir*) and provide feedback to the user (*os*);
- receive an input trigger (*it*) which results in the *interactor* sending the accumulated input to the application side (*is*).

The *interactor* model developed at York (Figure 14) is based on *states*, *commands*, *events* and *renderings*. Thus, at the abstract level, an *interactor* encapsulates a state (defined by a set of attributes) which is then reflected through a *rendering relation* (*rho*) onto some perceivable *representation* (*P*) (Duke et al. 1994). Typically, an *interactor* has a specific functionality and a unique appearance of its state, due to the fact that these models do not consider multiple rendering functions to represent the *interactor* state. This is the result of the specification model being based only on the *interactor* dialog, rather than on its visual appearance. The interface between an *interactor* and its *environment* consists on an event set of two types: *responses* (events generated by the *interactor*) and *stimuli* (external events reaching the *interactor*).

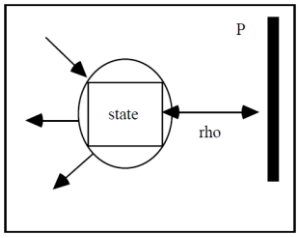

Figure 14: The York interactor (Duke et al. 1994).

It is important to note that an *interactor* is an algebraic conceptualization and as such, its meaning does not contemplate the existence of a visual presentation. Thus, don't exist a function of a visual renderer, but an algebraic renderer. As indicated by (Markopoulos 1997), the York *interactor* model is like a host for the abstract model that preceded it, but it does not cater well for modelling direct manipulation. Therefore, an *interactor* is more viewed as a useful concept that can be used in formal analysis of any interactive system.

2.4.2 Abstract Data View

Interactive systems are designed and implemented with the goal of providing a clear separation between the user interface and the application (Alencar et al. 1995). The authors indicate the importance of the dialog independence concept and the many advantages provided by it:

- Easy maintenance of an existing system;
- Support for alternative visualizations of a single system;
- Design and application reuse.

A user interface design concept, which includes the above mentioned characteristics, is the *Abstract Data View (ADV)*. *ADVs* are *Abstract Data Objects (ADOs)* or objects that have been specifically augmented to support interfaces specifications, with event-driven input and output operations, but also a mapping that ensures an *ADV* interface conformed with the state which is associated to a *ADO* (Alencar et al. 2002). *ADVs* are formal models of interface objects, which are used as an interface between the computer system and the user. They have been viewed as a design paradigm for user interfaces (Cowan & Lucena 1995) since different *ADVs* could visualize the same *ADO* and consequently, any interface can support alternate "*views*" of data (or *interaction* modes) (Alencar et al. 2002). In order to maintain the separation of concepts and to promote the reuse of components, *ADOs* are specified in order to not have any knowledge of their associated *ADVs*. The connection between the *ADO* and the *ADV* captures the *WYSIWYG* nature of the user interface (Cowan & Lucena 1995). At first glance, this design resembles the *MVC* model (Krasner & Pope 1988). However, in this case, the *controller* handles the input, whereas in the *ADV* model the input and the output are handled by each *ADV* procedure. Both *ADVs* and *ADOs* can be used to change or to query their state, and those actions can be divided into two categories: *causal actions* and *effectual actions*. *Causal actions* are the input events acting on *ADV* when acting as a user interface or an interface to some other media. These actions are triggered by external actions to the system and hence, internal objects cannot generate this type of actions. A "*keystroke*" or a "*mouse click*" are simple examples of input events that are *causal actions*. *Effectual actions* are the actions generated directly (or indirectly by a *causal action*) and are supported by both *ADVs* and *ADOs*. The triggering of an *effectual action* by another action will normally be a synchronous process. This type of actions can be viewed as activation or a procedure within the public interface of an *ADO* or an *ADV*. Additionally, a visual formalism denominated *ADVchart* notation has been created (Carneiro et al. 1993) in order to be possible to have both visual and textual representations of the *ADV* design.

2.4.3 Virtual Interaction Object

According with (Savidis 2004a) and within the context of user interface development, *interaction objects* play a key role in implementing the constructional and behavioural

aspects of *interaction*. *Virtual interaction objects (VIOs)* are synonymous of *AIOs*, in the sense that it can be totally separated from physical *interaction* processes. The support is provided by a interface programming language called *Interface Generation Tool Language* (*I-GET*) which falls into the domain of *user interface development systems* (Myers 1995) (usually consisting on several tools for fast user interface development). This language and its accompanying tools have been specifically designed to support several developmental features, namely the ability to support *AIOs*, with definitions of polymorphic mapping of toolkit objects, while supporting plural instantiations. In this sense, it is possible to programmatically define *VIO* classes within a subset of *I-GET*, to specify their mapping-logic and to physically bind virtual object classes across different target platforms (interface toolkits).

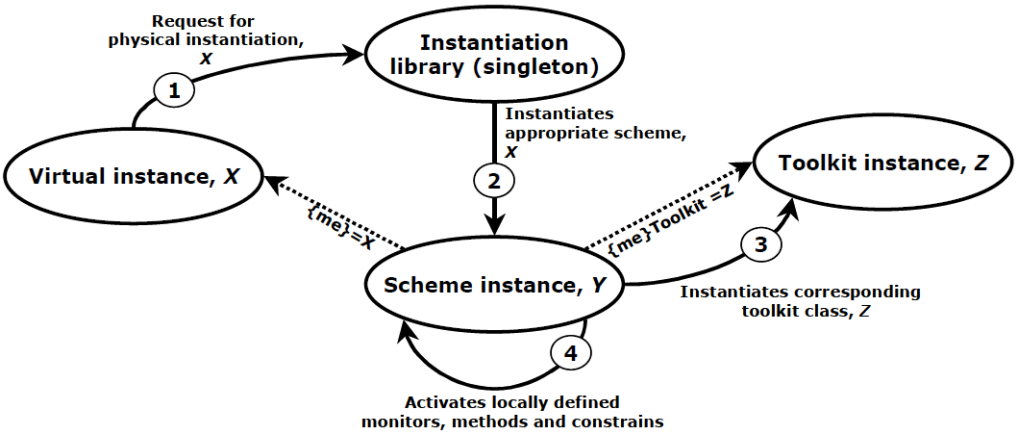

Figure 15: Automatic runtime steps for each target to physically instantiate a virtual class (Savidis 2004a).

The *I-GET* language supports some virtual classes and offers mechanisms for creating and manipulating new virtual object classes (Savidis 2004b). It is also possible to maintain a consistent mapping state between the virtual and the physical environment (by means of the toolkit object class). The Figure 15 represents the virtual instance of creating a process (at runtime) and requires the realisation of its physical instantiation from every linked instantiation library. As a result, each instantiation class will create an instance of the appropriate mapping scheme. Then, the newly created scheme instance will automatically produce an instance of its associated toolkit class. Finally, the scheme will activate any locally defined constraints or an implementation method, which will establish the runtime mapping state between the virtual instance and the newly created toolkit instance.

Comparative Analysis

Therefore, it is possible to identify some similar characteristics between *interactors* and *ADVs*, with the possibility of both receiving and sending events from/to the environment (Table 6).

	receives (in)	***sends (out)***
interactor	stimuli	response
ADV	causal action	effectual action

Table 6: *Interactor* and *ADV* input/output with the environment.

The *interactors* and the *ADVs* were originally developed to enable interface objects description. *Interactors* do not support visual presentation and, being algebraic conceptualizations, it could be more useful in the formal analysis of an interactive system. *ADVs* are structured to support visual presentation, but the dialog must be established through *ADOs*. Both components, *interactors* and *ADVs*, do not consider their specifications in the context of a complete interface, and they mainly focus on defining the simple components with multiple states and the relationship between them. As both were not conceived for allowing a complete interface description, its main goal was to establish the *interaction* between interface objects, to be used during the interface design process. In contrast, the interface specification used by the *VIOs*, mainly focuses on the polymorphic mapping to external objects across different target platforms (available in interface toolkits libraries). Following, the 3 analysed *AIOs* were disposed in Table 7 under the *CFFI* criteria perspective.

		Functional				
	Complete	Visual presentation	Components composition	Interface behaviour	***Free Design***	***Platform Independent***
Interactor				✓	(✓)	✓
ADV		✓		✓	(✓)	✓
VIO		✓		✓		(✓)

Table 7: Analysis of some XML-UIDLs, according with the *CFFI* criteria.

From the analysis to Table 7 is verifiable that none of the previous 3 *AIOs* totally complies with the *CFFI* criteria. With respect to Criteria 3 (Free), both *interactor* and *ADV* have a partial support of it. All 3 *AIOs* support Criteria 4, despite *VIO* has a limited number of platforms which support it. Even though the 3 previous *AIOs* are complex components, none of them support all characteristics established by the *CFFI* criteria, standing out the impossibility to do component composition, in order to create complete functional visual interfaces.

2.5 Conclusions

This chapter started by establishing the criteria (*CFFI*) to identify methods to specify complete visual user interfaces. As the term *specification* usually refers to a declarative model, with well defined and consistent rules to describe the interface appearance and functionality, an interface design process composed by several steps (represented by models) was described. Following, a comprehensive analysis of methods to specify visual user interfaces was presented. From that list, two methods to specify complete visual user interfaces have been identified (IOG and DGAIU), despite not totally complying with the established *CFFI* criteria (both just use *simple components* to represent visual interfaces and even IOG doesn't support hierarchy of components). That level of detail required to specify a visual interface grows the specification considerably, making it impractical to use.

Following, prototyping and development tools were introduced, with a special focus in the analysis of some XML-UIDLs. It was noticed the growing number of these languages, indicating that a paradigm shift is taking place. From the analysis made, it was possible to verify that some languages do not support components having visual appearance (e.g. CCXML, VoiceXML). Other languages that support the implementation of complete visual user interfaces for real applications are dependent of platforms and/or toolkits of predefined widgets (which merely allows the presentation manipulation), and also, only 3 of the analysed languages support free design of components in its specifications. Several of these XML-UIDLs have been created with a specific purpose and researchers face new challenges concerned with the fact that new types of user interfaces (sometimes called *next generation user interfaces*) are emerging. Their targets are multi-platform (e.g. MS-Windows, X-Windows, Mac OS, Windows Phone, Android, iOS) and have multimodal capabilities, as the result of the increasing diversity of devices and interaction styles. Because of that need to represent multimodal interfaces, several XML-Compliant languages have emerged (e.g. DISL, EMMA, MariaXML). However, concerning the *CFFI* criteria, none of the analysed languages full comply with that criteria and, a large number of languages don't allow to freely design components and aren't independent of any software platform. As none of the previous introduced methods supports the *CFFI* criteria, it was decided to increase the abstraction level and to seek for interaction objects with a higher abstraction level (AIOs), and which could be used to specify complete visual user interfaces. However, none of the studied AIOs comply with that objective.

From the several methods to specify visual user interfaces introduced in this chapter (specification models, tools and languages) the one which best meets the *CFFI* criteria is DGAIU (in spite of it just allows specifying complete visual user interfaces using simple components, and doesn't support the use of complex components) However, as the more complete method, it was decided to deepen the knowledge on the structure and functioning of the DGAIU representation system, through the design and implementation of a test bed visual user interface, presented in the following chapter.

Chapter 3

Visual User Interface Case Study

3.1 Introduction

Over the years several methods have been used to specify user interfaces. These include *grammars*, *algebraic specifications*, *task description languages*, *transition diagrams*, *state charts* and *interface representation graphs* (Carr 1994). All previously analysed methods were found to be unsuitable to specify complete functional visual user interfaces, with exception on DGAIU (Rodeiro 2001). The DGAIU representation system has application on GUI interfaces, based on direct manipulation interaction style, supporting interactive visual components which provide dialog (set of possible operations able to execute over the component) and using state diagrams to represent them. In this approach, the transitions represent user inputs and the nodes represent visual global interface states. Computer outputs are represented as annotations to the state or either to the transitions. This representation system is the one which best meets the steps of *composition* and *presentation* regarding the *interface visual appearance* represented by the *Interface Design Steps*. Thus, it has been set up an example, in order to have a test bed user interface, representing a complete functional user interface. It consists of a simple game for younger children, containing visual elements representing sport balls and sport fields. From the visual interface design, built using the IDE available (DGAIUDE), a file has been generated (containing the abstract interface specification). Then, a prototyper tool, able to read the

DGAIU specification language, was used to run a functional visual user interface from its abstract representation.

3.2 A User Interface Example

After the analysis described in previous chapter, concerning user interface specification methods existent in literature, the next goal was focused in determining a visual user interface with significant functionality details, in order to be used in following steps of this research. The established objective was to build an interface for a game, due to the dynamic provided by such this type of visual interface in generating multiple visual states, obtained from user interaction with visual components. Thus, an idea to create a simple game for younger children emerged, consisting of sport balls and sport fields (left side of Figure 16). The generic game rules established were:

- The child must associate the balls with the correct sport fields (using a mouse as the physical interaction device);
- The child starts by selecting a ball and then try to match the correct sport field;
- If one ball is already selected, the child can change the selected one, simply choosing other desired ball;
- If the child does not perform the correct match between the selected ball and the correspondent sport field, that ball is deselected;
- The child cannot select/deselect a ball/sport field that is already correctly matched;
- The game ends when all associations are established.

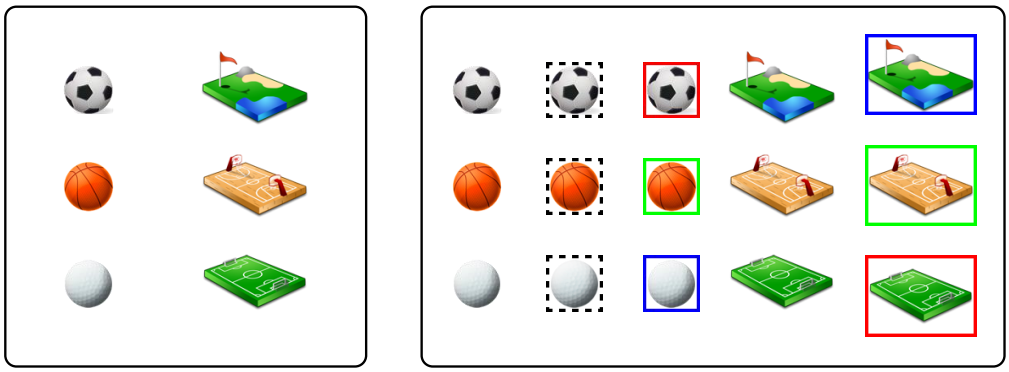

Figure 16: A simple game interface for younger children using 15 *SCs*.

In order to create the game interface, 15 *simple components (SCs)* (discrete visual elements) were used (15 images in right side of Figure 16). For each sport ball, three possible

visual states were considered (*normal*, *selected* and *correct*) while two visual states (*normal* and *correct*) were considered in the case of each sport field.

3.2.1 User Interface Interaction

The interaction with the game interface occurs between the user and the *SCs*. Other type of interaction derives from that previous interaction, and is the one that occurs between components themselves. Both types of interactions over a *SC* are supported on 2 features that a *SC* has, and which are, the ability:

- To receive events (from the user or from other components);
- To execute events on other components.

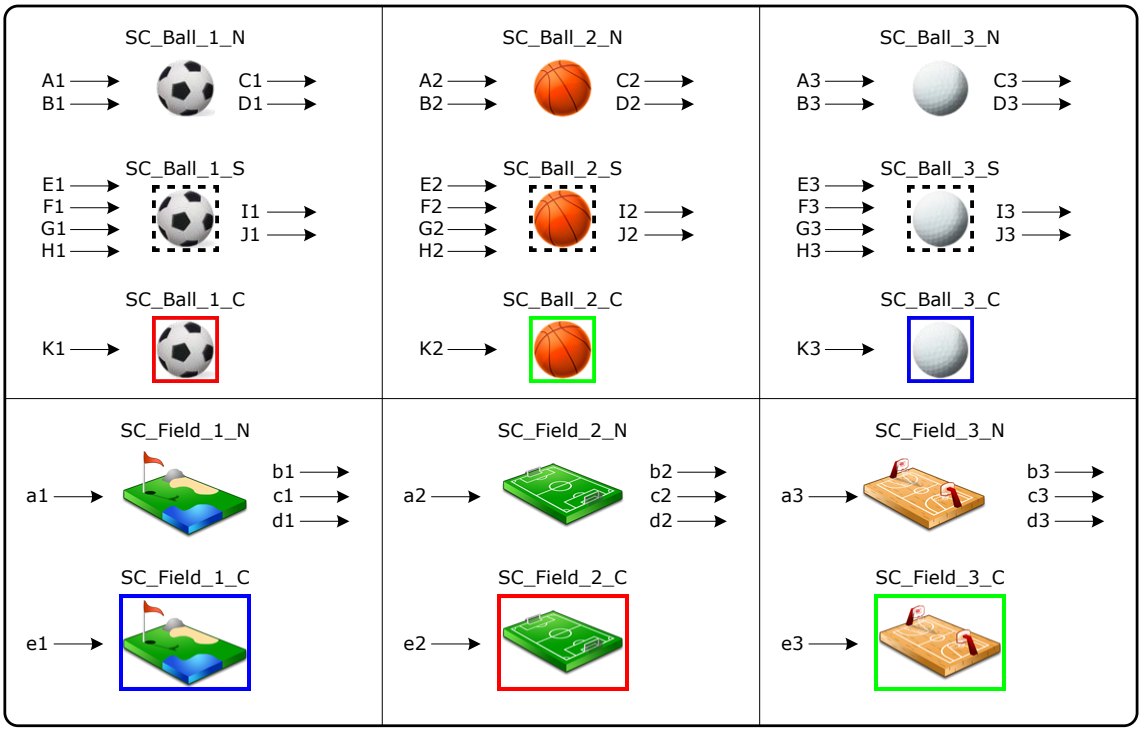

Figure 17: Input/output events over each one of the 15 game visual *SCs*.

After all possible visual states (*simple components*) were defined, it was established which components respond at which events (the balls and the sport fields in this case). In order to simplify the example, it was decided to use *mouseClick* event detection for user interaction. This interaction can be divided in two: interaction with the sport balls and interaction with the sport fields. The visual feedback provided to user is performed by visual states changes, resulting from user interaction. The events that are possible to occur with all 15 *SCs* are indicated in Figure 17. For each component, it is possible to verify the related

events (input and output events). Each sport ball is composed by 3 *SCs* (there are 3 sport balls) and has a similar event set. For example, *SC_Ball_1_S* component (correspondent to the *selected* visual state) supports 6 events: receives 4 (input to the component) and sends 2 (output from the component). Each one of the similar components, related with the other two balls (*SC_Ball_2_S* and *SC_Ball_3_S*), supports also 6 events. Then, considering the sport fields, the situation is identical as it is to the sport balls: each sport field has an event set similar to the three sport fields. For example, the *SC_Field_1_C* component (correspondent to the *correct* visual state) has only one event (input into the component) and doesn't have any output events. The user of this visual game will interact with the *SCs* to complete the objective of the game: to connect the sport balls with the correspondent sport fields. For the user is totally transparent with which *SC* is interacting at a given time (e.g. *SC_Ball_2_S* or *SC_Field_3_N*). Basically, he's selecting sport balls and sport fields.

	User event	Event type	Visual Component
1	A1	*mouseClick*	SC_Ball_1_N
2	G1	*mouseClick*	SC_Ball_1_S
3	A2	*mouseClick*	SC_Ball_2_N
4	G2	*mouseClick*	SC_Ball_2_S
5	A3	*mouseClick*	SC_Ball_3_N
6	G3	*mouseClick*	SC_Ball_3_S
7	a1	*mouseClick*	SC_Field_1_N
8	a2	*mouseClick*	SC_Field_2_N
9	a3	*mouseClick*	SC_Field_3_N

Table 8: Identification of 9 user events to trigger over 9 correspondent *SCs*.

Still considering this game interface example, is observable that the user may trigger 9 *user events* (*mouseClick*) over some of the 15 *simple* visual components. It is possible to verify in Table 8 those user events, which are responsible for interface visual changes. Each *user event* may result in one or more own actions (to change parameters of *SC* itself) and may also to trigger events on other interface components, changing its visual appearance. The behaviour (changes occurring in the interface visual appearance) resulting from user interaction with the game interface visual components is represented in Figure 18. When the user clicks over a sport ball (triggers over a *normal* or a *selected* visual component (e.g. *SC_Ball_1_N* or *SC_Ball_1_S*)) several events over other components are triggered (Table 9). When clicking on a sport field in *normal* state (e.g. *SC_Field_1_N*) other events are also triggered (Table 10). Following, in Table 9 and Table 10, the details of the interface interaction logic are explained.

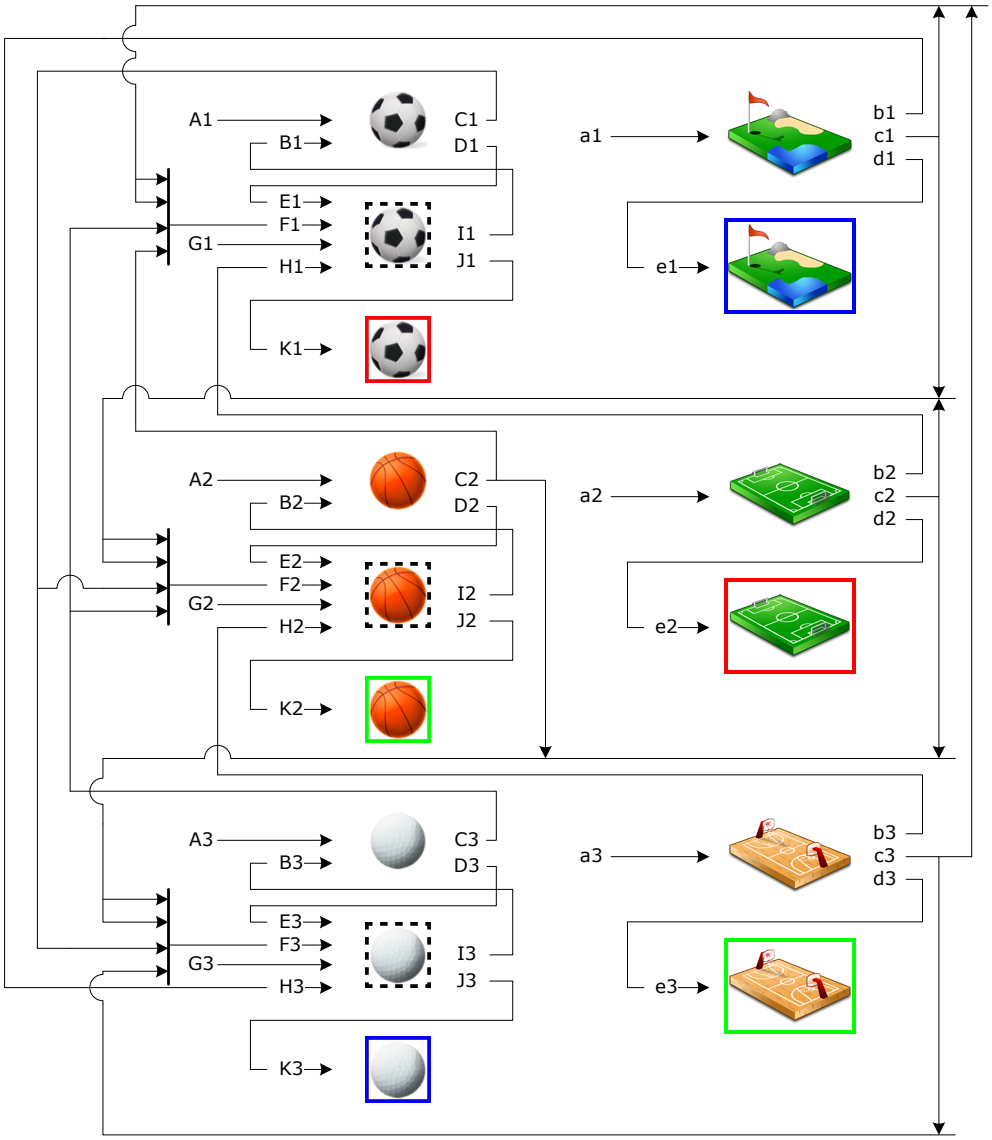

Figure 18: Structure of events resulting from user interaction.

Exemplifying for *SC_Ball_1*, three *SCs* are used: *SC_Ball_1_N*, *SC_Ball_1_S* and *SC_Ball_1_C*. According with the game rules established, the user may interact with only the first two components (*_N* and *_S*). This happens for all sport balls. The third visual component of each sport ball (*_C*) becomes visible only after a sport ball to be selected and the user has clicked over the correspondent sport field. The sequences of events regarding user interaction (*mouseClick*) with two *simple* visual components of a sport ball are depicted in Figure 19 (in this case, for the *SC_Ball_1* the components: *SC_Ball_1_N* and *SC_Ball_1_S*). For the other two balls, the events sequences are identical.

	Receive Events			*Perform Own Actions*	*Executes Events in Other Components*	
SC_Ball_1	*_N*	A1	*mouseClick*	SC_Ball_1_N.visible=**F**	C1	Executes F2 Executes F3
					D1	Executes E1
		B1	Evt from: **SC_Ball_1_S**	SC_Ball_1_N.visible=**T**		
	_S	E1	Evt from: **SC_Ball_1_N**	SC_Ball_1_S.visible=**T**		
		F1	Evt from: **SC_Ball_2_N** or **SC_Ball_3_N** or **SC_Field_1_N** or **SC_Field_3_N**	SC_Ball_1_S.visible=**F**	I1	Executes B1
		G1	mouseClick	SC_Ball_1_S.visible=**F**		
		H1	Evt from: **SC_Field_2_N**	SC_Ball_1_S.visible=**F**	J1	Executes K1
	_C	K1	Evt from: **SC_Ball_1_S**	SC_Ball_1_N.visible=**F** SC_Ball_1_S.visible=**F** SC_Ball_1_C.visible=**T**		
SC_Ball_2	*_N*	A2	*mouseClick*	SC_Ball_2_N.visible=**F**	C2	Executes F1 Executes F3
					D2	Executes E2
		B2	Evt from: **SC_Ball_2_S**	SC_Ball_2_N.visible=**T**		
	_S	E2	Evt from: **SC_Ball_2_N**	SC_Ball_2_S.visible=**T**		
		F2	Evt from: **SC_Ball_1_N** or **SC_Ball_3_N** or **SC_Field_1_N** or **SC_Field_2_N**	SC_Ball_2_S.visible=**F**	I2	Executes B2
		G2	mouseClick	SC_Ball_2_S.visible=**F**		
		H2	Evt from: **SC_Field_3_N**	SC_Ball_2_S.visible=**F**	J2	Executes K2
	_C	K2	Evt from: **SC_Ball_2_S**	SC_Ball_2_N.visible=**F** SC_Ball_2_S.visible=**F** SC_Ball_2_C.visible=**T**		
SC_Ball_3	*_N*	A3	*mouseClick*	SC_Ball_3_N.visible=**F**	C3	Executes F1 Executes F2
					D3	Executes E3
		B3	Evt from: **SC_Ball_3_S**	SC_Ball_3_N.visible=**T**		
	_S	E3	Evt from: **SC_Ball_3_N**	SC_Ball_3_S.visible=**T**		
		F3	Evt from: **SC_Ball_1_N** or **SC_Ball_2_N** or **SC_Field_2_N** or **SC_Field_3_N**	SC_Ball_3_S.visible=**F**	I3	Executes B3
		G3	mouseClick	SC_Ball_3_S.visible=**F**		
		H3	Evt from: **SC_Field_1_N**	SC_Ball_3_S.visible=**F**	J3	Executes K3
	_C	K3	Evt from: **SC_Ball_3_S**	SC_Ball_3_N.visible=**F** SC_Ball_3_S.visible=**F** SC_Ball_3_C.visible=**T**		

Table 9: Details concerning interaction over *SCs* used to represent the three sport balls.

For each sport ball (represented by three *SCs*) 11 events can be triggered (7 are input into the components and 4 are output from the components). Exemplifying with the task of selecting a sport ball, in this example, 3 of its events are not used, due to the game rules previously established (e.g. considering the *Ball_1*: the *H1*, *J1* and *K1* events are not triggered, in result of user interaction with the sport ball, but are triggered in result of user interaction with the correspondent sport field).

<table>
<tr><th></th><th colspan="3">Receive Events</th><th>Perform Own Actions</th><th colspan="2">Executes Events on other Components</th></tr>
<tr><td rowspan="4">SC_Field_1</td><td rowspan="3">_N</td><td rowspan="3">a1</td><td rowspan="3">mouseClick</td><td rowspan="3"></td><td>b1</td><td>*1</td></tr>
<tr><td>c1</td><td>*2</td></tr>
<tr><td>d1</td><td>*3</td></tr>
<tr><td>_C</td><td>e1</td><td>Event from:
SC_Field_1_N</td><td>SC_Field_1_N.visible=F
SC_Field_1_C.visible=T</td><td colspan="2"></td></tr>
<tr><td colspan="7">*1 – if (SC_Ball_3_S.Visible==T) → Executes H3
*2 – if (SC_Ball_3_S.Visible==F) { if (SC_Ball_1_S.Visible==T) → Executes F1
if (SC_Ball_2_S.Visible==T) → Executes F2 }
*3 – if (SC_Ball_3_S.Visible==T) → Executes e1</td></tr>
<tr><td rowspan="4">SC_Field_2</td><td rowspan="3">_N</td><td rowspan="3">a2</td><td rowspan="3">mouseClick</td><td rowspan="3"></td><td>b2</td><td>*4</td></tr>
<tr><td>c2</td><td>*5</td></tr>
<tr><td>d2</td><td>*6</td></tr>
<tr><td>_C</td><td>e2</td><td>Event from:
SC_Field_2_N</td><td>SC_Field_2_N.visible=F
SC_Field_2_C.visible=T</td><td colspan="2"></td></tr>
<tr><td colspan="7">*4 – if (SC_Ball_1_S.Visible==T) → Executes H1
*5 – if (SC_Ball_1_S.Visible==F) { if (SC_Ball_2_S.Visible==T) → Executes F2
if (SC_Ball_3_S.Visible==T) → Executes F3 }
*6 – if (SC_Ball_1_S.Visible==T) → Executes e2</td></tr>
<tr><td rowspan="4">SC_Field_3</td><td rowspan="3">_N</td><td rowspan="3">a3</td><td rowspan="3">mouseClick</td><td rowspan="3"></td><td>b3</td><td>*7</td></tr>
<tr><td>c3</td><td>*8</td></tr>
<tr><td>d3</td><td>*9</td></tr>
<tr><td>_C</td><td>e3</td><td>Event from:
SC_Field_3_N</td><td>SC_Field_3_N.visible=F
SC_Field_3_C.visible=T</td><td colspan="2"></td></tr>
<tr><td colspan="7">*7 – if (SC_Ball_2_S.Visible==T) → Executes H2
*8 – if (SC_Ball_2_S.Visible==F) { if (SC_Ball_1_S.Visible==T) → Executes F1
if (SC_Ball_3_S.Visible==T) → Executes F3 }
*9 – if (SC_Ball_2_S.Visible==T) → Executes e3</td></tr>
</table>

Table 10: Details regarding interaction with *SCs* used to represent the 3 sport fields.

The sequences of events regarding user interaction with one sport field *simple* visual component (e.g. the *SC_Field_1_N*) are depicted in Figure 20. For the other two sport fields, the sequences of events are identical.

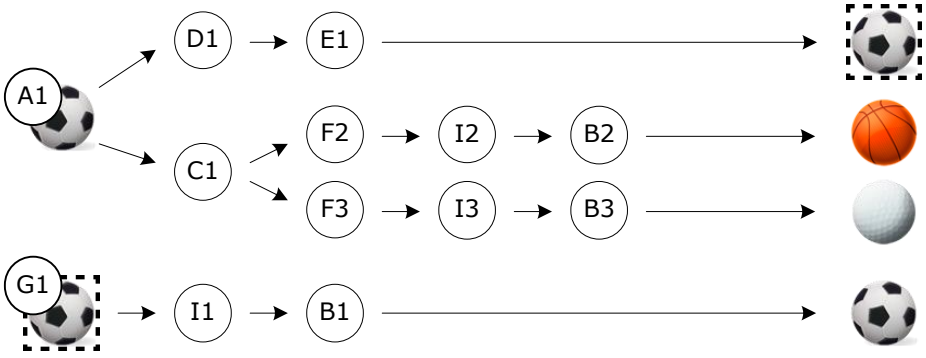

Figure 19: Events sequences regarding user interaction with a sport ball.

The same reasoning is used with the *simple* visual components regarding the sport fields. For example, *SC_Field_1* has two *SCs*: *SC_Field_1_N* and *SC_Field_1_C*. The user may interact with only 3 of those *SCs* (*SC_Field_1_N*, *SC_Field_2_N* and *SC_Field_3_N*). The

second visual component of each sport field (*_C*) becomes visible only when a sport ball is selected and the user clicks the correspondent (*_N*) sport field. For each sport field (represented by two *SCs*), 5 events exist that can be triggered (2 of input and 3 of output). According with the established game rules and considering the task of selecting a sport field: after the user has triggered a *mouseClick* over (e.g. *SC_Field_1_N*), three other events can be triggered (e.g. *b1*, *c1*, *d1*).

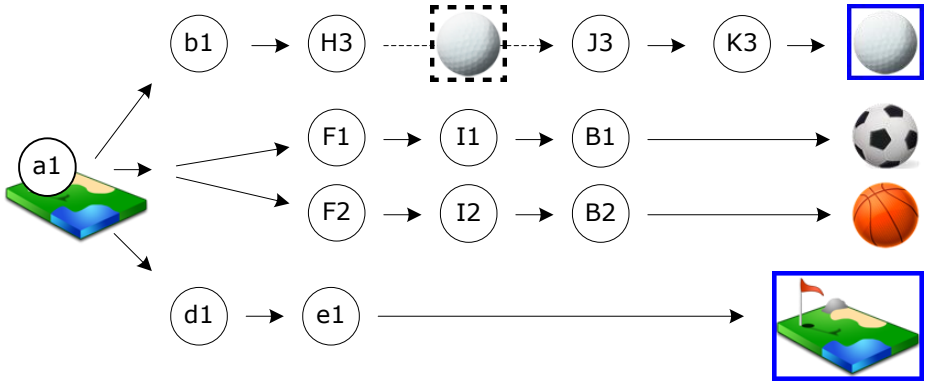

Figure 20: Events sequences regarding user interaction with a sport field.

For each *SC* related with the game sport fields, the received events, the own actions and the events to be triggered on other components are detailed in Table 10. The game control functionality is also described, mainly regarding the test of correct correspondence between a selected ball and a chosen sport field. The events are also indicated (*c1*, *c2* and *c3*), responsible for enabling the interface to return to previous visual state (when a user tries to select a sport field that is not the correct one) according to the selected ball at that precise moment.

3.2.2 Interface Visual States and Transitions

A GUI is mainly a visual structure, composed of a finite number of elements called components (Rodeiro 2001). A component can be perceived by user and to respond at events originated from an internal (application) or external (user) event. A group of visual components waiting for user interaction is called a visual interface state, and a GUI consists of a finite number of states, each one connected through visual transitions and enabled by events. Each visual state is different of other state in its components visual appearance. The complete functionality of the game can be represented through 20 interface visual states (nodes) (Figure 21) and 48 transitions (arcs) (Figure 22) between those states. A state diagram (Figure 22) is a model to represent a UI and some reasons to use it are:

- It allows formal description of UI behaviour;
- The dialog is represented as a set of states with transitions between them;
- It allow representing events to execute on components;
- The course of dialog is linked-up with the current state;
- The transitions between states can be conditional;
- The information concerned with the transitions can be added (change of screen).

Figure 21: The 20 game interface visual states.

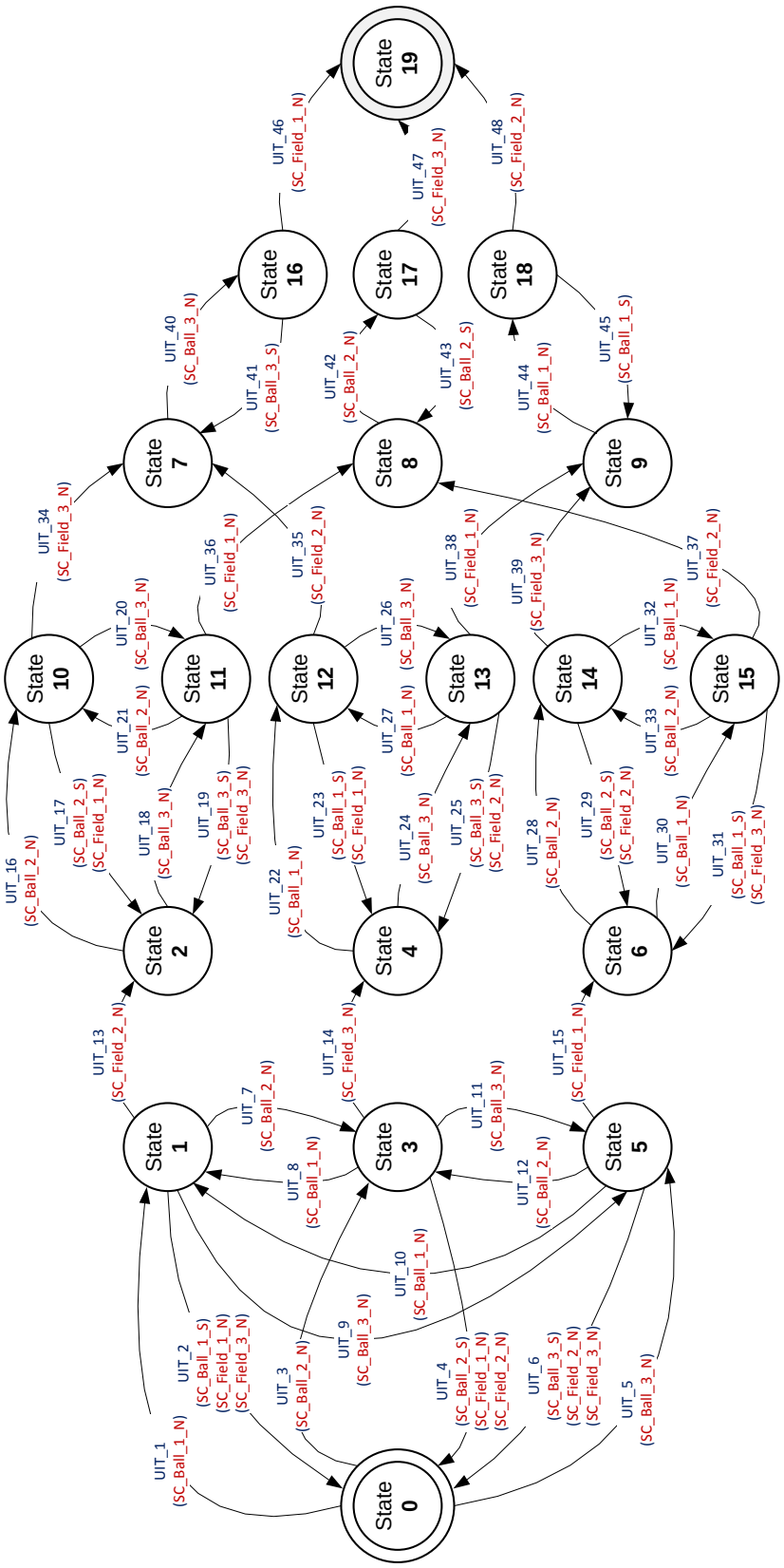

Figure 22: State diagram representing 20 visual states (nodes) and 48 transitions (arcs).

#	T_ID	Event over Component	Initial State	Final State	#	T_ID	Event over Component	Initial State	Final State
1	UIT_1	SC_Ball_1_N	0	1	**25**	UIT_25	SC_Ball_3_S SC_Field_2_N	13	4
2	UIT_2	SC_Ball_1_S SC_Field_1_N SC_Field_3_N	1	0	**26**	UIT_26	SC_Ball_3_N	12	13
3	UIT_3	SC_Ball_2_N	0	3	**27**	UIT_27	SC_Ball_1_N	13	12
4	UIT_4	SC_Ball_2_S SC_Field_1_N SC_Field_2_N	3	0	**28**	UIT_28	SC_Ball_2_N	6	14
5	UIT_5	SC_Ball_3_N	0	5	**29**	UIT_29	SC_Ball_2_S SC_Field_2_N	14	6
6	UIT_6	SC_Ball_3_S SC_Field_2_N SC_Field_3_N	5	0	**30**	UIT_30	SC_Ball_1_N	6	15
7	UIT_7	SC_Ball_2_N	1	3	**31**	UIT_31	SC_Ball_1_S SC_Field_3_N	15	6
8	UIT_8	SC_Ball_1_N	3	1	**32**	UIT_32	SC_Ball_1_N	14	15
9	UIT_9	SC_Ball_3_N	1	5	**33**	UIT_33	SC_Ball_2_N	15	14
10	UIT_10	SC_Ball_1_N	5	1	**34**	UIT_34	SC_Field_3_N	10	7
11	UIT_11	SC_Ball_3_N	3	5	**35**	UIT_35	SC_Field_2_N	12	7
12	UIT_12	SC_Ball_2_N	5	3	**36**	UIT_36	SC_Field_1_N	11	8
13	UIT_13	SC_Field_2_N	1	2	**37**	UIT_37	SC_Field_2_N	15	8
14	UIT_14	SC_Field_3_N	3	4	**38**	UIT_38	SC_Field_1_N	13	9
15	UIT_15	SC_Field_1_N	5	6	**39**	UIT_39	SC_Field_3_N	14	9
16	UIT_16	SC_Ball_2_N	2	10	**40**	UIT_40	SC_Ball_3_N	7	16
17	UIT_17	SC_Ball_2_S SC_Field_1_N	10	2	**41**	UIT_41	SC_Ball_3_S	16	7
18	UIT_18	SC_Ball_3_N	2	11	**42**	UIT_42	SC_Ball_2_N	8	17
19	UIT_19	SC_Ball_3_S SC_Field_3_N	11	2	**43**	UIT_43	SC_Ball_2_S	17	8
20	UIT_20	SC_Ball_3_N	10	11	**44**	UIT_44	SC_Ball_1_N	9	18
21	UIT_21	SC_Ball_2_N	11	10	**45**	UIT_45	SC_Ball_1_S	18	9
22	UIT_22	SC_Ball_1_N	4	12	**46**	UIT_46	SC_Field_1_N	16	19
23	UIT_23	SC_Ball_1_S SC_Field_1_N	12	4	**47**	UIT_47	SC_Field_3_N	17	19
24	UIT_24	SC_Ball_3_N	4	13	**48**	UIT_48	SC_Field_2_N	18	19

Table 11: 48 interface visual transitions identification.

The user has several paths to reach the final state (the solution) without passing for all possible visual states. The initial node (initial state) is the one in which all associated arcs are concerned with output, and don't have any input arc from a node, to which no arc can achieve, without passing for itself. The final node (final state) is the one in which all its associated arcs are concerned with input and none is concerned with output. In Figure 22, the initial state is represented by (*State 0*) and the final state is represented by (*State 19*). The connections (transitions) between nodes (states) always have two associated parameters: the transition identification and the original node identification, which receives an event (and which allows switching between two visual states). It is also possible to verify

the existence of some visual states (*State 2*, *State 4*, *State 6*, *State 7*, *State 8*, *State 9* and *State 19*) which do not allow go back to earlier visual states (e.g. *State 2* does not allow to return to *State 1* and therefore does not allows to access *State 0*, *State 3* and *State 5*). Such situations occur when a sport field is correctly selected or the final game state has been reached.

In order to simplify the state diagram understanding, it was decided to not include the information concerned with event identification, because in this user interface example the user event was considered to be always the same: *mouseClick*. It is verifiable, by analyzing Table 11, which visual transitions result from interaction with components having *normal* or *selected* visual state. Those visual transitions are triggered from 60 possible events performed over the referred components. Interaction with components which are in *correct* visual state does not cause any visual transitions, because that state corresponds to the component final visual state. The components identification, includes all possibilities to produce a visual transition from user interaction with those components (e.g. the *UIT_2* transition concerned with deselecting the football sport ball may be triggered from one of three events: 1) the user clicks in the "selected ball"; 2) and 3) the user clicks in one of the two not correspondent sport field).

After the detailed analysis made on game functionality, focused in all possible visual states and transitions that may occur, in result of user interaction with the game interface, it was decided to design and to implement the prototype of a concrete visual interface. The process is following explained.

3.3 User Interface Structure in DGAIU

From the previous analysis made in Chapter 2, it was verified that DGAIU method (Rodeiro 2001) is the one that is able to specify and to represent a complete functional visual user interface prototype (according with the established *CFFI* criteria). It considers that a visual user interface is not a continuous structure but it is composed of discrete finite elements. From events triggered on visual components, it is possible to calculate the events over the states. Also, according with Harel (1987) (when introduced the *statecharts* concept) much of the literature seems to be in agreement that states and events are a quite natural medium for describing the dynamic behaviour of a complex system. A basic fragment of such a description is a *state transition*, which takes the general form "*when*

event x occurs in state A, if condition C is true at the time, the system transfers to state B". The DGAIU method establishes a model for specifying interfaces using XML (W3C Recommendation 2008). The XML-UIDL languages have the advantage of being transparent to different interface technologies and provide a uniform resource for heterogeneous communication modes. Basically, it is a design system that supports interface representation allowing the interface designer:

- To create new individual components called *SCs* establishing it visual appearance;
- To indicate individual components composition to create the *concrete* interface, with which the user interacts; the components follow a topological hierarchical structure, so that each one can be contained within others;
- To represent *dialog* between components inside the user interface; it is possible to indicate the events that each component listens and how the user interface responds to them (by changing the interface components or doing application calls).

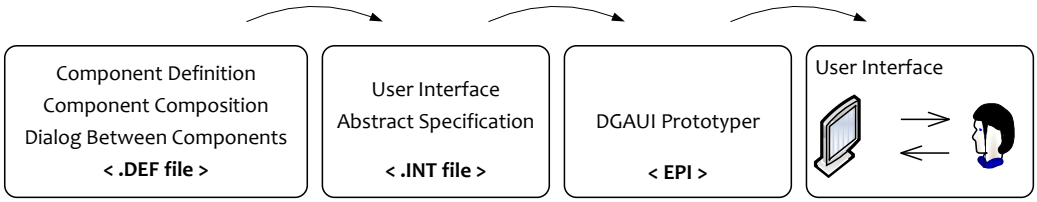

Figure 23: DGAIU interface building process.

To create an interface using the DGAIU method is necessary to follow several steps, briefly represented in (Figure 23). Initially, the interface designer creates visual components, their composition and the dialogs between them. The result is a XML document – the *.def* file. Then, from that document, another one is generated: the *.int* file. This file contains the visual states and the transitions between them. Finally, using an adequate interface prototyper (with the ability to read a *.int* file) the user interface prototype is generated and the user can start interacting with it. This process of creating an interface using the DGAIU system is explained below.

3.3.1 Component Definition (DGAIU-DEF)

The components definition step takes place using XML as a support structure, which is divided in three parts: components visual representation; components composition and dialog between components.

Components Visual Representation

This part is responsible for the components description, which can be described in three ways: *graphic* (based on graphical primitives such as line, rectangle or ellipse), *text* (based on text primitives) or *bitmap* (based on pixels).

DGAIU XML Tags	*In English*	*Description*
Activo	Active	Component activated (yes/no)
Alineado	Aligned	Component aligned (yes/no)
AnchoLinea	LineThickness	Line tickness in pixels
AnguloFin	EndAngle	End angle in pixels
AnguloInicio	StartAngle	Start angle in pixels
Cambio	Change	Used to change a state
Circulo	Circle	Design a circle
ColorFuente	FontColor	Text font color
ColorLinea	LineColor	Component line color
ColorRelleno	FillColor	Component fill color
Componente	Component	To indicate a component
Composicion	Composition	Components composition
Continua / Discontinua	Continuous / Discontinuous	Line style
Coordenada	Coordinate	Component coordinates
Datos	Data	To support data types
Decimales	Decimal	To support decimal data types
Descripcion	Description	Available components in a given state
Dialogo	Dialog	To represent a dialog
Elipse	Ellipse	Design an ellipse
Enumeracion	Enumeration	Used to represent a file by enumeration
Equiespaciado	Equispaced	Identical spacing between components
Estilo	Style	Continuous/discontinuous (LineStyle)
EstiloFuente	FontStyle	Text font style
EstiloLinea	LineStyle	To represent line style
Fichero	File	Represent an external file (e.g. an image)
Fija / Relativa	Fixed / Relative	To represent component positioning type
Fuente	Font	Text font
Grafico	Graphic	To represent a graphic primitive
ItemDialogo	DialogItem	To represent a dialog item
Linea	Line	Design a line
Longitud	Length	Primitive length
Nombre	Name	Component name
OpcRel	OpcRel	Option relative (Centred / justified)
Poligono	Polygon	Design a polygon
Posicion	Position	Component position
Precondiciones	Preconditions	Preconditions to state changing
Radio	Radius	Circle radius
RangoInf	InfRank	Component inferior rank
RangoSup	SupRank	Component superior rank
Rectangulo	Rectangle	Design a rectangle
Respuesta	Response	Used in conjunction with the dialog
Subcomponentes	Subcomponents	To represent subcomponents
Tamano	Size	Graphic component size
TamanoFuente	FontSize	Text fontsize
Texto	Text	To represent text
Visible	Visible	Component visible (yes/no)

Table 12: Some DGAIU XML tags description.

All components have 3 properties: *name*, *size* and *position*, and also have two important component attributes, which are: *visible* (indicates the component visibility on the interface) and *active* (indicates if the component responds to the established events or not). The component start position and size can be fixed or relative to another component (usually the container component).

An IDE (*Integrated Development Environment)* was designed to support the DGAIU XML specification (Table 12). It is called DGAIUDE (Figure 24) and it allows generating visual user interfaces from basic geometric shapes. It supports the establishment of transitions between global visual states, from events triggered over the interface components. In DGAIUDE a *tool palette* is available having tools like: *text*, *rectangle*, *circle*, *ellipse* and *enumeration*. It is possible to access and to change the properties of each component using the appropriate *properties palette*. Depending on the component type, its properties are available, such as: *active*, *colour*, *font*, *name*, *visible*. Other operations with the components supported by the IDE are: *elimination*, *copy*, *paste*, *alignment* and *size adjustment*.

As indicated, the visual user interfaces are saved under a XML structure according with the DGAIU notation.

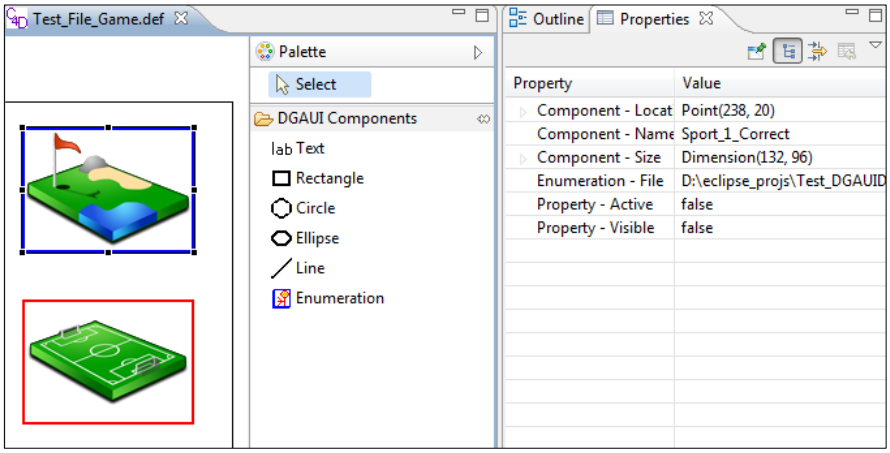

Figure 24: Partial example of the DGAIUDE environment.

Following, an example of a component using the DGAIU representation in XML is presented. The component was created based on an image file (*GF-Correct.png*) correspondent to an enumeration primitive. Some of the properties able to be set are its *position* on the *xy* axis and its *size*.

```
<Enumeracion Nombre="Sport1-C" Visible="t" Activo="t">
  <Fichero>D:\DGAIUDE\imgs\GF-Correct.png</Fichero>
  <Posicion>
      <Relativa>
          <Coordenada>
              <Px>238</Px>
              <Py>20</Py>
          </Coordenada>
      </Relativa>
  </Posicion>
  <Tamano Tipo='fijo'>
      <Valorx>132</Valorx>
      <Valory>96</Valory>
  </Tamano>
</Enumeracion>
```

Components Composition

It is possible to establish the components spatial composition. The DGAIU method allows to establish the abstract position of a component in relation to other (e.g. *center*, *right align*). The IDE supports creating composition relationships between components. This can be done by directly dragging the components to the design area, or by using the *outline palette*. The following lines represent the partial example of a component composition, using the DGAIU representation in XML. It is possible to verify a hierarchy of *SCs*: *background* with its sub-components.

```
<Composicion>
  <Componente Nombre="Background">
      <Subcomponentes>
          <Cont>Ball1</Cont>
          <Cont>Ball1-S</Cont>
          <Cont>Ball1-C</Cont>
          <Cont>Ball2</Cont>
          <Cont>Ball2-S</Cont>
          <Cont>Ball2-C</Cont>
          <Cont>Ball3</Cont>
          <Cont>Ball3-S</Cont>
          <Cont>Ball3-C</Cont>
          <Cont>Sport1</Cont>
          <Cont>Sport1-C</Cont>
          <Cont>Sport2</Cont>
          <Cont>Sport2-C</Cont>
          <Cont>Sport3</Cont>
          <Cont>Sport3-C</Cont>
          ...
      </Subcomponentes>
  </Componente>
</Composicion>
```

Dialog Established Between Components

The user interaction with the interface is here represented. Visual changes resulting from an event triggered over a component itself or over other components are here established. In the IDE it is possible to establish which previously created components respond at which specific events (e.g. *LeftClick*, *RightClick*, *MouseOn*, *Keys*). The user may specify preconditions and events through one of the two possible event types:

- *Component Event*: event concerned with user interaction over visual components (e.g. with the mouse or the keyboard);
- *System Event*: no predefined event. It is only applicable to the interface in general and not to the components.

The relation between an event over a component or over the interface is called a *dialog element* (Carnero 2007). The IDE guarantees the inexistence of two or more identical *dialog elements* (same event for the same component or interface, with different behaviour). Following, a partial example of a component dialog using DGAIU representation in XML is listed. The example represents a component behaving like a button, by accepting events. In this case, the component (*Sport1*) is waiting for the specific mouse event (*LeftClick*). It is also possible to represent preconditions to be met, in order that component answers might occur (*Respuesta*).

```
<ItemDialogo Elemento="Sport1" Evento="LeftClick">
  <Precondiciones>
      <Precondicion Visible="t" Activo="t">
          Ball3-S
      </Precondicion>
  </Precondiciones>
  <Respuesta>
      <Cambio Visible="f" Activo="f">Ball3-S</Cambio>
      <Cambio Visible="t" Activo="t">Ball3-C</Cambio>
      <Cambio Visible="f" Activo="f">Sport1</Cambio>
      <Cambio Visible="t" Activo="t">Sport1-C</Cambio>
  </Respuesta>
</ItemDialogo>
```

In order to describe *dialog*, two properties are included in components representation:

- *Activo (Active)*: its value can be (*t=True*) or (*f=False*). When a component has (*t*) as the value, that indicates it may respond to the associated events;
- *Visible*: its value can also be (*t=True*) or (*f=False*). The (*t*) value determines that component has *visual appearance* perceptible by the user, until other component completely hides it (in this case, the user cannot realize to be in the presence of the

component until the *visible* value be *true*). This is similar to a 3D representation where a tri-dimensional object is included inside another opaque object (the user cannot see the object inside, even if this has a shape, a colour and a texture).

The final interface obtained from components definition results on a *.def* file, which is automatically generated by the DGAIUDE environment.

An interesting feature available in DGAIUDE enables the possibility to create component libraries, which can later be used to build new visual interfaces. The user interface designer only has to select an abstract interface definition file (*.def*) and to choose the components he wants to use, in order to build the new interface. The DGAIUDE also has a tool to generate the interface visual states and transitions, resulting on the *.int* file.

3.3.2 Generation of States and Transitions (DGAIU-INT)

The second step of the interface building process concerns states and transitions generation. This step takes place after the prior component definition step. A *.int* file is obtained by storing diverse information concerning components visual representation contained in each state, components composition and transitions (between visual states) as a result of the dialog set in the *.def* file. An interface state obtained at a given time, results from the combination of components indicating that it is waiting for a user event (excluding the final state). From *active* and *visible* properties values, used in components description, it is possible to establish an initial interface visual state, which is given by the combination of all components having the two properties with the values *active=True* and *visible=True*. The list of events, associated with the components and their answers, are presented in the dialog description.

$$E_x = C_0, C_1, \ldots, C_m \forall C_i \nexists\, Visible(T)\ \ y/o\ \ Activo(T)$$

DGAIU considers that from events over components it is possible to deduce the events over states and hence the next interface state, when an event over the actual interface state is triggered. From initial state it is possible to obtain all possible following states, through triggering each one of the possible events on the active components of that initial state. From the generated states, and applying the same process, the following states can be obtained, until arrive a final state, in which all interface components have the properties (*active=False*) and (*visible=False*). During the interface states obtaining process, transitions

can be determined by the events triggered by the user over existing states. Two states are equal (and hence, the same state) if all its components have equal values of (*active*) and (*visible*) properties. The components that are part of a visual state are those which have functionality within the state. This functionality can be given by:

- a *visual appearance* that provides relevant information to the user;
- a response to an event associated with the use of a particular area of the screen;
- any component belonging to a component container.

During the obtaining states process, it is possible to determine the transitions between them. To represent these transitions (identifying the user events that trigger them) a format to indicate a state, an event, the component on which the event is applied and the new generated state is used. Each transition may have an associated set of preconditions to allow this to occur. A small excerpt of a *.int* file is presented below.

```
<Transiciones>
  <Transicion EstadoInicial="0" Elemento="Ball1"
    Evento="LeftClick" EstadoFinal="1" Alcanzable="t"/>
  <Transicion EstadoInicial="1" Elemento="Ball1-S"
    Evento="LeftClick" EstadoFinal="0" Alcanzable="t"/>
  <Transicion EstadoInicial="0" Elemento="Ball2"
    Evento="LeftClick" EstadoFinal="2" Alcanzable="t"/>
  <Transicion EstadoInicial="1" Elemento="Ball2"
    Evento="LeftClick" EstadoFinal="2" Alcanzable="t"/>
  <Transicion EstadoInicial="0" Elemento="Sport1"
    Evento="LeftClick" EstadoFinal="4" Alcanzable="f">
        <Precondiciones>
            <Precondicion Visible="t" Activo="t">
                Ball-3-S
            </Precondicion>
        </Precondiciones>
  </Transicion>
  ...
</Transiciones>
```

Following, a simple prototyper which supports the DGAIU specification is introduced. It is the tool used to generate the prototype of a concrete visual user interface representing the test bed user interface described in this chapter.

3.3.3 DGAIU Prototyper

The next step taken was to create the prototype of a concrete user interface, with which a user could interact. The interface design was previously obtained using DGAIUDE environment, and following a *.int* file containing the abstract interface specification was

generated by that tool. From the generated specification, it is showed below an example of a concrete component (DGAIU rectangle primitive) which could be available on a *.int* file, in order to be used to create the prototype of a complete and functional visual user interface. Basically, the following specification allows to visually represent a concrete rectangle primitive having width, height, position (x, y 2D coordinates) and being a continuous line with a defined thickness, a border and a fill colour. All other *SCs* included in a *.int* file have a similar specification. Following, it was necessary to use an interface prototyper (supporting DGAIU specification) in order to create the game interface, obtained from its abstract representation in XML (Figure 25).

```
<Grafico Nombre="Background" Visible="t" Activo="t" InfI="f"
  InfO="f">
  <Rectangulo>
      <Coordenada>
          <Px>0</Px>
          <Py>0</Py>
      </Coordenada>
      <Coordenada>
          <Px>400</Px>
          <Py>400</Py>
      </Coordenada>
  </Rectangulo>
  <EstiloLinea Estilo='continua' />
  <AnchoLinea>1</AnchoLinea>
  <ColorLinea>000000</ColorLinea>
  <ColorRelleno>ffffff</ColorRelleno>
  <Posicion>
      <Relativa>
          <Coordenada>
              <Px>50</Px>
              <Py>50</Py>
          </Coordenada>
      </Relativa>
  </Posicion>
  <Tamano Tipo='fijo'>
      <Valorx>400</Valorx>
      <Valory>400</Valory>
  </Tamano>
</Grafico>
```

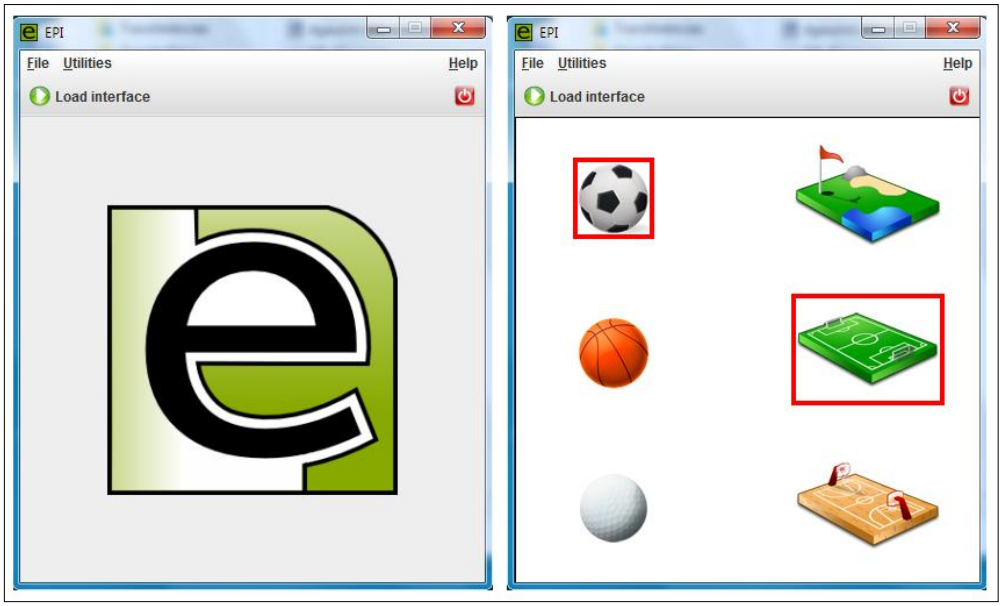

Figure 25: Example of using DGAIU Prototyper (EPI).

The prototyper used was EPI (*Interface Proof Evaluator*) (Carnero 2007). The system has several features, such as:

- the ability to read and to parse a XML file containing the specification of a user interface prototype;
- from the specification, a concrete visual user is represented in the visible window;
- a module manages the interaction between the user and the user interface prototype. Thus, the user may start to interact with the visual components (using the supported physical interaction device indicated in the specification), triggering the global visual states available for that specific visual user interface.

The original EPI prototyper supports several other features that were not need to use to validate the visual user interface described in this chapter. Thus, some changes were performed in the original code, in order to disable some of those features and to enable other specific features to support the test bed user interface here presented.

Initially, the prototyper loads the *.int* file containing the visual interface specification, and interprets it. Then, the initial state of the game interface previously designed in DGAIUDE is putted at user disposal. The user may start to interact with the interface components, which in this game example can be done using the mouse to trigger events to select/deselect the sport balls and the sport fields, in order to complete the game objective.

3.4 Conclusions

In this chapter, an example of a user interface was created, with significant functionality details to be used in following steps of this research and also to be a reference to other authors to test and to validate its methods. From the study here described, a functional visual user interface was obtained. The DGAIU system was selected to implement a 2D visual game interface and two of the DGAIU tools were used, supporting free design of components, independently of toolkit or programming language. One of the tools was used to design the visual interface (DGAIUDE). The other tool (the EPI prototyper) allowed implementing a concrete prototype of an interactive simple visual game for younger children. After the 2D visual game interface had been specified using *SCs*, and a prototype be created, it was possible to verify and to conclude that the level of detail needed to specify visual interface components in DGAIU is too deep and the specification becomes too complex for the size of actual direct manipulation interfaces, becoming impracticable the extensive use of it. Thus, an opportunity arises to develop a concept that allows complexity decrease on components design and behaviour definition processes.

The game interface previously described includes input and output events over *SCs* with its own actions. Hence, the following step is to develop a new component concept that aggregates these features, involving visual presentation, composition and interaction with the user. That will allow increasing abstraction on the interface representing process and consequently to allow simplifying a user interface representation. Next chapter is dedicated to the presentation of the originally proposed *complex component* concept.

Chapter 4

Complex Component Concept Proposal

4.1 Introduction

Previously is indicated that a user interface can be specified by considering several *interface design steps* (Rodeiro 2001), represented by a proper user interface specification. These steps correspond to models which describe different aspects of the user interface. There exists work available in the field of interactive graphics, which involves describing the interface of a system as a collection of *interaction objects* (components). This approach applies both to the implementation level, when the term *widget* is typically used (Myers 1990), and to a more abstract level of the system specification (Jacob 1986). Although most of today software systems use more or less complex components to build their interfaces, it was not found in the literature any approach to components which usage could be used to represent the complete interface functionality, based in free design of components. Thus, it is intended to characterize a new interface component concept, by establishing the features that it should support (especially in relation to its visual appearance, component composition and interaction). Also, in order to simplify the specification of complete and functional visual user interfaces, according to the *CFFI* criteria (cf. Chapter 2) the union mechanisms between those interface components will be determined.

The study presented in this chapter starts by analyzing the three *AIOs* introduced in Chapter 2, considering its visual presentation and interaction features, regarding to include some of them in a new *complex component* concept, introduced and detailed in the

following section. The main features that a *complex component* abstraction should include, in order to build visual user interfaces, are identified, considering the characteristics analysed for the 3 *AIOs* and expanding the concept by adding new characteristics to it, in order to achieve the *CFFI* criteria. Additionally, an example of a simple interface design process using *complex components* is presented, focused in *interaction* between components. Furthermore, the analysed *AIOs* and the new *complex component* concept are compared with the well established object-oriented paradigm.

4.2 AIOs Visual Presentation and Interaction

The three *AIOs* previously considered in Chapter 2 are focused in interactive components representation to create user interfaces. It was possible to verify those *AIOs* do not support components composition, but some of them support visual presentation and interaction features. Thus, a comparative analysis between the three *AIOs* was done, concerning:

- Visual presentation and visual states;
- Input/output from/to user;
- Input/output from/to other components;
- Input/output from/to application;

A brief description of these properties regarding *visual presentation* and *interaction* are summarised in (Table 13 to Table 20). The first two tables (Table 13 and Table 14) are concerned with the visual components presentation, whereas the remaining 6 tables refer to the bi-directional *interaction* from/to an *interaction object*, considering that in general, three elements may interact with it: the *user*, another *interaction object* and the *application*.

Visual States Presentation

interactor (a)	no visual presentation is supported (it is algebraically represented, instead of showing a visual renderer).
abstract data view (ADV) (b)	different *ADVs* may view the same *ADO*, since interfaces can support alternate "views" of data (or *interaction* modes). An *ADO* has no knowledge of its interfaces (*ADVs*). The *ADVs* do not support visual representation.
virtual interaction object (c)	it acts as a supporting tool for the visual (and non visual, if necessary) components of the presentation. This presentation is provided by some interface toolkits described in this specification (e.g. *MFC*, *JFC*).
(a) does not consider visual presentation, but *(c)* does. In the case of *(b)* its visual presentation depends on the *ADO* which is connected to.	

Table 13: Visual Presentation.

interactor (a)	it has states which are not visual. Despite having a renderer function, *interactors* do not allow visual representation.
abstract data view (ADV) (b)	one *ADO* can be represented by several *ADVs*. It state is changed according with the *interaction* of the user over the *ADVs*. An *ADV* can change its visual state by receiving an action from the *ADO* which is connected to, at any given time. *ADVs* and *ADOs* represent a two-way relationship.
virtual interaction object (c)	it allows all possible visual states provided by the components of the interface toolkit(s) being used.
Not all techniques consider the existence of visual states (e.g. *(a)*); *(c)* is limited by the features of the considered toolkit; *(b)* consider visual states in their properties.	

Table 14: Visual States

Components Interaction

interactor (a)	it receives (and accumulates) input from the user and provides feedback to the user (*stimuli* receptor).
abstract data view (ADV) (b)	input events exerted on an *ADV* when acting as a user interface (*causal actions*).
virtual interaction object (c)	it refers to the device-level user-input. Input events can be associated with event handlers (input event classes).
Feature supported by all AIOs.	

Table 15: Input from User.

interactor (a)	it sends output to the user (response events) from an output trigger.
abstract data view (ADV) (b)	an *ADV* sends output actions such as display commands and changes its *appearance* as a result of an input message or a change in the state of an associated *ADO* (output messages).
virtual interaction object (c)	similar to function calls, but simplified by the toolkit model implemented. Output events can be "called" in a similar way to the conventional function calls (output event classes).
Feature supported by all AIOs. In case of *(c)* is limited to the events provided by the toolkits.	

Table 16: Output to User.

interactor (a)	objects can communicate by synchronizing on specific events (in the intersection of their events' alphabets; cross-synchronization).
abstract data view (ADV) (b)	*ADVs* are interconnected with *ADOs* (the former do not exist without the latter). The communication is achieved between *ADOs* (attributes mapping and parameters transference).
virtual interaction object (c)	functions of the virtual and lexical expressions that can be called, but the virtual interaction can only be performed when it can be publicly accessed.
All three AIOs allow for component inputs from other components. *(b)* has the particularity of its inputs only be managed by the *ADO* with which it is connected to.	

Table 17: Input from other Components.

interactor (a)	objects can communicate by synchronizing on specific events (in the intersection of their events' alphabets; cross-synchronization).
abstract data view (ADV) (b)	the *ADVs* are interconnected with *ADOs* (the former do not exist without the latter). The communication is achieved between *ADOs* (attributes mapping and parameters transference).
virtual interaction object (c)	it broadcasts an output event to the toolkit server which then manages the component events (output event statement).
All three AIOs allow for component outputs to other components. *(b)* has the particularity of its output only be managed by the *ADO* which it is connected to.	

Table 18: Output to other Components.

interactor (a)	it receives (and accumulates) output from the application (receptor of *stimuli*).
abstract data view (ADV) (b)	Inexistent; the *ADVs* and the *ADOs* are interface objects which communicate among themselves or with an "external" system such a user, a network or a timer.
virtual interaction object (c)	"*For instance, the developer may wish to generate a mouse click for a particular object artificially, while such event would be treated as if it was real mouse click*" (Savidis 2004a) (artificial event broadcasting).
This feature is not contemplated by *(b)* and *(c)*.	

Table 19: Input from the Application.

interactor (a)	it receives an input trigger which results in the *interactor* sending the accumulated input to the application (response events).
abstract data view (ADV) (b)	Inexistent; the *ADVs* and the *ADOs* are interface objects which communicate among themselves or with an "external" system such a user, a network or a timer.
virtual interaction object (c)	this technique maps the specification of the toolkits, but it does not perform a similar action from the components to the application.
(b) and *(c)* do not provide the ability for the *interaction objects* to communicate with the application to which it belong.	

Table 20: Output to the Application.

	Interactor	***abstract data view (ADV)***	***virtual interaction object***
Visual Presentation		×	×
Visual States		×	×
Input from User	×	×	×
Output to User	×	×	×
Input from Other Components	×		×
Output to Other Components	×		×
Input from the Application	×		
Output to the Application	×		

Table 21: *AIOs* analysis summary.

A cross comparison between the three different *AIOs* (Table 21) allows to verify that none of the *AIOs* completely supports the analysed features regarding visual presentation

and interaction. This observation, together with a previous one, in which none of these three *AIOs* also supports components composition, enhances several characteristics that may be established, when considering a future *AIO* concept. In the following section, a new *complex component* concept is introduced, considering the features analysed for the three *AIOs* and expanding the concept by adding new characteristics to it.

4.3 Complex Component

The prototype of a test bed user interface was created (Rodeiro & Teixeira-Faria 2011): a simple game for younger children containing several visual elements, including sport balls and sport fields (cf. Chapter 3). After the game's interface has been implemented, it was considered to use a more complete interface component. The characteristics that a new interface component should have are not totally satisfied by the previous analysed AIOs, concerning visual presentation and interaction. Therefore, a hypothesis has emerged and it consists in the introduction of a new interface component, the *complex component* (Teixeira-Faria & Rodeiro 2011), in order to use it to simplify the specification of a visual user interface. The *complex component* should has the ability to be integrable in any specification, as long as it be able to support visual presentation, components composition and user interaction.

Usually, a visual and interactive component can be individually established, supporting relationships with other components within the interface. Basically, the new *complex component* consists of several other components (*simple/complex components*) which interact with each other through *self* and *delegate events/actions* (see below) and which work towards a common goal (e.g. a *toolbar* allows a user selecting a specific tool to perform a certain task at any given time). Also, this *complex component* concept follows a hierarchical topological structure, so that a component container (not necessarily visual, represented by a dashed rectangle in Figure 26) includes a number of other visual *simple/complex components* (e.g. in Figure 26, three *simple components* are represented by one *complex component*). Besides acting as a container, a *complex component* also has its own visual states and the ability to interact with other external *simplex/complex components*.

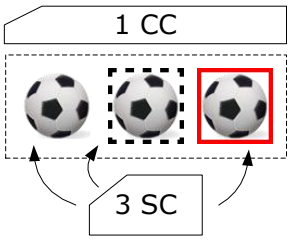

Figure 26: An example of a *complex component* containing three *simple components*.

A *complex component* should support dialog (between the user and the component, and also between the components). Otherwise, it would just merely be components having *visual appearance* and *topological composition*. By using *complex components* to design visual user interfaces supporting direct manipulation, is possible to define one or more different activation spaces (interaction areas). These areas should have a visual significance, i.e. they should be visual components. One example of a complex component with just one activation area is the traditional *button* (with several visual states). And, the user usually interacts in the same physical area. On the other hand, a *toolbar* can be defined as a component with more than one *interaction* area (visual options) which is related to different spatial positions.

Some of the advantages of *complex components* usage are:

- Reduced user interface design complexity;
- Allowance for reusing components;
- Allowance for individual specification of the UI parts;
- Use tasks decomposition similar to user tasks model.

To specify a user interface using *complex components* results in decreased interface complexity because this type of components increases the abstraction level to specify it, due to the fact that the interface design process can be subdivided into small parts represented by the *complex components*. Additionally, by the fact that a *complex component* does not depend on the interface where it has been created, it can be reused in different visual interfaces contributing to simplify the interface specification process. This means that the interface designer can reuse a *complex component* (with a specific behaviour) from one interface to another by keeping the functionality, but showing a different *visual appearance*. Also, he can reuse the same visual appearance provided by a *complex component* and to redefine its behaviour. Each *complex component* can be individually specified, as a UI part, simplifying the overall *global interface* specification process. Therefore, by individually establishing *complex components*, also is possible to verify them separately. Regarding the

complex component architecture, each component can be decomposed into a hierarchy of components, with specific properties.

4.3.1 Complex Component Features

This section details the features concerned with the *complex component* concept. Its characterization is divided in three sets of features:

- Visual Appearance;
- Topological Composition;
- Interaction.

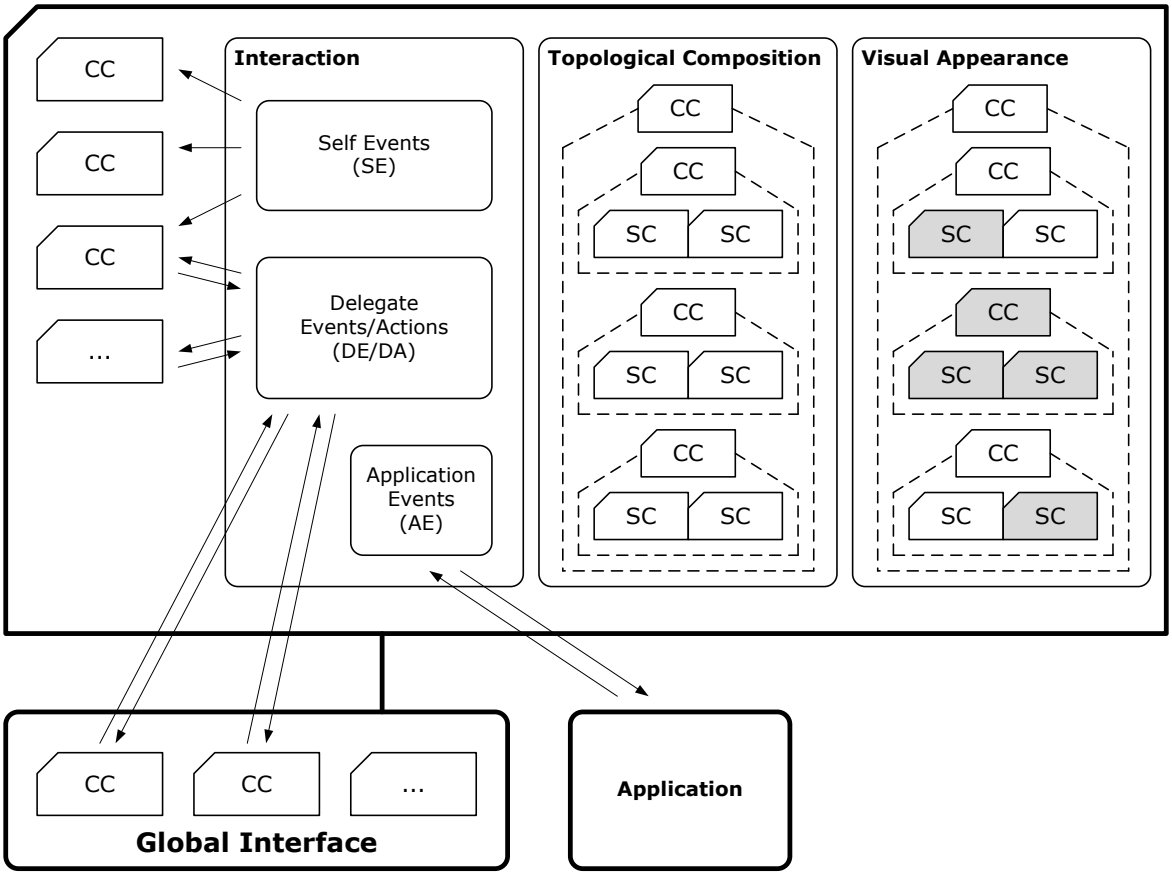

Figure 27: *Complex component* typical structure.

The features can be related with some of the *interface design steps* (Rodeiro 2001) (cf. Chapter 2). Typically, the *visual appearance* is indicated in the *abstract presentation model*, which includes the visual description of each component. The *topological composition* is included in the *abstract composition model*, which defines the spatial relationships between components (which may establish the spatial rules between components inside the container, but also in relation to positioning the container inside the *global interface*). The

interaction is concerned with the interface components (*dialog*) and is established in the *dialog model*. The structure of a *complex component* is represented in Figure 27. It is arranged according to the three sets of features considered in its definition. A *complex component* manages *visual appearance* of *simple/complex components* and the *topological composition* of those components contained inside of it (due to its characteristics of component container). Furthermore, a *complex component* also supports the *interaction* either between the components inside of it or between that *complex component* and components external to it.

Visual Appearance

In Figure 27, the *visual appearance* is depicted by the rectangle with rounded corners, on the right side of the figure. In the example represented by this figure, the *complex component* contains, inside it, three other *complex components* (each one containing two *simple components*). Every *simple/complex component* has several properties and one of them is the *visibility* (*visible* (coloured in white) or *invisible* (coloured in gray)) which allows to determine the component visibility. A *complex component* encapsulates several possible visual states, obtained from the *complex component* visual presentation properties. The *global interface* visual appearance is derived from integration of visual appearance of each *complex component* at a given time (the visual state that each *complex component* shows at that moment). Thus, from a *global interface* perspective it is possible to know which components are visible at any given time (obtained as a result of events acting on the components) and as well the overall *global interface* visual states.

Topological Composition

The *complex component* concept considers components *topological composition*. It is a hierarchical structure and in the example given in Figure 27 (the rectangle with rounded corners, in the middle of the figure) several (*simple/complex components*) are represented, and each one can be contained within others. In the figure is illustrated the topological composition of 4 *complex components* and 6 *simple components*, arranged in three topological composition hierarchy levels. The *complex component* on the top corresponds to the first hierarchy level.

Interaction

The *interaction* involved in a *complex component* can be distinguished in two: one occurring inside the *complex component* and the other occurrs between the *complex component* and other components outside that *complex component* (as indicated by the rectangle with rounded corners, on the left side of Figure 27). Therefore, additionally to the *visual appearance* and *topological composition*, a *complex component* triggers and receives events of three different types: *self events*, *delegate events/actions* and *application events*. It can be described as follows.

Self Events (SEs)

These events are encapsulated in a *complex component* and its internal functionality is responsible for triggering *delegate actions* (explained below). These *delegate actions* will be responsible for triggering *delegate events* (explained below) that other *complex components* will receive. A *complex component* supports visual transitions (changes from one visual state to another visual state) between the contained components (*simple* and/or *complex*), which affects the visual appearance of that *complex component*. These transitions are triggered by *SEs* resulting from user interaction with the *complex component*.

Delegate Events/Actions (DE/DA)

These events (*DEs*) are established inside a *complex component* and are responsible for changing visual appearance of that *complex component*. The events are triggered from other external *complex components*. That other *complex component* can be part of the same group (at same level) of the component which receives the trigger or it can be other *complex component* outside that group and available at a different level in the user interface. A close related concept is the *delegate action* (*DA*), also available inside a *complex component*. A *DA* has a functionality which is dependent of some visual state (of that *complex component*) be obtained. When that occurs, the *DA* triggers the corresponding *DE* on other *complex component*. The detail of these internal *DAs* functionality is occluded from the interface designer who just needs to establish the connection between *complex components* through their *DEs*. Both *DEs* and *DAs* are together a union mechanism concept regarding communication and functionality between *complex components*.

Application Events (AE)

It refers to dialog processes established between the *complex component* and the application (core), i.e. the *complex component* can send/receive events to/from the application.

Events and Actions

Some of the studied *AIOs* (e.g. *interactor* and *ADV*) are concerned to *event synchronisation,* i.e. communication between events. A question that might arise is to what extent an event should be considered as a programmed action of a component over another, since the events are external to the interface, either originated from the user or from the system. Duke & Harrison (1993) established an event definition: *"an abstraction of some action or change within a system, for example, that a mouse has been moved, a button clicked, or a warning tone sounded... Formally, the set of all events is represented by a 'given' event set*". Alencar et al. (2002) indicated that *"in the (ADV/ADO) approach, object interaction is modelled by a mapping mechanism. That mapping mechanism defines morphisms between public attributes and actions of an ADO and the corresponding attributes and actions of an associated (ADV). A second mode of interaction is by means of actions, which are the programming functions and input events that act on an object to change or query its state*". Therefore, it becomes important to emphasise that the terms *event* and *action* might have a different significance, depending on when the *interaction object* component is acting as a sender or as a receiver at any given moment of the *interaction*. If a component behaves as a receiver, it means that it receives *events*. Otherwise, if a component behaves as a sender it means that it invokes procedures to perform a set of *actions*. In any case, a *complex component* will always interpret a change on its properties in the same way, regardless of wherever the event/action has originated from. Additionally, is important to refer that if a *complex component* has *DEs*, it are triggered from *DAs* available in other complex component (consequently, *DEs* are dependent of *DAs)*.

Relation Between Events

The *SEs* are encapsulated in a *complex component* and its internal functionality is responsible for triggering *DAs*, which will then be responsible for triggering *DEs* that other *complex components* will receive. The details of the *DAs* functionality is occluded from the interface designer who just needs to establish the connection between *complex components* through their *DEs*. The *interaction* process with a *complex component* is initiated by the

user interacting with the component (using an *interaction* device such as the mouse in a *GUI* or a finger in a tablet). From that interaction it could result the trigger of a *SE*, a *DE* (or simultaneously *SE* and *DE*) or an *AE*. The occurrence of a *SE* or a *DE* causes a visual transition in the *complex component* and hence, in the *global interface*. However, some global visual transitions may only occur through the simultaneous triggering of both *SE* and *DE*. It is possible that repeated interface visual transitions may occur, due to the triggering of different events from different *complex components*. However, this does not increase the number of valid interface visual transitions, since these are considered only once.

Visual States

A *global interface* has a finite number of possible discrete visual states, which are obtained in result of user interaction. Those visual states can be classified into three types, due to events acting in *complex components*:

- ***Self visual state***: this state is obtained in direct result of user interaction with a *complex component*, triggering a *SE*;
- ***Delegate visual state***: state obtained from the triggering of a *DE* available in the *complex component*; other *complex component* is responsible for triggering that *DE*;
- ***Simultaneously self and delegate visual state***: a state of this type is obtained from the simultaneous trigger of at least on *SE* on a *complex component* and a *DE* on other *complex component*.

The user *interaction* with a *complex component* may trigger *self visual states* and/or *delegate visual states*. If the occurrence of a visual state transition inside a *complex component* is dependent on the visual state of other *complex component*, the transition only occurs when the criteria is fulfilled. Otherwise, no global visual transition is executed. (Table 22) summarises the range of possible *self* and *delegate visual states* which originate variations in the global visual states. According to the table, a user *interaction* may lead to the achievement of (0) *self visual states* and (0), (1) or several *delegate visual states*. Also, may result in (1) *self visual state* and (0), (1) or several *delegate visual states*. Both situations could imply a change in the *global interface* visual state (one global visual change occurs for each user *interaction*). The *global interface* visual state integrates all visual states of individual *complex components* used in the interface design.

Self Visual State	*Delegate Visual State*	***Global Visual State***
0	0	**0**
0	1	**1**
0	Several	**1**
1	0	**1**
1	1	**1**
1	Several	**1**

Table 22: Global interface visual states in result of interaction with a *complex component*.

Taking into account the close link between visual states and events, the *interaction* with a *complex component* may result in none or one *SE* and/or none, one or several *DE*s. This means when the user interacts with a *complex component*, a series of events can be triggered. Therefore, the apparent complexity generated by the high number of potential events is simplified by using *complex components*. This happens because the whole *interaction* process between *complex components*, which triggers transitions in result of *DE*s, is being currently perceived by the interface designer allowing the abstraction from the internal functionality of the components. Thus, once a *complex component* is created, it can be used in the interface design as a "black box" (term also used by (Alencar et al. 2002)) where the *interaction* is the result of *SEs* and *DEs* (sending/receiving them to/from other components or to/from the system).

Visual Transitions

The events triggered on *complex components* are responsible for changing visual states. Those changes are called visual transitions and can be divided in two: the ones concerned to each *complex component* and the other concerned to the *global interface*. All are responsible for changing visual states at the global interface level. In Figure 28, several events (E_n) may trigger several visual transitions (*T1*, *T2*, *T3* and *T4*).

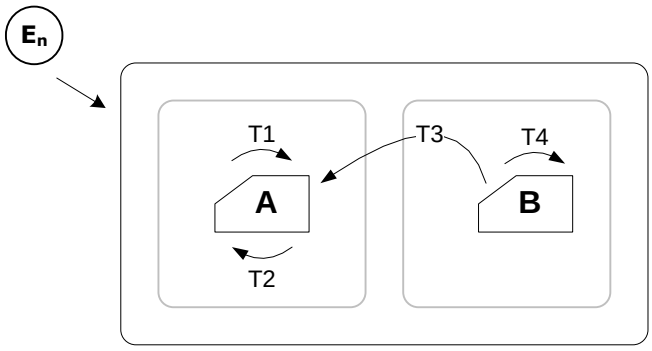

Figure 28: Visual transitions resulting from events/actions over two *complex components*.

Exemplifying with the previous interface example of a *complex component A* and a *complex component B* (Figure 28), *T1* and *T2* transitions could be initiated by 2 *SEs* over *complex component A* and *T4* transition could result from one *SE* over *complex component B*. *T3* transition represents the result of a *DE* received by the *complex component A*, which was triggered by a *DA* commanded by the *complex component B* (e.g. initiated by a *SE* in result of user interaction with this *complex component*).

4.3.2 Complex Component Abstraction Process

Designing hierarchically organised *complex components*, allows increasing the abstraction level when designing a visual interface. The method to obtain *complex components* by increasing their abstraction levels can be done through a process based on an iterative cycle. Before starting the process is necessary to establish an entrance level, in which *simple components* are identified together with user events and all possible visual transitions between them (those components will be responsible for the interface visual appearance). Then, the iterative cycle to abstract *complex components* follows 3 criteria:

- ***Criterion 1***: a *complex component* must have *simple/complex components* inside of it (internal components) and the (*SEs*) in result of user interaction with the components be identified;
- ***Criterion 2***: the identified *complex components* relate to each other through their input/output *delegate events/actions* which (or other conditions) are not totally satisfied;
- ***Criterion 3***: with respect to new *complex component* creation conditions, if the interface is not fully functional and some of the *complex components* still have input/output (*DE*/*DA*) or other conditions not totally satisfied (e.g. still waiting for some preconditions to be accomplished), the abstraction level can be increased through grouping components.

After the introduction of this *complex component abstraction process*, it was verified that is not possible to represent a complete and functional user interface through one *complex component*. This happens because ***Criterion 3*** of the abstraction process is not achievable, due to *complex component* concept not be a complete component. Thus, the *final interface* cannot be represented through a *complex component*. However, in order to respond to the interface conditions, all unions between *complex components* must be fully functional and the *final interface* will be obtained. This allows introducing a new concept,

the *complete complex component* (*CCC*), which represents the complete visual object, totally independent of any other interface components.

4.4 Abstraction Example

In order to exemplify some concepts previously introduced, it was decided to select some visual elements used in the test bed user interface presented in Chapter 3. A simple user interface example is introduced, concerning:

- the establishment of union mechanisms between *complex components*, in order to introduce communication between them;
- to obtain user interface design simplification, concerned with the number of user events, visual components, events classification and global visual transitions.

During a user interface design process, the *simple components* to be used need to be created together with all its possible visual transitions that can occur among those components.

Figure 29: Example of 3 global interface visual states.

In Figure 29, a part of a simple interface example is illustrated, with the images representing the process of selecting a ball and the related football field. The *interaction* takes place when:

- the user clicks in the ball (e.g. using a mouse as the physical interaction device) and that event changes its visual presentation to the *selected* visual state (enclosed by a dashed line) and by clicking again in the ball, it will return to its initial (*normal*) visual state;
- the user clicks in the sport field which results in the *correct* visual state (enclosed by a solid red line) and the simultaneous change in the ball visual state (also enclosed by a solid line).

Five *simple components* were used in this example (Figure 30).

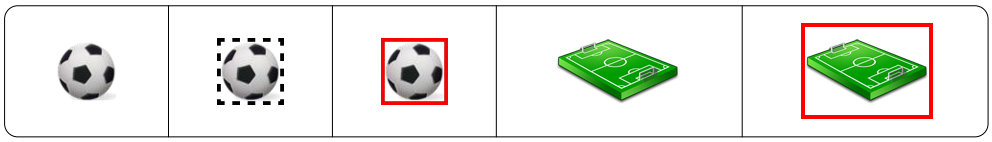

Figure 30: The five *simple components* used to illustrate the global visual states.

The abstraction process can then be established starting from the five *simple components*, which can be re-organised in two groups of *simple components*: the 3 balls and the 2 football fields, grouped in two *complex components*:

The *CC_BtnBall* composed by:

– *SC_BtnBall_N*: ball as the *Normal* visual state;

– *SC_BtnBall_S*: ball as the *Selected* visual state;

– *SC_BtnBall_C*: ball as the *Correct* visual state.

The *CC_BtnField* composed by:

– *SC_BtnField_N*: sport field as the *Normal* visual state;

– *SC_BtnField_C*: sport field as the *Correct* visual state.

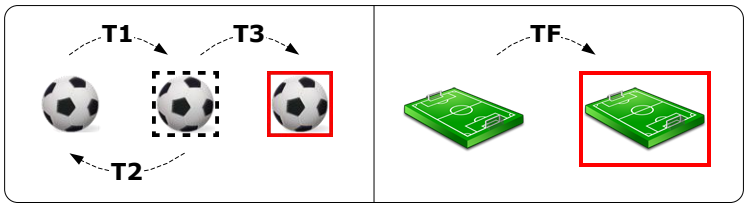

Figure 31: The four visual transitions (*T1*, *T2*, *T3* and *TF*).

Several possible visual transitions may occur between *simple components* of each group (Figure 31) in result of events triggered over the components.

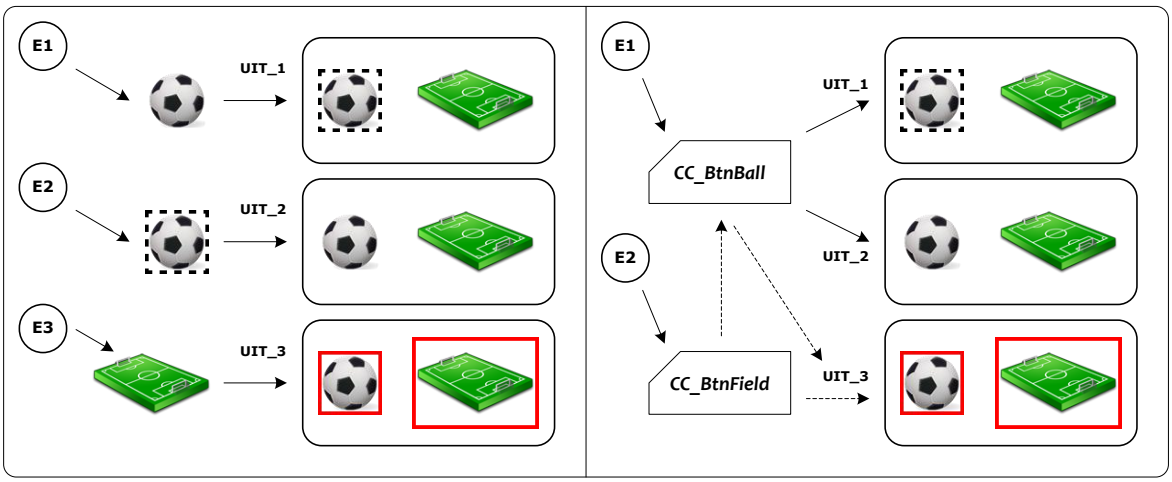

Figure 32: *Interaction* representation using *SCs* (on the left) and *CCs* (on the right).

In this example, by using *simple components*, each visual transition of the global interface corresponds to each event (Figure 32). However, when using *complex components*, an event can be related to more than one global visual transition activation (e.g. when the *E1* event is triggered, two global visual transitions occur (*UIT_1*, *UIT_2*)). Furthermore, an event involving a *complex component* may trigger it own *SE* and *DEs* (the linking mechanism between *complex components*) in other components (e.g. *UIT_3* transition will result in a *SE* in the football field and a *DE* in the ball).

The identification of user events, visual components, events classification, global interface visual transitions identification and global visual states involved in this example are listed in (Table 23). It is possible to verify a decrease in the number of events that the interface designer has to be care at representation level, when using *complex components* in comparison with using *simple components* (to the user of the interface, the components representation is visually the same).

	user events	***visual components***	***events classification***	***global transitions***	***global visual states***
simple components	E1	*SC_BtnBall_N*	*SC_BtnBall_N* event	UIT_1	3
	E2	*SC_BtnBall_S*	*SC_BtnBall_S* event	UIT_2	
	Ø	*SC_BtnBall_C*	Ø	Ø	
	E3	*SC_BtnField_N*	*SC_BtnBall_S* event	UIT_3	
	Ø	*SC_BtnField_C*	Ø	Ø	
complex components	E1	*CC_BtnBall*	2 *SE*	UIT_1, UIT_2	3
	E2	*CC_BtnField*	1 simultan. *SE* and *DE*	UIT_3	

Table 23: Visual interface representation using *simple* and *complex components*.

As stated earlier, the *complex component* concept assumes that its structure acts as a container for other components (*simple* and/or *complex*) and therefore higher or lower *complex component* functionality, can be determined by:

- the topological structure of the components contained inside the *complex component* (i.e. number of abstraction levels);
- the number of *SEs* that can be trigged by the *complex component* and consequently, the number of relationships established between components contained inside of it;
- the number of *DEs*/*DAs* resulting from the relationships established with other external *complex components* (a visual transition can also occur by the simultaneous triggering of a *SE* and a *DE*).

At the same time, higher is the number of events, consequently higher could be the number of components visual transitions that may occur, and hence, higher the number of global visual transitions. Taking the simple interface example previously described, only one *complex component* level was used (having two *complex components*) to respond to the final interface conditions (according to the above *complex component abstraction process*). Following, it is presented an example of a user interface having *complex components* increased by one abstraction level (Figure 33).

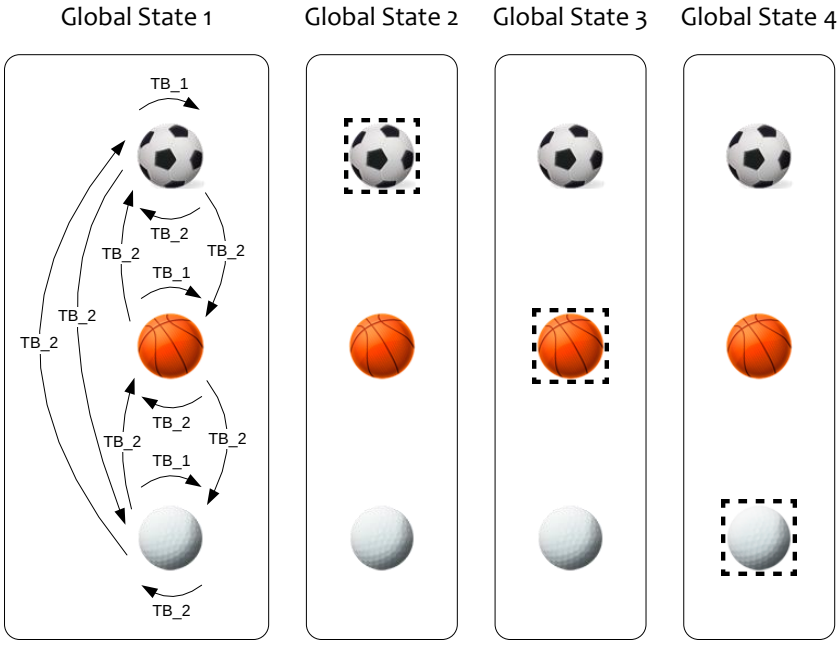

Figure 33: 4 global visual states obtained from 1 *complex component*, by grouping 3 other components.

The same previous *complex component abstraction process* was used and 3 *complex components* were obtained in the first abstraction level (representing 3 sport balls with its visual transitions, e.g. *TB_1* and *TB_2* of each ball). Then, a new *complex component* was abstracted, having the three balls inside of it (each one with their visual transitions). Those internal *complex components* are related between themselves through their union mechanisms: the *DEs*/*DAs*. This new *complex component* can be related with other *complex components*, if it still has input/output (*DE*/*DA*) or other conditions not totally satisfied. It is possible to observe by this example, the level of abstraction introduced by encapsulating *complex components*, allowing the interface designer to abstract himself from the components internal functionality, regarding communication between components. These communications are responsible for triggering components visual transitions, which will compose the interface global visual states.

4.5 Some Similarities with OO Paradigm

After the initial analysis made to the three existent *AIOs* and the new concept be introduced (the *complex component*) it was decided to verify to what extent the use of these different interaction objects could be comparable to a fully validated and widely used paradigm: the *object-oriented paradigm* (Coad & Yourdon 1990). Over the years, its robustness has been tested and validated by a growing number of programmers who use it. Thus, some *OOP* concepts were considered to be verified, such as encapsulation, polymorphism, inheritance, identity, state and behaviour:

- *Encapsulation* is a mechanism that usually brings together the code (*procedures*) and the data (*attributes*) and the internal representation of an object (usually hidden from outside view of object's definition). One benefit obtained from encapsulation is a reduction in the system complexity;
- *Polymorphism* is a Greek term which means "many forms". In *OOP*, it indicates the possibility of using an abstract interface or abstract class to create multiple objects by reusing procedures. *OOP* lets programmers create procedures for objects whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on the screen, in response to mouse movement, can be written for "cursor" and polymorphism lets that cursor take simulating system behaviour;
- *Inheritance*: is a common technique for reusing functionality in the *OOP* systems. It provides the ability to specify classes (object types) in order a new class contain a new series of characteristics, additionally to the structural and behavioural ones originally established. From an enclosing object perspective, *inheritance* is viewed as a "clear-box" since all contents and structures of the enclosed object are visible. *Inheritance* is particularly useful for designing abstractions, when a nesting object can build its structure based on an existing object design.
- *Identity* is what makes a certain object unique and identifiable, never used by any other component;
- *State* is the set of values encapsulated by an object and although two objects could eventually share the same identical *state*, they remain distinct, as separate identifiable objects;

– *Behaviour* is the set of actions which can be taken by an object (usually identified as *methods*).

The 3 previously referred AIOs and the new *complex component* concept are analysed under the *OOP* paradigm with a focus on supporting reuse. The possibility of an interaction object to support reuse of components is a major contribution to simplify the user interfaces design.

Interactor and (OOP) similarities

There is a link between the general concept of an object and the *interactor* in the *OOP* (Duke & Harrison 1993; Markopoulos 1997). The *interactor* can be distinguished from a generic object by the introduction of a rendering function ρ. This provides an environment together with a representation of the *interactor* internal state.

(OOP) paradigm	***Interactor***
encapsulation	it encapsulates the state, the events that manipulate the state and the ways by which the state becomes perceivable to the users (the presentation).
polymorphism	not verified in the searched literature.
inheritance	not verified in the searched literature.
identity	each one has its own unique identifier.
state	it has several attributes that define the state.
behaviour	*interactor* behaviour could be split into two distinct parts: (i) external, which influences the modification of the appearance and (ii) internal, which includes receiving, processing and sending data to other *interactors* or application processes (Paternò 1994).

Table 24: Similarities between *interactor* and *OOP* paradigm.

It seems that interactor doesn't support several *OOP* concepts.

Abstract Data View (*ADV*) and (OOP) similarities

The *ADV*/*ADO* approach can be mapped using "standard" *OOP* techniques (Cowan & Lucena 1995; Alencar et al. 2002) (Table 25).

(OOP) paradigm	***abstract data view (ADV)***
encapsulation	separation of conflicts between *ADVs* and *ADOs*.
polymorphism	several *ADVs* can be associated with the same *ADO* in order to provide different functionality controls.
inheritance	new objects structure created from an existing object design.
identity	each one has its own unique identifier.
state	*ADVs* support one or more mappings which allow an *ADV* to query the state of any associated *ADO* and to change the state of the *ADO* through its action interface.
behaviour	the action interface of an *ADV* can be invoked both through input messages and changes in the state of the accompanying *ADO*.

Table 25: Similarities between the *ADV* and *OOP* paradigm.

The authors demonstrated how *ADVs* relate to the *ADO* classes having *vertical consistency* (between the visual object (*ADV*) and its associated ADO) and *horizontal consistency* (consistency among the different *ADVs*), since the design is mapped in the implementation. Another mechanism supported by *ADVs* for combining and reusing specifications is *composition*. The *composition* operators constitute a formally defined set of specification constructors that operate on both *ADVs* and *ADOs* to produce composite *ADVs* or *ADOs* from much simpler components. It is a common technique for reusing functionality in the *OOP* systems. *Composition* is considered to be a "black-box" since the nesting object knows its nested objects only through the nested object public interface.

Virtual Interaction Object and (OOP) similarities

It is important to note that *virtual object* classes should not be viewed as the typical abstract *OOP* classes, where hierarchical levels are built (e.g. one class C is a subclass of another class B and consequently, B is a super-class of C) and only derived classes may be instantiated.

(OOP) paradigm	***virtual interaction object***
encapsulation	it encapsulates specific software programming information for abstract interaction objects.
polymorphism	multiple physical appearances, manipulated via the same abstract object (all may be active at the same time).
inheritance	not supported.
identity	unique identifier within the container instantiation definition.
state	the programmer supplies the code to maintain a consistent mapping state between the virtual object and the particular physical appearance.
behaviour	it is represented by those events which are supported by the interface toolkits.

Table 26: Similarities between *virtual interaction object* and *OOP* paradigm.

Virtual interactions in the *I-GET* (user interface programming language) are separate programming objects, which can be created even without a physical mapping. A virtual interaction could have multiple concurrent physical active objects whereas *OOP* classes are *unitary objects*, and an abstract object is a super-class pointer type casting (Savidis 2004b). Savidis & Stephanidis (2006) also used *polymorphism* for creating virtual *interaction objects* and allowed plural objects instantiations from different *application programming interfaces* (*APIs*) which were manipulated by the same abstract object. Originally, *virtual interaction objects* do not support *inheritance* (Table 26) and future work was planned to extend the user interface programming language *I-GET* in order to support the agent classes' *inheritance* (Savidis & Stephanidis 2006).

Complex Component and (OOP) similarities

The new *complex component* concept, introduced in this chapter, is an abstraction, i.e. a simplified description of a system (a user interface visual component, in this case) which captures its essential elements. In *OOP*, the abstractions are mostly represented as classes which support encapsulation, polymorphism and inheritance. Classes are mostly static and define the object capabilities, unlike the object which is a dynamic entity showing identity, state and behaviour. It is then possible to find similarities with the *OOP* paradigm by establishing an association between the *complex component* features and the class definition (Table 27).

(OOP) paradigm	***complex component***
encapsulation	modularity and information hiding.
polymorphism	it allows changes in the visual presentation, keeping its functionality.
inheritance	reusing features and events and adding new ones.
identity	each one has its own unique identifier.
state	a visual appearance at any given moment in response to any event exerted by the user or the application.
behaviour	it is represented by events on the *complex component* dialog.

Table 27: Similarities between *complex component* and *OOP* paradigm.

The new complex *component concept* supports component's encapsulation. This *encapsulation* feature is very important, since a *complex component* contains information at three levels (*visual*, *topological* and *dialog*). *Dialog* characterization is the most difficult to achieve, due to the *complex component* interactive nature. When a *complex component* is created (topologically above other(s) that already exist(s)) its features and behaviour (events) are inherited, being possible to add new features and behaviours. This has the advantage of being possible to reuse its functionality and to increase the *complex components* organization within the user interface (this reuse feature is more detailed in Chapter 6). It has two main advantages:

- *Modularity*: *complex components* can be reused in other interfaces, by allowing the interface designer to do changes in their *visual appearance* and maintaining the *behaviour*. This enables the system to be flexible and to potentiate a better description and organisation of the used components, which contributes to much easier global user interface maintenance.
- *Information hiding*: the possibility to separate the public information from the private information is important when establishing communication rules between

complex components, in relation to either to trigger or to respond to *SE*s, *DEs* and *DA*s. This could also be related to private and public class procedures.

The *complex component* supports also polymorphism and an example occurs when its *visual appearance* is modified but not its functionality. In the case of a *complex component*, distinct *visual appearances* represent different *states*. Every *complex component* has its own identifier, never used by any other component. The dialog via *complex components* involves *SEs* and *DEs*/*DA*s, resulting from the *interaction* between the user and the visual interface.

It is possible to identify several similarities between the *AIOs* studied and the *object-oriented paradigm*. As it was stated before, *complex component* is the most complete *AIO* when referring to visual *presentation* and *interaction* properties and by supporting several *OOP* characteristics. It provides reuse features which increases its flexibility in order to create different user interfaces.

4.6 Conclusions

From the study described in this chapter, it was possible to identify characteristics related with visual presentation and interaction that complex components must have (the meaning of interaction is related to the user interaction with components and the interaction between components). The *AIOs* previously analysed (cf. Chapter 2) indicate to have some limitations in its features, in order to use complex components to specify complete and functional visual user interfaces. Thus, a new *complex component* concept was introduced here, by assembling characteristics found in that previous *AIOs*, with new ones introduced in this new component (mainly related with components composition). Thus, a new and most complete *complex component* concept was presented here, in order to be possible to use it to meet the *CFFI* criteria, simplifying the user interface specification process. The latter goal is achieved as a consequence of the higher level of abstraction introduced by the concept, when compared with using *simple components*. As higher becomes the *complex component* abstraction level, simpler becomes the visual user interface representation. Furthermore, through this study two new key concepts were also introduced: the *self event (SE)* directly related with the user interaction with a *complex component*; and the *delegate events*/*actions* (*DE*/*DA*) which is a new communication mechanism between *complex components*. Both concepts allow to specify and to encapsulate the components' behaviour.

The new AIO concept here proposed (*complex component*) is more complete than the other ones studied. It provides the possibility to do visual presentation customization (using graphical primitives, images and text), supports component composition and behaviour. The functional customization concerned with the user interface functionality is done through the (*DE*/*DAs*) events occurring inside the *complex component*. The features supported by this new component concept greatly improve its versatility when compared with a typical *widget*.

Following, a detailed process to abstract *complex components* and assemble them at the highest abstraction level was developed. A new concept was introduced, called *complete complex component*, representing a complete visual object (of a functional user interface), totally independent of any other interface components. Also, several similarities between the AIOs studied and the fully validated and widely used object-oriented paradigm were identified. That contributes to reinforce the validity of these interaction objects.

In order to verify the level of abstraction introduced by the use of *complex components*, in particular comparing with the use of *simple components*, the test bed user interface introduced in Chapter 3 will be used. Next chapter describes that process.

Chapter 5

Abstraction Process for Complex Components

5.1 Introduction

On a previous study (Rodeiro & Teixeira-Faria 2011) an interactive visual user interface prototype was created, and consists of a simple game for younger children, containing visual elements representing sport balls and sport fields. It was possible to create the visual interface from an abstract representation based on DGAIU system (Rodeiro 2001) which uses *simple components* (*SCs*) (cf. Chapter 3). After the game's interface has been implemented it was considered that the implementation effort could be reduced if that system embraced the *CC* concept (Teixeira-Faria & Rodeiro 2011) introduced in Chapter 4.

This chapter details the application of the previously introduced CCs abstraction process in order to create a complete and functional user interface. The abstraction process is used in order to establish groups of *SCs* as *CCs* and also, to group *CCs* into more *CCs* (but with more functionality) increasing the abstraction level and concomitantly simplifying the interface design. The process is detailed by levels, and increasing one level corresponds to increase the interface abstraction. Following, a detailed analysis is presented over the identified entities (components, events, global visual states, and visual transitions).

5.2 Complex Components Approach

The *complex component (CC)* concept was previously introduced in (cf. Chapter 4) and it is possible to indicate, in a simplified way, that a *complex component* formally corresponds to an algebraic representation of visual states set, whose combination enables to reduce original set of states. The Figure 34 represents the simplified version of a state diagram correspondent to the interactive visual user interface prototype described in Chapter 3, and which will be used in the study presented in this chapter.

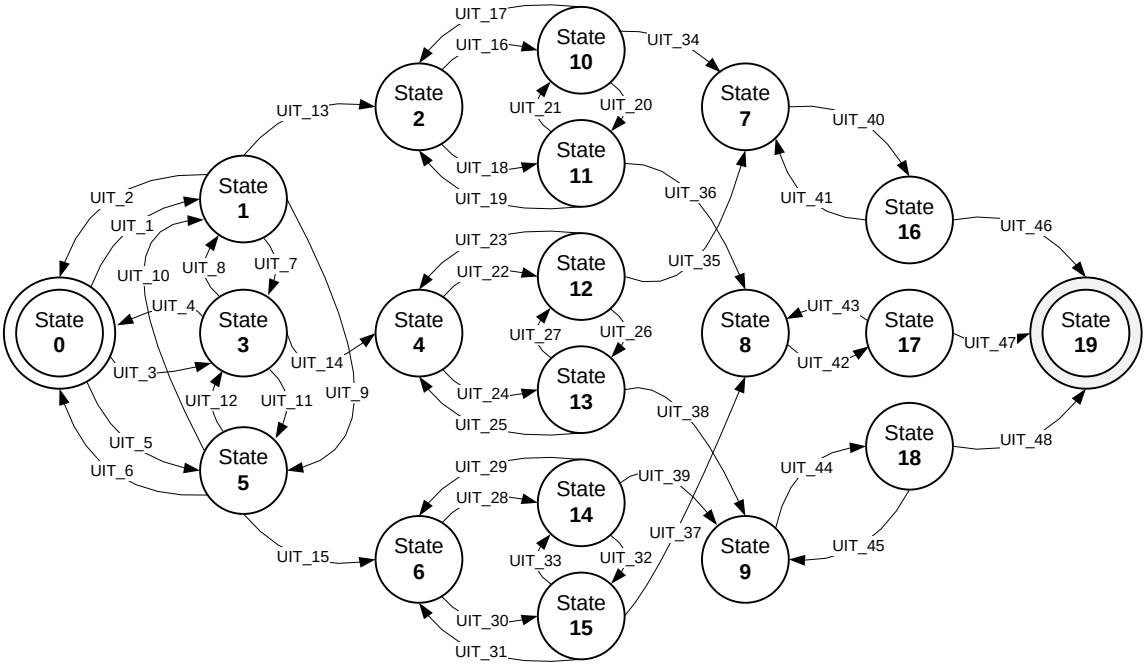

Figure 34: Visual game interface state diagram (simplified version).

A process to abstract *complex components* (introduced in Chapter 4) is used here to verify its applicability in order to obtain *complex components*. The test bed user interface presented in Chapter 3 is used to verify the components assembling process, in order to increase their abstraction level. The abstraction term is here introduced with the meaning of simplification through components encapsulation (more abstraction means fewer components for the same functionality, but each component progressively more complex). The *complex component abstraction process* is following detailed (distributed over 4 levels – *Level 0*, *1*, *2* and *3*) and successive refinement of the process gives different levels of grouping, when considering user events.

5.2.1 Complex Components: Level 0

The approach here presented allows passing from visual elements on the screen to an abstract structure: the *CC*. When considering the basic *CC* concept it is always necessary to establish an entrance level, in which *simple components (SC)* will be defined together with all possible visual transitions between them. Those *simple components* will be responsible for the interface visual aspect. In the case of the game previously described, at this level (*Level 0*), 15 visual *simple components* and 12 possible visual transitions (Figure 35) between them are available. A visual transition is considered to be a visual state change triggered by an event.

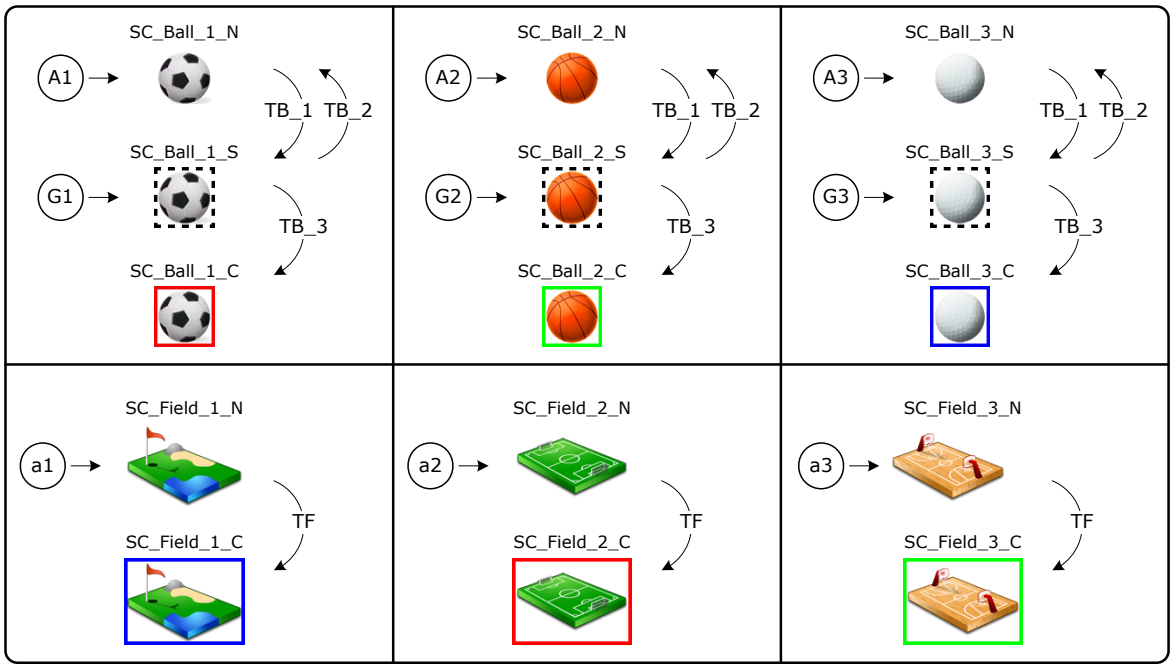

Figure 35: 15 visual states and 12 visual transitions from *simple components*.

The interface designer should know what he intends to have regarding visual and interface functionality, mainly the visual simple components to be used and the possible transitions between them.

Simple Visual Components: Level 0

Throughout this chapter the terms (*normal*, *selected* and *correct*) are used to identify the *CCs* visual states. In this example, each of the three balls can assume the *normal* visual state (*SC_Ball_i_N*, *i* = *{1, 2, 3}*), the *selected* visual state (*SC_Ball_i_S*, *i* = *{1, 2, 3}*) and the *correct* visual state (*SC_Ball_i_C*, *i* = *{1, 2, 3}*). Concerning the sport fields, it can assume (*normal*) visual state (*SC_Field_i_N*, *i* = *{1, 2, 3}*) and (*correct*) visual state (*SC_Field_i_C*, *i* =

{1, 2, 3}). The user events over components are indicated in Figure 35 and identified as: {*(A1)*, *(A2)*, *(A3)*, *(G1)*, *(G2)*, *(G3)*} over the balls and {*(a1)*, *(a2)*, *(a3)*} over the sport fields.

Visual Transitions: Level 0

The game functionality indicates 4 distinct rules concerned with visual state change that may occur between the referred components:

- *TB_1* – change the ball from *normal* to *selected* state
  - SC_Ball_i_N → SC_Ball_i_S, where i = {1, 2, 3}
- *TB_2* – change the ball from *selected* to *normal* state
  - SC_Ball_i_S → SC_Ball_i_N, where i = {1, 2, 3}
- *TB_3* – change the ball from *selected* to *correct* state
  - SC_Ball_i_S → SC_Ball_i_C, where i = {1, 2, 3}
- *TF* – change the sport field from *normal* to *correct* state
  - SC_Field_i_N → SC_Field_i_C, where i = {1, 2, 3}

According with the number of *SCs* and the game rules established, each one of the 4 distinct visual transitions may occur 3 times (3 visual transitions for each ball and 1 visual transition for each sport field) which sums 12 possible visual transitions (Figure 35). These transitions may occur triggered from user events or from the components itself. On next level (*Level 1*) *SCs* will be grouped into *CCs*. Each ball can have 3 different visual representations and will be grouped in one *CC*. Each sport field will be created from 2 different visual representations, which will also be grouped in one *CC*.

5.2.2 Complex Components: Level 1

The *complex component abstraction process* is initiated using *SCs* and considering the interface features to be obtained. This second level corresponds to the *CC* first abstraction level: 6 *CCs* are created. Each ball *CC_Ball_i*, *i = {1, 2, 3}* aggregates 3 *SCs* (correspondent to 3 possible visual states) with their 2 user events and their 3 visual transitions. It can be represented as *CC_Ball_i* = {*SC_Ball_i_N*, *SC_Ball_i_S*, *SC_Ball_i_C*}. The visual states of each *CC* ball (*CC_Ball_i*) are directly concerned with the *SCs* that represent them. For this game interface, each visual state of a *CC* ball is represented by a single *SC* (*SC_Ball_i_k, k = {N, S, C}*). The visual state of a *CC* ball (*CC_Ball_i*) is assumed by some *SC*: (*CC_Ball_i_k*, where *i = {1, 2, 3}* and *k = {N, S, C}*). The other 3 *CCs* are concerned with the sport fields: *CC_Field_i*, *i = {1, 2, 3}*. Each sport field aggregates 2 *SCs*: (*CC_Field_i* = {*SC_Field_i_N*,

SC_Field_i_C}) with its unique user event and its unique visual transition. Analogously to the balls, each *CC* sport field (*CC_Field_i*) has two visual states (*CC_Field_i_k*, where *i = {1, 2, 3}* and *k = {N, C}*).

At this first abstraction level, the 6 *CCs* are mutually independent. Each one has its visual states and accepts events which produce their transitions. Furthermore, initially there isn't any communication between these *CCs*.

Components User Interaction: Level 1

The *CCs* composition is not only topological but also at *dialog* level between components. Thus, the *CCs* have different communication mechanisms and it is possible to distinguish the events involved in the interaction with the visual components. Basically, as previously indicated, there are three event types (Teixeira-Faria & Rodeiro 2011): *self event (SE)*, *delegate event/action (DE/DA)* and *application event (AE)* (if necessary, in the particular case of the game interface presented here, an example of using an *AE* could be a *CC*, visually acting as a time bar and connected with a timeout, whose execution could be controlled by the application).

In *Level 1*, 9 user events were considered, with which user interacts in the game interface. At this present level, *Level 1*, by creating the three *CCs* ball (*CC_Ball_i*, *i = {1, 2, 3}*) each of them reduces by one, the number of user events to be individually considered. Exemplifying for one of the balls, to click (*mouseClick* event adopted) in *CC_Ball_1* through *A'1* user event, corresponds to click in *SC_Ball_1_N* with *A1* user event or to click in *SC_Ball_1_S* with *G1* user event (from previous level).

	User event	*Event type*	*Visual Component*
1	A'1	*mouseClick*	CC_Ball_1
2	A'2	*mouseClick*	CC_Ball_2
3	A'3	*mouseClick*	CC_Ball_3
4	a1	*mouseClick*	CC_Field_1
5	a2	*mouseClick*	CC_Field_2
6	a3	*mouseClick*	CC_Field_3

Table 28: The events triggered by the user over the *CCs* on *Level 1* of abstraction.

In Table 28, 6 user events are indicated (*mouseClick A'1*, *A'2*, *A'3*, *a1*, *a2* and *a3)* which may be triggered over six different *CCs*. Exemplifying for *A'1* event, this represents a mouse event on *CC_Ball_1*, which is responsible for 2 visual transitions (*TB_1* or *TB_2*). These two transitions are triggered by user interaction with *normal* visual state or with *selected* visual state, respectively. In the case of sport fields, for each *CC*, only one visual state may receive user interaction (e.g. to click in *CC_Field_1* is equivalent to click in *CC_Field_1_N* with *a1*

user event). At present level (*Level 1*) the user may trigger 6 user events over 6 *CCs*, reducing in 3 the number of events to be managed.

Visual States and Transitions: Level 1

After 6 user events occurring over 6 *CCs* have been identified, it is important to detail the *SEs* and *DEs*/*DAs* that proceed in result of user interaction with these *CCs*, producing transitions between visual states. Figure 36 represents the components visual transitions that occur in result of these events, possible to be triggered by user or by *CCs*.

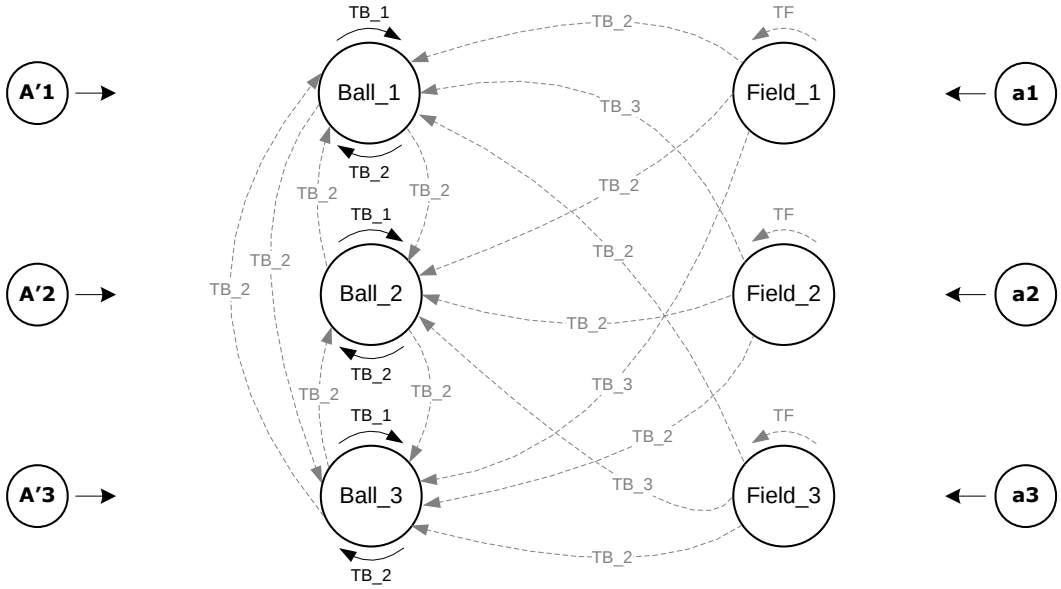

Figure 36: Visual transitions of 6 (*CCs*) on (*Level 1*).

The interface functionality does not become complete, just individually using the 6 *CCs*. It is necessary to establish the missing links between *CCs*. The dashed arrows in gray represent *DEs*/*DAs* necessary to be enabled in order the game interface become fully functional (each arrow corresponds to one *DA* triggered by a *CC* and one *DE* received by another *CC*). Each ball and each sport field may receive a user event (*mouseClick*). In the case of the balls, the *TB_1* or *TB_2* transitions (represented by 6 black continuous arrows) are triggered after user interaction (*A'1*, *A'2* or *A'3*), which produces a *SE* over a ball. For each ball, the *CC_Ball_i_N* visual state may change to the *CC_Ball_i_S* visual state, or the reverse, through *TB_1* or *TB_2* visual transitions, respectively. These transitions are triggered by user interaction *A'i* over a ball visual state (*CC_Ball_i_k*, where *i = {1, 2, 3}* and *k = {N, S}*) and can be represented by the function:

$$A'i = \begin{cases} TB\_1 \ \ if \ k = N \\ TB\_2 \ \ if \ k = S \end{cases} \qquad (1)$$

In the case of the sport fields, any visual transition triggered by user interaction doesn't occur, at this moment. *TF* transitions still inactive because they depend on relations to be established with components, external to the sport fields. Still considering the information provided by Figure 36, is verifiable that besides 6 user events (*A'1*, *A'2*, *A'3*, *a1*, *a2* and *a3*) another 30 events/actions (15 *DEs* and 15 *DAs*) produce (*TB_2* and *TB_3*) transitions (indicated over gray dashed arrows). *TF* transitions are triggered by (*a1*, *a2* and *a3*) user events. Each ball receives 1 *user event* (corresponding to 2 *SEs*), receives 5 *DEs* and triggers 2 *DAs* on other components. In the case of the sport fields, each one receives 1 user event (corresponding to 1 *SE*), receives none *DEs* and triggers 2 *DAs* on other components. These 30 events/actions (15 *DEs* and 15 *DAs*) received/triggered by *CCs*, indicate the need of work to be accomplished in the process of completing the interface functionality.

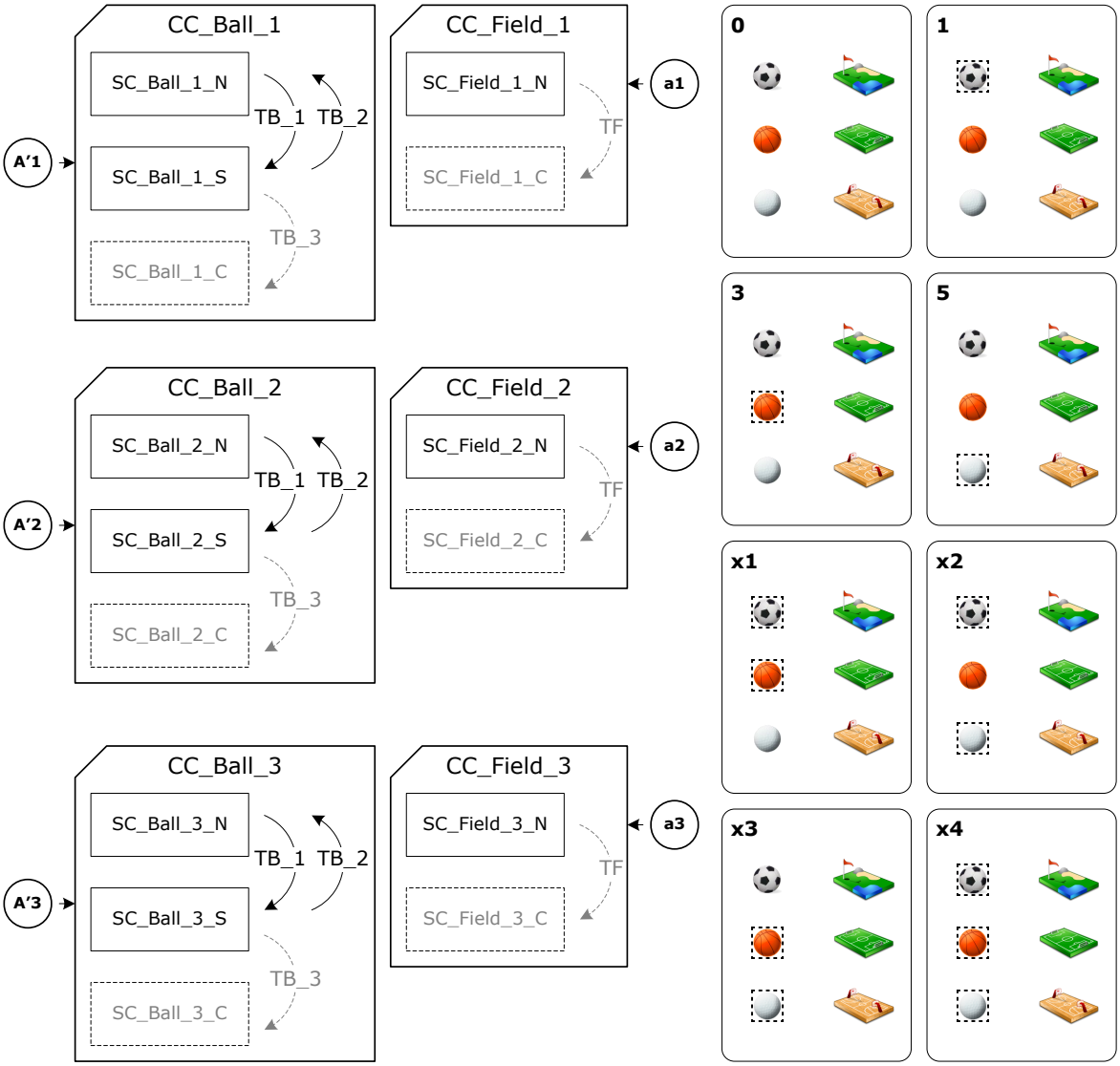

Figure 37: Six *CCs* and 8 global visual states.

Invalid Visual States and Transitions

The combination of all visual states of all components generates all potential global visual states of an interface, together with its potential visual transitions. However, when considering this specific user interface rules, established in order to accomplish the interface objectives, several of those potential states and transitions are not considered. Thus, it can be identified as invalid visual states and transitions for that specific interface purpose. Hence, at this *Level 1*, has not been established yet, a relation between the three balls (only one be selected at a time) indicated by the 6 *TB_2* transitions, represented through the dashed lines in gray (on the left side of Figure 36). As previously referred, each *CC* ball will allow enabling several visual transitions. Two of them (*TB_1* and *TB_2*) occur as direct result of *SEs* triggered by user. Thus, considering the 3 *CCs* balls, 6 distinct visual transitions may occur (2 for each ball). Considering the 3 *CCs* sport fields, none visual transition may occur. Together, all 6 *CCs* may enable 8 interface global visual states. In Figure 37, the possible visual transitions are represented in black on the left side of that figure, and the interface visual states are represented on the right side.

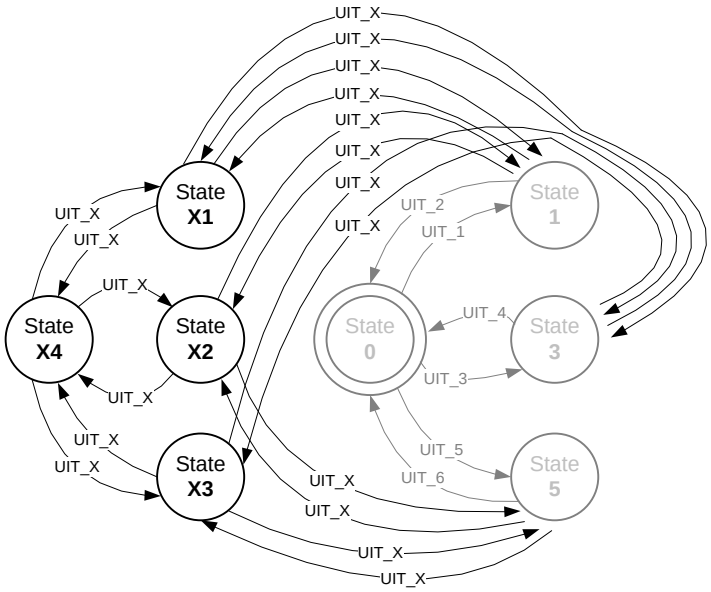

Figure 38: State diagram representing 8 interface visual states and 24 interface game transitions (*Level 1*).

It is important to reinforce that, until now, the analysis has been made only considering visual transitions that occur directly triggered from user interaction, producing *SEs*. Some visual states (*CC_Ball_i_C*, *CC_Field_i_C* where *i = {1, 2, 3}*) and some transitions, (*TB_3* and *TF*) are not yet activated at this abstraction level (represented with gray dashed

lines in Figure 37). Keeping the analysis over the present level and considering the game interface still being incomplete, 8 global interface visual states and 24 transitions between them may exist (Figure 38). However, considering the game context, only 4 interface visual states (*State 0*, *State 1*, *State 3* and *State 5*) and 6 interface visual transitions are valid (*UIT_1*, *UIT_2*, *UIT_3*, *UIT_4*, *UIT_5*, *UIT_6*). Thus, (*State X1*, *State X2*, *State X3* and *State X4*) visual states represented in (Figure 38) are not possible to occur and therefore the related 18 visual transitions represented by (*UIT_X*) will not exist as soon as the game functionality becomes complete.

Completing Visual Interface: Level 1

Until now, the analysis on creating the game interface using the 6 *CCs* (individually) demonstrates the interface functionality is not complete, due to some transitions be only activated by *DEs*, triggered at a higher *CC* abstraction level, or directly by the user interface designer, who has to manually establish the missing links between *CCs*. At this first *CC* level, it was possible also to verify that by using 6 *CCs*, without completing its functionality, some other invalid visual states and transitions may occur (Figure 39).

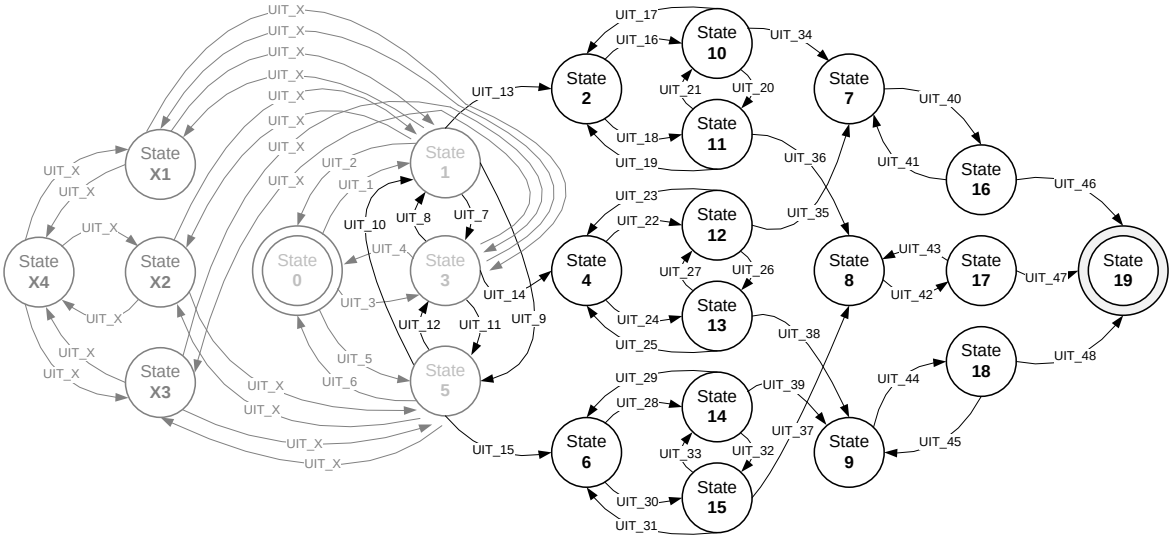

Figure 39: State diagram representing 16 new valid visual states and 42 new valid transitions.

Additionally to the visual states (*State 0*, *State 1*, *State 3* and *State 5*) and transitions (*UIT_1*, *UIT_2*, *UIT_3*, *UIT_4*, *UIT_5* and *UIT_6*) 16 new valid visual states and 42 new valid transitions are indicated (completing the game interface functionality). It is important to remember one of the established game rules that indicates only one ball can be selected at a time. That precondition, per se, will eliminate the 4 (virtual) invalid visual states and 18 (virtual) transitions. Additionally, it will allow creating 6 new valid transitions (*UIT_7*,

UIT_8, *UIT_9*, *UIT_10*, *UIT_11*, *UIT_12*) between the 4 previously existent visual states (*State 0*, *State 1*, *State 3* and *State 5*).

Considering the game rules (in which only one ball can be selected at a time) and the possible user events over a ball, the following function was set: assuming that *A'j* represents a user event (*mouseClick*) over a Ball (*CC_Ball_j, j = {1, 2, 3}*) which can have one of the three possible visible states (*normal (N)*, *selected (S)* or *correct (C)*) identified as *CC_Ball_j_k*, where *k = {N, S, C}* and *j = {1, 2, 3}* a new *BT* function is defined, based on possible visual transitions:

$$BT(j,i) = \begin{cases} TB\_1 \ \ if \ k = N \wedge j = i \\ TB\_2 \ \ if \ k = S \wedge j = i \\ TB\_2 \ if \ k = S \wedge j \neq i \end{cases} \qquad (2)$$

The *BT* function guarantees that just one ball can be selected at a time and allows changing *CC_Ball_i* visual states, according with game rules. When the user selects a ball (*CC_Ball_j, j = {1, 2, 3}*) that means *BT* function will verify that ball visual state, comparing the visual states of all balls (*CC_Ball_i, i = {1, 2, 3}*):

- if the user selects a ball, represented by *CC_Ball_j*, and that ball is in *normal* state, the *BT* function executes a *TB_1* transition, which changes the ball visual state from *normal* to *selected*;
- if the user selects a ball, already in *selected* state, the *BT* function executes a *TB_2* transition;
- concerning the other balls, the *BT* function executes *TB_2* transitions for each ball in its *selected* state.

The *SEs* will be those resulting from user interaction with each ball, causing direct visual changes on them (*BT(1,1)*, *BT(2,2)* and *BT(3,3)*).

The Figure 40 represents the components visual transitions that occur in result of *SEs* and *DEs*, which can be triggered by user or by *CCs*. Some *DEs* (resulting from *DAs*) are here activated. It will be those that cause visual changes in other balls.

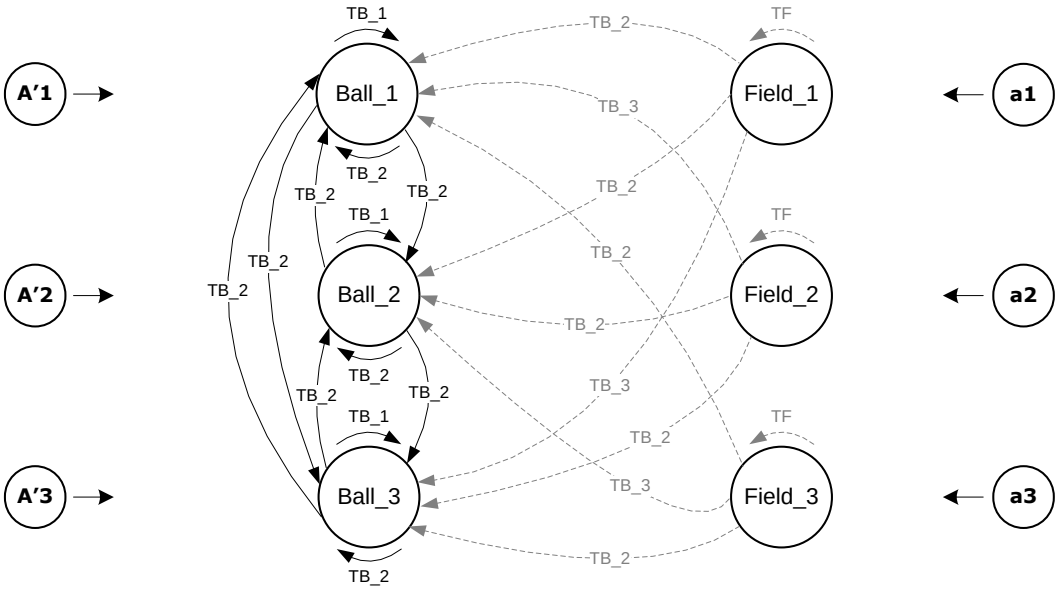

Figure 40: Visual transitions of 6 *CCs*, in *Level 1*, considering the selection balls constraint.

The *DEs*/*DAs* identification concerning the 3 balls are indicated in Table 29. This identification is given by the *BT* function previously established.

	Receives Delegate Events			***Sends Delegate Actions***		
	Transition	***From CC***	***DE id***	***Transition***	***To CC***	***DE id***
CC_Ball_1	TB_2	*CC_Ball_2*	BT (2,1)	TB_2	*CC_Ball_2*	BT (1,2)
	TB_2	*CC_Ball_3*	BT (3,1)	TB_2	*CC_Ball_3*	BT (1,3)
CC_Ball_2	TB_2	*CC_Ball_1*	BT (1,2)	TB_2	*CC_Ball_1*	BT (2,1)
	TB_2	*CC_Ball_3*	BT (3,2)	TB_2	*CC_Ball_3*	BT (2,3)
CC_Ball_3	TB_2	*CC_Ball_1*	BT (1,3)	TB_2	*CC_Ball_1*	BT (3,1)
	TB_2	*CC_Ball_2*	BT (2,3)	TB_2	*CC_Ball_2*	BT (3,2)
total	**6**			**6**		

Table 29: *DEs*/*DAs* identification over each one of the 3 *CCs* balls.

In Figure 41, 4 visual states and 12 game interface global transitions are represented, resulting from the *BT* function acting on the 3 *CCs* (balls). The visual presentation of these 4 states is indicated on the right side of that figure. In Figure 42, 4 global visual states and 12 global visual transitions are represented in gray. However, to complete the functionality of the game interface, 16 global visual states and 36 visual transitions still need to be created. That can be achieved through the establishment of links between the 6 *CCs*. Still considering the information provided by Figure 40, is verifiable that besides visual transitions triggered by *SEs*, another 18 *DEs*/*DAs* produce *TB_2* and *TB_3* transitions, indicated by gray dashed lines. The *TF* transitions are triggered by (*a1*, *a2* and *a3*) user events.

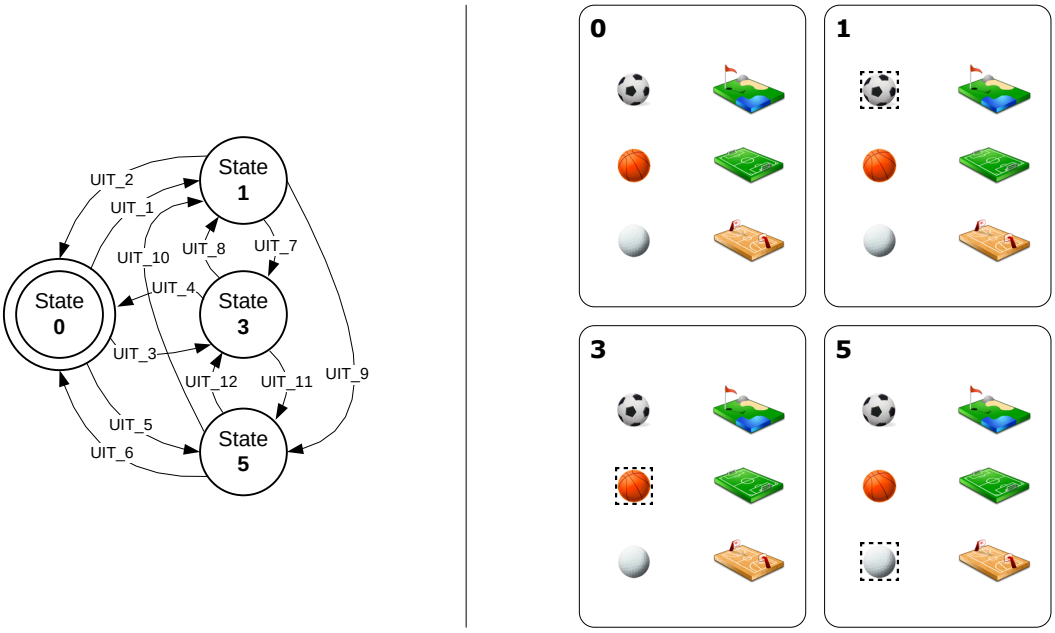

Figure 41: State diagram representing 4 visual states and 12 global interface transitions.

The interface designer needs to establish relationships between the 3 *CCs* (balls) and the other 3 *CCs* (sport fields), in order to ensure that (after a ball has been selected) by clicking in respective sport field, both components (ball and sport field) change to *correct* visual state. Otherwise, in the case that selected ball does not match the correct sport field, it is deselected. Therefore, according with one of the objectives of the game (to do the correct match of the *selected* ball with the *correct* sport field) the following *FT* function was set. Considering that *aj* represents an event (*mouseClick*) over the sport field identified by (*CC_Field_j_N, where j = {1, 2, 3}*):

$$FT\ (j,i) = \begin{cases} TF \wedge TB\_3 \ if\ \text{CC\_Field\_j\_N} \rightarrow \text{CC\_Ball\_i\_S} \\ TB\_2 \qquad if\ \text{CC\_Field\_j\_N} \nrightarrow \text{CC\_Ball\_i\_S} \end{cases} \tag{3}$$

The *FT* function allows changing the interface visual state by activating new *correct* visual states (if the user does a correct choice) or by changing a ball in *selected* visual state, to *normal* visual state (if the user does an incorrect choice). After one ball be on its *selected* visual state, the user may select one of the 3 sport fields (*mouseClick*) (*CC_Field_j, j = {1, 2, 3}*) which has one *normal* visual state identified as (*CC_Field_j_N, where j = {1, 2, 3}*). The *FT* function allows executing the following transitions:

– *TB_3* and *TF* transitions, if the sport field in which the user has clicked corresponds to the correct ball already selected (*CC_Ball_i_S*). That means, if the selected ball corresponds to the chosen sport field, both *CCs* (*CC_Ball_i* and *CC_Field_j*) change to the correct visual state (*CC_Ball_i_C* and *CC_Field_j_C*);

– *TB_2* transition, if there isn't correct correspondence between the selected ball and the chosen sport field, which means the ball in *selected* visual state changes to its *normal* visual state.

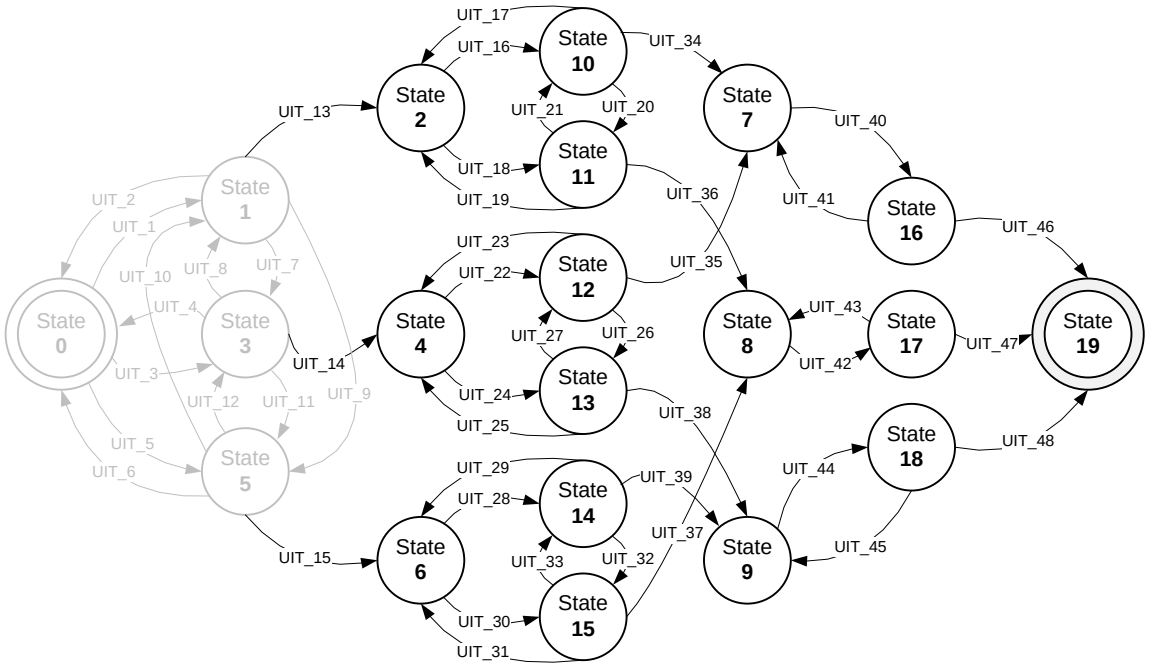

Figure 42: State diagram representing 16 new visual states and 36 new transitions (to complete the UI).

The correct connection will be validated by the *FT* function. The *TF* transition, of *CC_Field_j*, results from a *SE* triggered by user interaction with one sport field. The *TB_3* transition, of *CC_Ball_i*, is not directly concerned with user interaction, but is triggered by a *DE* received from the sport field, correctly chosen by the user. Both (*TF* and *TB_3*) transitions are simultaneously triggered and correspond both to ball and sport field final visual states (the *correct* states). Thus, the interface designer achieves the complete game interface functionality through establishment of missing connections between the 6 *CCs*. The complete connections are represented by the 30 *DEs*/*DAs* (as previously referred) which results from user interaction with the components.

In Table 30, the *DEs*/*DAs* related with the 6 *CCs* are identified and the events, previously created with *BT* function are here enhanced by gray table lines. The other table lines indicate events established by the *FT* function. These last created events are concerned with connecting balls and sport fields, in order to validate the user interface game choices. The number of *DEs* is related to the number of *DAs* (15+15 in this interface example).

	Receives Delegate Events			***Sends Delegate Actions***		
	Transition	***From CC***	***DE id***	***Transition***	***To CC***	***DE id***
CC_Ball_1	TB_2	*CC_Ball_2*	BT (2,1)	TB_2	*CC_Ball_2*	BT (1,2)
	TB_2	*CC_Ball_3*	BT (3,1)	TB_2	*CC_Ball_3*	BT (1,3)
	TB_2	*CC_Field_1*	FT (1,1)			
	TB_3	*CC_Field_2*	FT (2,1)			
	TB_2	*CC_Field_3*	FT (3,1)			
CC_Ball_2	TB_2	*CC_Ball_1*	BT (1,2)	TB_2	*CC_Ball_1*	BT (2,1)
	TB_2	*CC_Ball_3*	BT (3,2)	TB_2	*CC_Ball_3*	BT (2,3)
	TB_2	*CC_Field_1*	FT (1,2)			
	TB_2	*CC_Field_2*	FT (2,2)			
	TB_3	*CC_Field_3*	FT (3,2)			
CC_Ball_3	TB_2	*CC_Ball_1*	BT (1,3)	TB_2	*CC_Ball_1*	BT (3,1)
	TB_2	*CC_Ball_2*	BT (2,3)	TB_2	*CC_Ball_2*	BT (3,2)
	TB_3	*CC_Field_1*	FT (1,3)			
	TB_2	*CC_Field_2*	FT (2,3)			
	TB_2	*CC_Field_3*	FT (3,3)			
CC_Field_1				TB_2	*CC_Ball_1*	FT (1,1)
				TB_2	*CC_Ball_2*	FT (1,2)
				TB_3	*CC_Ball_3*	FT (1,3)
CC_Field_2				TB_2	*CC_Ball_2*	FT (2,2)
				TB_2	*CC_Ball_3*	FT (2,3)
				TB_3	*CC_Ball_1*	FT (2,1)
CC_Field_3				TB_2	*CC_Ball_1*	FT (3,1)
				TB_2	*CC_Ball_3*	FT (3,3)
				TB_3	*CC_Ball_2*	FT (3,2)
total	**15**			**15**		

Table 30: The *DEs*/*DAs* identification over each one of the 6 *CCs*.

The following approach to continue to reduce the complexity of the interface design, could be accomplished by increasing the abstraction (new *CCs* could be created and to act as containers of these already existent 6 *CCs*). According to the *complex component abstraction process* criteria (cf. Chapter 4) and verifying the 6 *CCs*, previously created, have pending *DEs* (which need to be activated, in order to complete the interface) the abstraction level can be increased through the creation of 2 new *CCs*. These components will also behave as 2 containers of the 6 previous components (3 *CCs* distributed by each one). Therefore, it is possible to identify 2 sets of elements with similar characteristics (the set composed by *CC_Ball_1*, *CC_Ball_2* and *CC_Ball_3* and the other set composed by *CC_Field_1*, *CC_Field_2* and *CC_Field_3*). This approach is detailed in the following section, where it will be verified how to achieve the final user interface, using these 2 new *CCs*. Another consideration regarding *CCs* features can be focused in components reuse (to be introduced in Chapter 6). Looking at an individual ball and an individual sport field, both can be reused through some generic ("*ball*" and "*field*") *CCs*. This reuse feature optimizes the design process, due to the fact that only properties concerned with visual presentation

need to be changed (the images used to represent the visual states) and the functionality still the same.

5.2.3 Complex Components: Level 2

At this abstraction level, 2 *CCs* are created (*CC_Balls* and *CC_Fields*). Each one aggregates 3 *CCs* established in previous level (*Level 1*): (*CC_Balls* = {*CC_Ball_i, i = 1, 2, 3*} and *CC_Fields* = {*CC_Field_i, i = 1, 2, 3}*). Each one (*CC_Balls* and *CC_Fields*) acts as a container and represent examples of components with more than one visual state and more than one interaction area (in the case of the balls and the sport fields, each three *CCs* represents three different areas). At this abstraction level, it is considered that the user may interact with 2 *CCs*, representing the balls and the sport fields. The events representing the user interaction over those components are listed in Table 31.

	User event	***Event type***	***Visual Component***
1	A	*mouseClick*	*CC_Balls*
2	a	*mouseClick*	*CC_Fields*

Table 31: Events triggered by the user over the *CCs*, in *Level 2* of abstraction.

Therefore, at this level, a single user event acts on the *CC_Balls* component, which corresponds to click in one of the 3 interaction areas, and whose action may trigger *SEs* or *DEs*. The analogy is identical in the case of *CC_Fields*. By grouping the three balls into a new *CC* (*CC_Balls*) and also doing the same with the 3 sport fields (*CC_Fields*), the user interaction with both *CCs* can be simplified. This occurs by considering 2 user events (*mouseClick*) on these new components (user event (*A*) for the 3 balls and user event (*a*) for the 3 sport fields).

Completing Visual Interface: Level 2

At this level, verifying the way *CCs* are structured, initially 2 new components exist, independent between each other (*CC_Balls* and *CC_Fields*). After these 2 components have been created, the interface designer will verify if he can use them to complete the visual interface. In order the interface functionality becomes complete (to obtain all possible global visual transitions) it becomes necessary to do the connection between the 2 *CCs*, manually establishing the missed links between these 2 components, through their *DEs*/*DAs* which will be responsible for obtaining visual transitions (dashed arrows in gray in Figure 43).

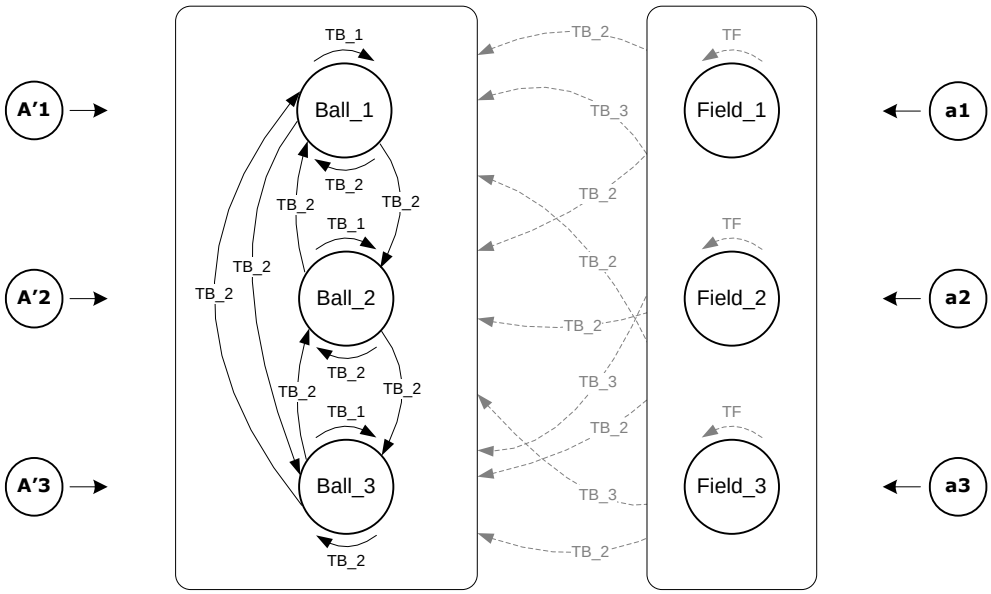

Figure 43: Visual transitions of 2 (*CCs*) on (*Level 2*).

The figure represents the 2 *CCs* visual transitions that occur in result of *SEs* and *DEs*, possible to be triggered by user or by *CCs*. Considering the 2 user events previously identified, representing 9 *SEs* over 6 interaction areas (3 balls and 3 sport fields) of 2 *CCs*, the *SEs* and *DEs* that proceed in result of user interaction with those *CCs*, producing transitions between visual states, will be indicated.

As soon as the *CC_Balls* component is created and 12 *DEs*/*DAs* (internal to it) are enabled, that becomes totally transparent to the interface designer (events concerned with one of the game conditions, which impose that when the user selects a ball, automatically deselect other (if it is selected)). One possible way to complete this game interface is indicated by the rules previously established in *FT* function.

	Receive user events	***Receive self events***	***Receive delegate events***	***Actions enabled in other components***
CC_Balls	3	6	9	0
CC_Fields	3	3	0	9
Total	**6**	**9**	**9**	**9**

Table 32: Number of *SEs* and *DEs*/*DAs* over each one of the 2 *CCs*.

The Table 32 indicates that *CC_Balls* receives 3 user events (corresponding to 6 *SEs*), receives 9 *DEs* and doesn't trigger any action in other components.

	Receives Delegate Events			*Sends Delegate Actions*		
	Transition	***From CC***	***DE id***	***Transition***	***To CC***	***DE id***
CC_Balls	TB_2	*CC_Field_1*	FT (1,1)			
	TB_3	*CC_Field_2*	FT (2,1)			
	TB_2	*CC_Field_3*	FT (3,1)			
	TB_2	*CC_Field_1*	FT (1,2)			
	TB_2	*CC_Field_2*	FT (2,2)			
	TB_3	*CC_Field_3*	FT (3,2)			
	TB_3	*CC_Field_1*	FT (1,3)			
	TB_2	*CC_Field_2*	FT (2,3)			
	TB_2	*CC_Field_3*	FT (3,3)			
CC_Fields				TB_2	*CC_Ball_1*	FT (1,1)
				TB_2	*CC_Ball_2*	FT (1,2)
				TB_3	*CC_Ball_3*	FT (1,3)
				TB_2	*CC_Ball_2*	FT (2,2)
				TB_2	*CC_Ball_3*	FT (2,3)
				TB_3	*CC_Ball_1*	FT (2,1)
				TB_2	*CC_Ball_1*	FT (3,1)
				TB_2	*CC_Ball_3*	FT (3,3)
				TB_3	*CC_Ball_2*	FT (3,2)
total (news)	**9**			**9**		

Table 33: The *DEs*/*DAs* identification over each one of the 2 *CCs*.

In the case of the sport fields (*CC_Fields*) it receives 3 user events (corresponding to 3 *SEs*), doesn't receive any *DEs* and triggers 9 *DAs* on *CC_Balls* component. Using the 2 *CCs* individually on the interface, that allows interface designer to have 4 visual states (*State 0*, *State 1*, *State 3* and *State 5*) and 12 possible visual transitions (*UIT_1*, *UIT_2*, *UIT_3*, *UIT_4*, *UIT_5*, *UIT_6*, *UIT_7*, *UIT_8*, *UIT_9*, *UIT_10*, *UIT_11* and *UIT_12*) indicated in gray in Figure 42. As previously referred, to complete the game interface functionality, 16 new valid visual states and 36 new valid transitions are necessary to be obtained. That can be done through the establishment of missing links (using the *DEs*/*DAs*) between these 2 *CCs*. The relation between *DAs* (triggered by *CC_Fields*) and *DEs* (received by *CC_Balls*) is detailed in (Table 33). These 18 (9 events/9 actions) are those necessary to establish the relation between balls and sport fields, and to complete the game interface functionality, using 2 *CCs*, in this higher abstraction level.

The following section describes a new abstraction level (*Level 3*) by grouping the 2 *CCs* and which will correspond to the final user interface.

5.2.4 Complex Components: Level 3

A structure composed of 8 *CCs* (Figure 44) can be obtained when the *CC* concept is extended to the entire game interface.

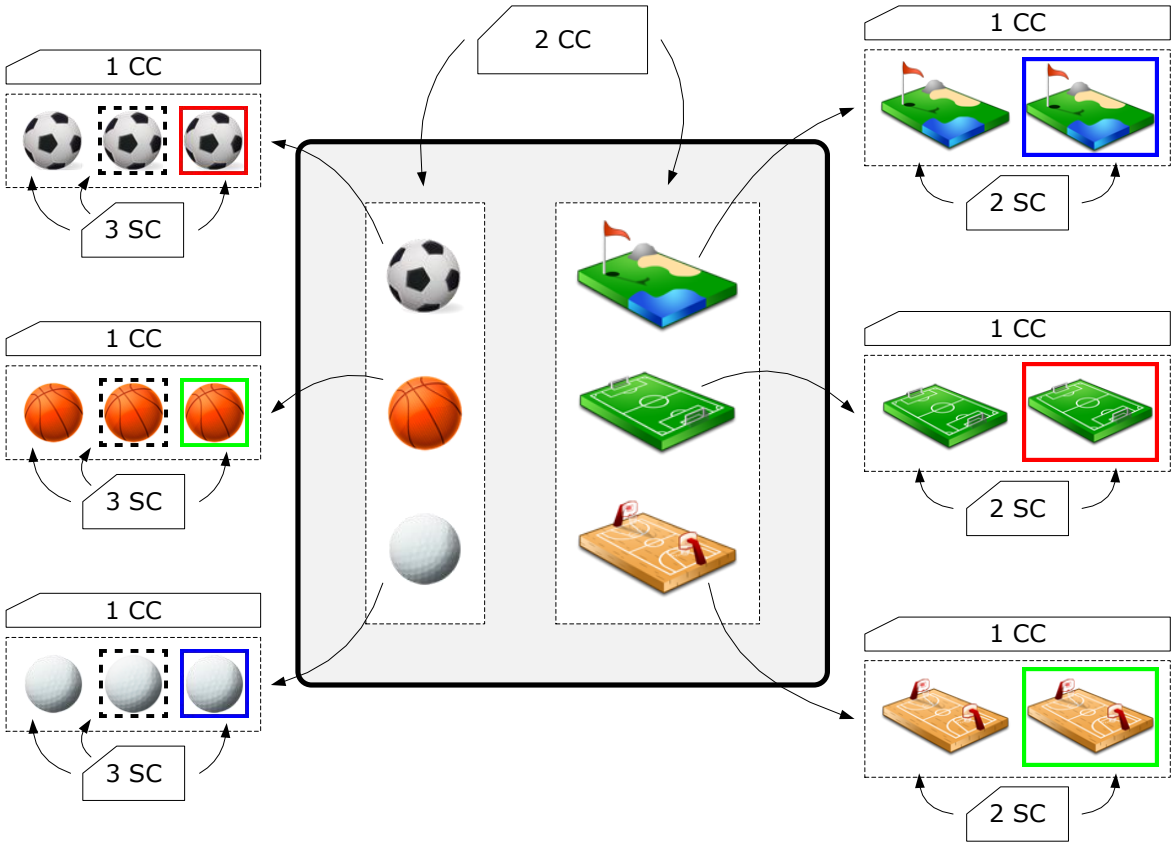

Figure 44: Game structure considering the usage of 8 *CCs*.

The interface components structure previously detailed, at the highest hierarchy level, has two components (one containing 3 balls (*CC_Balls*) and the other containing 3 sport fields (*CC_Fields*)).

Visual Interface Representation: Level 3

Interaction may occur between *SC*/*CC* themselves and between them and the *final interface*. This responsibility for union between components at the highest topological level of the interface cannot be represented through a *CC*. This happens because the final interface cannot fully respect the *Criterion 3* concerned with the *CC abstraction process* (cf. Chapter 4): *"with respect to new complex component creation conditions, if the interface is not fully functional and some of the complex components still have input/output (DE/DA) or other conditions not totally satisfied (e.g. still waiting for some preconditions to be accomplished), the abstraction level can be increased through grouping components."*. Thus, the *final interface* is obtained through a *complete complex component* (*CCC*) which represents a complete visual object composed of related visual components, absolutely independent of any other interface components.

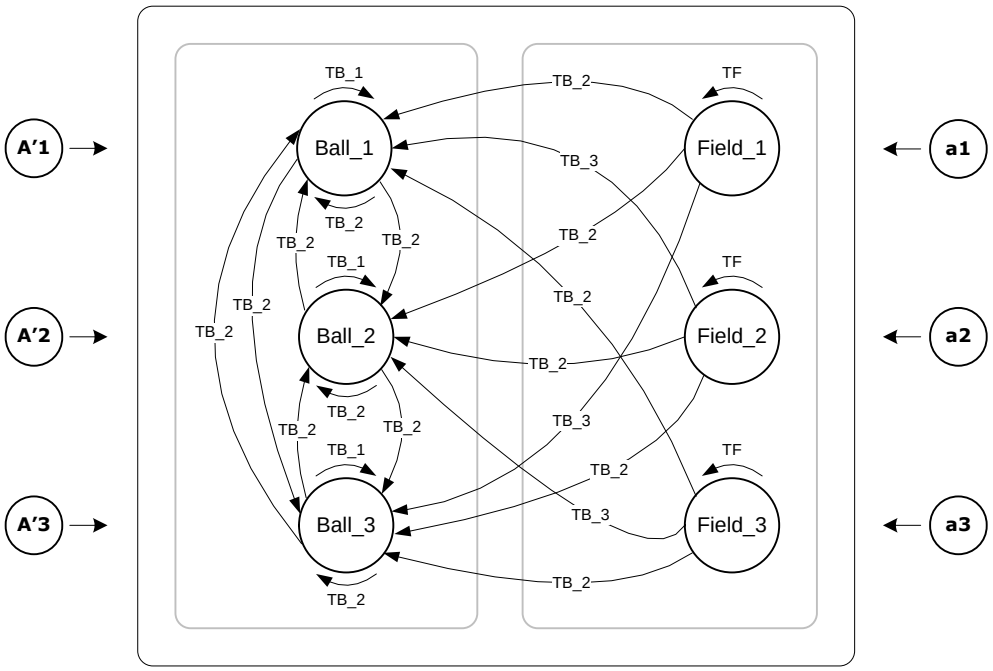

Figure 45: The complete game interface components functionality.

All necessary visual transitions to complete the game interface functionality are depicted on Figure 45. The union between the 2 *CCs* (through the missing 18 *DEs*/*DAs* indicated in previous level) allows obtaining the *CCC*. At this abstraction level (*Level 3*) the final user interface can be referred through this *CCC*, responsible for aggregating 2 *CCs* (*CCC_UI* = {*CC_Balls*, *CC_Fields*}). That ensures not having more *DEs*/*DAs* pending to be connected with some *CC* belonging to this last abstraction level. The interaction between components is fully established and the user may interact with these *CCs*, through events over 6 interaction areas (3 balls and 3 sport fields). At this level, the *CCC* will represent the complete game interface.

5.3 Interface Abstraction Analysis

In this section, a user interface analysis is detailed, considering the level of encapsulation (abstraction) introduced by *CCs* usage to create the game interface previously described. The parameters considered in this analysis are: the events by levels, the global visual transitions and the interface visual states.

5.3.1 Events Identification by Levels

During the previous *CCs abstraction process*, regarding the game interface presented, the occurrence of *SEs* (in result of user events) over those components was verified. It was also noted the existence of *DEs*/*DAs* as a communication mechanism between *CCs*. Some of the *SEs* and *DEs* are triggered simultaneously. Following, the distribution of these events by the *CCs* abstraction levels is detailed.

User Events	*Events Classification*	*Visual Transitions*
A'1	CC_Ball_1 *SE*	*UIT_1*
	CC_Ball_1 *SE*	*UIT_2*
A'2	CC_Ball_2 *SE*	*UIT_3*
	CC_Ball_2 *SE*	*UIT_4*
A'3	CC_Ball_3 *SE*	*UIT_5*
	CC_Ball_3 *SE*	*UIT_6*
a1	Ø	Ø
a2	Ø	Ø
a3	Ø	Ø

Table 34: Events classification by default when creating the 6 *CCs* (*Level 1*).

Considering the *CCs* first abstraction level (*Level 1*) in which 6 *CCs* were created, there are 9 *SEs* resulting from 6 user events over these *CCs*. When the user interacts with each ball (*CC_Ball_i, i = {1, 2, 3}*) he may trigger 2 *SEs* (which change each ball visual state from *normal* to *selected* state and vice-versa). However, when the user interacts with each sport field (*CC_Field_j, j = {1, 2, 3}*) just triggers 1 *SE* (which changes the sport field visual state from *normal* to *correct* state). In order to execute the actions concerning this event it is necessary that one ball being correctly selected. The mere creation of 6 *CCs* automatically provides the achievement of 6 valid global visual transitions (Table 34). Just by the fact of 6 *CCs* have been created, the user events concerning the sport fields don't allow achieving any valid global visual transition. The 42 missing visual transitions (to complete the game interface) have to be manually established by the interface designer.

At abstraction *Level 2*, two *CCs* were created (*CC_Balls* and *CC_Fields*) and there are also 9 *SEs* resulting from 6 user events over these *CCs*. Each one has three interaction areas (3 balls and 3 sport fields). In *CC_Balls*, 12 *SEs* exist and are represented in one of the columns of (Table 35) with light gray colour. Six of these 12 events are inherited from the *CCs* inside *CC_Balls*, which are (*CC_Ball_1*, *CC_Ball_2* and *CC_Ball_3*).

User Events		Events Classification		Visual Transitions
A	A'1	CC_Ball_1 *SE*	CC_Balls *SE*	*UIT_1*
		CC_Ball_1 *SE*	CC_Balls *SE*	*UIT_2*
		CC_Ball_1 *SE* CC_Ball_1 *DE triggered on* CC_Ball_2	CC_Balls *SE*	***UIT_8***
		CC_Ball_1 *SE* CC_Ball_1 *DE triggered on* CC_Ball_3	CC_Balls *SE*	***UIT_10***
	A'2	CC_Ball_2 *SE*	CC_Balls *SE*	*UIT_3*
		CC_Ball_2 *SE*	CC_Balls *SE*	*UIT_4*
		CC_Ball_2 *SE* CC_Ball_2 *DE triggered on* CC_Ball_1	CC_Balls *SE*	***UIT_7***
		CC_Ball_2 *SE* CC_Ball_2 *DE triggered on* CC_Ball_3	CC_Balls *SE*	***UIT_12***
	A'3	CC_Ball_3 *SE*	CC_Balls *SE*	*UIT_5*
		CC_Ball_3 *SE*	CC_Balls *SE*	*UIT_6*
		CC_Ball_3 *SE* CC_Ball_3 *DE triggered on* CC_Ball_1	CC_Balls *SE*	***UIT_9***
		CC_Ball_3 *SE* CC_Ball_3 *DE triggered on* CC_Ball_2	CC_Balls *SE*	***UIT_11***
a	a1	Ø	Ø	Ø
	a2	Ø	Ø	Ø
	a3	Ø	Ø	Ø

Table 35: Events classification by default when creating the 2 *CCs* (*Level 2*).

Each one of the other 6 new events acts simultaneously as *SE* of a ball and *DE* of another ball. Thus, 2 new called *simultaneously SE and DE* are created by each one of the 3 balls. Hence, it was possible to respect one of the game constraints, in which only one ball can be selected at a time. These 12 (*CC_Balls*) *SEs* automatically provide the achievement of 12 valid global visual transitions. Just by the fact of these 2 *CCs* have been created, the user events related with the sport fields still don't allow achieving any valid global visual transition. Using these 2 components, there are still 36 more visual transitions to be obtained by the interface designer, in order to complete the game interface.

User Events		*Events Classification*		*Visual Transitions*
A	A'1	CC_Ball_1 *SE*	CC_Balls *SE*	*UIT_1*
		CC_Ball_1 *SE*	CC_Balls *SE*	*UIT_2*
		CC_Ball_1 *SE* CC_Ball_1 *DE triggered on* CC_Ball_2	CC_Balls *SE*	***UIT_8***
		CC_Ball_1 *SE* CC_Ball_1 *DE triggered on* CC_Ball_3	CC_Balls *SE*	***UIT_10***
		CC_Ball_1 *SE*	CC_Balls *SE*	UIT_22
		CC_Ball_1 *SE*	CC_Balls *SE*	UIT_23
		CC_Ball_1 *SE* CC_Ball_1 *DE triggered on* CC_Ball_3	CC_Balls *SE*	UIT_27
		CC_Ball_1 *SE*	CC_Balls *SE*	UIT_30
		CC_Ball_1 *SE*	CC_Balls *SE*	UIT_31
		CC_Ball_1 *SE* CC_Ball_1 *DE triggered on* CC_Ball_2	CC_Balls *SE*	UIT_32
		CC_Ball_1 *SE*	CC_Balls *SE*	UIT_44
		CC_Ball_1 *SE*	CC_Balls *SE*	UIT_45
	A'2	CC_Ball_2 *SE*	CC_Balls *SE*	*UIT_3*
		CC_Ball_2 *SE*	CC_Balls *SE*	*UIT_4*
		CC_Ball_2 *SE* CC_Ball_2 *DE triggered on* CC_Ball_1	CC_Balls *SE*	***UIT_7***
		CC_Ball_2 *SE* CC_Ball_2 *DE triggered on* CC_Ball_3	CC_Balls *SE*	***UIT_12***
		CC_Ball_2 *SE*	CC_Balls *SE*	UIT_16
		CC_Ball_2 *SE*	CC_Balls *SE*	UIT_17
		CC_Ball_2 *SE* CC_Ball_2 *DE triggered on* CC_Ball_3	CC_Balls *SE*	UIT_21
		CC_Ball_2 *SE*	CC_Balls *SE*	UIT_28
		CC_Ball_2 *SE*	CC_Balls *SE*	UIT_29
		CC_Ball_2 *SE* CC_Ball_2 *DE triggered on* CC_Ball_1	CC_Balls *SE*	UIT_33
		CC_Ball_2 *SE*	CC_Balls *SE*	UIT_42
		CC_Ball_2 *SE*	CC_Balls *SE*	UIT_43
	A'3	CC_Ball_3 *SE*	CC_Balls *SE*	*UIT_5*
		CC_Ball_3 *SE*	CC_Balls *SE*	*UIT_6*
		CC_Ball_3 *SE* CC_Ball_3 *DE triggered on* CC_Ball_1	CC_Balls *SE*	***UIT_9***
		CC_Ball_3 *SE* CC_Ball_3 *DE triggered on* CC_Ball_2	CC_Balls *SE*	***UIT_11***
		CC_Ball_3 *SE*	CC_Balls *SE*	UIT_18
		CC_Ball_3 *SE*	CC_Balls *SE*	UIT_19
		CC_Ball_3 *SE* CC_Ball_3 *DE triggered on* CC_Ball_2	CC_Balls *SE*	UIT_20
		CC_Ball_3 *SE*	CC_Balls *SE*	UIT_24
		CC_Ball_3 *SE*	CC_Balls *SE*	UIT_25
		CC_Ball_3 *SE* CC_Ball_3 *DE triggered on* CC_Ball_1	CC_Balls *SE*	UIT_26
		CC_Ball_3 *SE*	CC_Balls *SE*	UIT_40
		CC_Ball_3 *SE*	CC_Balls *SE*	UIT_41

Table 36: Events classification on *Level 3* (part I).

The third abstraction level (*Level 3*) represents the complete game interface functionality and is represented by the (*CCC_UI*) *complete complex component*. All events

involved on the interface functionality are classified and distributed by 2 tables (Table 36 and Table 37). The first one indicates the user events over the balls and the other indicates the user events over the sport fields.

User Events		*Events Classification*		*Visual Transitions*
a	a1	CC_Field_1 *SE* CC_Field_1 *DE triggered on* CC_Ball_3	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_15
		CC_Field_1 *self event* CC_Field_1 *DE triggered on* CC_Ball_3	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_36
		CC_Field_1 *self event* CC_Field_1 *DE triggered on* CC_Ball_3	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_38
		CC_Field_1 *SE* CC_Field_1 *DE triggered on* CC_Ball_3	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_46
		CC_Field_1 *DE triggered on* CC_Ball_1	CC_Fields *DE triggered on* CC_Balls	UIT_2
		CC_Field_1 *DE triggered on* CC_Ball_1	CC_Fields *DE triggered on* CC_Balls	UIT_23
		CC_Field_1 *DE triggered on* CC_Ball_2	CC_Fields *DE triggered on* CC_Balls	UIT_4
		CC_Field_1 *DE triggered on* CC_Ball_2	CC_Fields *DE triggered on* CC_Balls	UIT_17
	a2	CC_Field_2 *SE* CC_Field_2 *DE triggered on* CC_Ball_1	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_13
		CC_Field_2 *SE* CC_Field_2 *DE triggered on* CC_Ball_1	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_35
		CC_Field_2 *SE* CC_Field_2 *DE triggered on* CC_Ball_1	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_37
		CC_Field_2 *self event* CC_Field_2 *DE triggered on* CC_Ball_1	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_48
		CC_Field_2 *DE triggered on* CC_Ball_2	CC_Fields *DE triggered on* CC_Balls	UIT_4
		CC_Field_2 *DE triggered on* CC_Ball_2	CC_Fields *DE triggered on* CC_Balls	UIT_29
		CC_Field_2 *DE triggered on* CC_Ball_3	CC_Fields *DE triggered on* CC_Balls	UIT_6
		CC_Field_2 *DE triggered on* CC_Ball_3	CC_Fields *DE triggered on* CC_Balls	UIT_25
	a3	CC_Field_3 *SE* CC_Field_3 *DE triggered on* CC_Ball_2	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_14
		CC_Field_3 *SE* CC_Field_3 *DE triggered on* CC_Ball_2	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_34
		CC_Field_3 *SE* CC_Field_3 *DE triggered on* CC_Ball_2	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_39
		CC_Field_3 *SE* CC_Field_3 *DE triggered on* CC_Ball_2	CC_Fields *SE* CC_Fields *DE triggered on* CC_Balls	UIT_47
		CC_Field_3 *DE triggered on* CC_Ball_1	CC_Fields *DE triggered on* CC_Balls	UIT_2
		CC_Field_3 *DE triggered on* CC_Ball_1	CC_Fields *DE triggered on* CC_Balls	UIT_31
		CC_Field_3 *DE triggered on* CC_Ball_3	CC_Fields *DE triggered on* CC_Balls	UIT_6
		CC_Field_3 *DE triggered on* CC_Ball_3	CC_Fields *DE triggered on* CC_Balls	UIT_19

Table 37: Events classification on *Level 3* (part II).

In total, there are 60 *events* that may occur within *CCC_UI*. It is verifiable, from Table 36, the occurrence of 12 events for each ball (36 in total):

– 8 *SEs*; and 4 simultaneously *SE* and *DE*.

Considering the sport fields (Table 37), 24 events may occur (8 events for each sport field) distributed as:

– 4 *DEs*; and 4 simultaneously *SE* and *DE*.

All user events occurring over the sport fields trigger *DEs* on the components (balls) (Table 37), changing a ball to its *normal* or *correct* state, according with the situation of the *selected* ball be, or not be, the correct one. Following, an analysis on global visual transitions is made.

5.3.2 Global Visual Transitions Identification

Continuing to verify the process to abstract *CCs*, it was decided to analyze in detail all global visual transitions that may occur. Specifically, by analyzing Table 38, it is clear in the right column of the table that, from the 48 possible global interface visual transitions, 36 may occur only once:

- 12 refer to the transition to a visual state in which a ball and a sport field pass both to *correct* (*State 13*, *State 14*, *State 15*, *State 34*, *State 35*, *State 36*, *State 37*, *State 38*, *State 39*, *State 46*, *State 47* and *State 48*);
- 24 concerns different possibilities to select a ball (*State 1*, *State 3*, *State 5*, *State 7*, *State 8*, *State 9*, *State 10*, *State 11*, *State 12*, *State 16*, *State 18*, *State 20*, *State 21*, *State 22*, *State 24*, *State 26*, *State 27*, *State 28*, *State 30*, *State 32*, *State 33*, *State 40*, i and *State 44*).

The remaining transitions (12) appear 3 times repeated. These transitions are related with passing a ball from the *normal* to the *selected* state (Table 39). The repetition of these transitions happens due to the fact there are 3 possibilities to go back to the state preceding the *SE* triggered by the user (*mouseClick* on a *selected* ball or in one of the incorrect sport fields):

- by clicking in a *selected* ball (user events of (*A*) type):
  - to trigger 1 *SE* from the 3 possible tasks (*BT (1,1)*, *BT(2,2)* and *BT (3,3)*);
- by clicking in one of the 2 sport fields that do not correspond to the *selected* ball (according with this game rules) (user events of (*a*) type):
  - to trigger one of the 2 *DEs* concerned with each sport field:
    - CC_Ball_1 *selected*: *FT (1,1)* or *FT (3,1)*;
    - CC_Ball_2 *selected*: *FT (1,2)* or *FT (2,2)*;
    - CC_Ball_3 *selected*: *FT (2,3)* or *FT (3,3)*.

User Event (Level 2)	*User Event* (Level 1)	*Tasks from SE*	*Tasks from DE*	*Tasks conditions*	*Tasks*	*Interface Visual Transitions*
A (*CC_Balls*)	A'1 (*CC_Ball_1*)	BT (1,1)		(if CC_Ball_1_N == Visible)	**TB_1** (CC_Ball_1_N = Invisible CC_Ball_1_S = Visible)	UIT_1 UIT_27 UIT_2 UIT_30 UIT_8 UIT_31 UIT_10 UIT_32 UIT_22 UIT_44 UIT_23 UIT_45
				(if CC_Ball_1_S == Visible)	**TB_2** (CC_Ball_1_S = Invisible CC_Ball_1_N = Visible)	
			BT (1,2)	(if CC_Ball_2_S == Visible)	**TB_2** (CC_Ball_2_S = Invisible CC_Ball_2_N = Visible)	
			BT (1,3)	(if CC_Ball_3_S == Visible)	**TB_2** (CC_Ball_3_S = Invisible CC_Ball_3_N = Visible)	
	A'2 (*CC_Ball_2*)	BT (2,2)		(if CC_Ball_2_N == Visible)	**TB_1** (CC_Ball_2_N = Invisible CC_Ball_2_S = Visible)	UIT_3 UIT_21 UIT_4 UIT_28 UIT_7 UIT_29 UIT_12 UIT_33 UIT_16 UIT_42 UIT_17 UIT_43
				(if CC_Ball_2_S == Visible)	**TB_2** (CC_Ball_2_S = Invisible CC_Ball_2_N = Visible)	
			BT (2,1)	(if CC_Ball_1_S == Visible)	**TB_2** (CC_Ball_1_S = Invisible CC_Ball_1_N = Visible)	
			BT (2,3)	(if CC_Ball_3_S == Visible)	**TB_2** (CC_Ball_3_S = Invisible CC_Ball_3_N = Visible)	
	A'3 (*CC_Ball_3*)	BT (3,3)		(if CC_Ball_3_N == Visible)	**TB_1** (CC_Ball_3_N = Invisible CC_Ball_3_S = Visible)	UIT_5 UIT_20 UIT_6 UIT_24 UIT_9 UIT_25 UIT_11 UIT_26 UIT_18 UIT_40 UIT_19 UIT_41
				(if CC_Ball_3_S == Visible)	**TB_2** (CC_Ball_3_S = Invisible CC_Ball_3_N = Visible)	
			BT (3,1)	(if CC_Ball_1_S == Visible)	**TB_2** (CC_Ball_1_S = Invisible CC_Ball_1_N = Visible)	
			BT (3,2)	(if CC_Ball_2_S == Visible)	**TB_2** (CC_Ball_2_S = Invisible CC_Ball_2_N = Visible)	
a (*CC_Fields*)	a1 (*CC_Field_1*)	FT (1,3)	FT (1,3)	(if CC_Field_1_N → CC_Ball_3_S)	**TF** (CC_Field_1_N = Invisible CC_Field_1_C = Visible) **TB_3** (CC_Ball_3_S = Invisible CC_Ball_3_C = Visible)	UIT_15 UIT_38 UIT_36 UIT_46
			FT (1,1)	(if CC_Field_1_N ↛ CC_Ball_3_S)	**TB_2** (CC_Ball_1_S = Invisible CC_Ball_1_N = Visible)	UIT_2 UIT_31 UIT_23 UIT_45
			FT (1,2)	(if CC_Field_1_N ↛ CC_Ball_3_S)	**TB_2** (CC_Ball_2_S = Invisible CC_Ball_2_N = Visible)	UIT_4 UIT_29 UIT_17 UIT_43
	a2 (*CC_Field_2*)	FT (2,1)	FT (2,1)	(if CC_Field_2_N → CC_Ball_1_S)	**TF** (CC_Field_2_N = Invisible CC_Field_2_C = Visible) **TB_3** (CC_Ball_1_S = Invisible CC_Ball_1_C = Visible)	UIT_13 UIT_37 UIT_35 UIT_48
			FT (2,2)	(if CC_Field_2_N ↛ CC_Ball_1_S)	**TB_2** (CC_Ball_2_S = Invisible CC_Ball_2_N = Visible)	UIT_4 UIT_29 UIT_17 UIT_43
			FT (2,3)	(if CC_Field_2_N ↛ CC_Ball_1_S)	**TB_2** (CC_Ball_3_S = Invisible CC_Ball_3_N = Visible)	UIT_6 UIT_25 UIT_19 UIT_41
	a3 (*CC_Field_3*)	FT (3,2)	FT (3,2)	(if CC_Field_3_N → CC_Ball_2_S)	**TF** (CC_Field_3_N = Invisible CC_Field_3_C = Visible) **TB_3** (CC_Ball_2_S = Invisible CC_Ball_2_C = Visible)	UIT_14 UIT_39 UIT_34 UIT_47
			FT (3,1)	(if CC_Field_3_N ↛ CC_Ball_2_S)	**TB_2** (CC_Ball_1_S = Invisible CC_Ball_1_N = Visible)	UIT_2 UIT_31 UIT_23 UIT_45
			FT (3,3)	(if CC_Field_3_N ↛ CC_Ball_2_S)	**TB_2** (CC_Ball_3_S = Invisible CC_Ball_3_N = Visible)	UIT_6 UIT_25 UIT_19 UIT_41

Table 38: The global visual transitions considering *CCs*.

User Event	*User Event (Level 2)*	*User Event (Level 1)*	*Tasks from SE*	*Tasks from DE*	*Visual Transitions*
UI mouseClick	A	A'1	BT (1,1)		
	a	a1		FT (1,1)	UIT_2
	a	a3		FT (3,1)	
	A	A'2	BT (2,2)		
	a	a1		FT (1,2)	UIT_4
	a	a2		FT (2,2)	
	A	A'3	BT (3,3)		
	a	a2		FT (2,3)	UIT_6
	a	a3		FT (3,3)	
	A	A'2	BT (2,2)		
	a	a1		FT (1,2)	UIT_17
	a	a2		FT (2,2)	
	A	A'3	BT (3,3)		
	a	a2		FT (2,3)	UIT_19
	a	a3		FT (3,3)	
	A	A'1	BT (1,1)		
	a	a1		FT (1,1)	UIT_23
	a	a3		FT (3,1)	
	A	A'3	BT (3,3)		
	a	a2		FT (2,3)	UIT_25
	a	a3		FT (3,3)	
	A	A'2	BT (2,2)		
	a	a1		FT (1,2)	UIT_29
	a	a2		FT (2,2)	
	A	A'1	BT (1,1)		
	a	a1		FT (1,1)	UIT_31
	a	a'3		FT (3,1)	
	A	A'3	BT (3,3)		
	a	a2		FT (2,3)	UIT_41
	a	a3		FT (3,3)	
	A	A'2	BT (2,2)		
	a	a1		FT (1,2)	UIT_43
	a	a2		FT (2,2)	
	A	A'1	BT (1,1)		
	a	a1		FT (1,1)	UIT_45
	a	a3		FT (3,1)	

Table 39: The 12 repeated interface visual transitions.

Having until now carried out the identification and classification of global visual transitions that occur in result of events triggered on components, in the following section a global visual states classification is made.

5.3.3 Game Interface Visual States Classification

Depending on the events and the level of components abstraction (encapsulation), the global visual states of the interface have a different classification. Associated with the *SEs* and *DEs* concepts, there exists also the *self visual state* and the *delegate visual state*

concepts. *Self visual state* results from a *SE* on a *CC*, and *delegate visual state* results from a *DE* triggered from other *CC*. Considering the events classification presented in Table 36 and Table 37, 20 game global visual states were classified and are detailed in Table 40. The classification took into account the 6 *CCs* created in *Level 1* (3 dark gray columns) and the 2 *CCs* created in *Level 2* (2 medium gray columns).

Global Visual States	*Self*	*Delegate*	*Self and Delegate*	*Self*	*Delegate*
State 0	×	×		×	×
State 1	×		×	×	
State 2	×	×	×	×	×
State 3	×		×	×	
State 4	×	×	×	×	×
State 5	×		×	×	
State 6	×	×	×	×	×
State 7	×	×	×	×	×
State 8	×	×	×	×	×
State 9	×	×	×	×	×
State 10	×		×	×	
State 11	×		×	×	
State 12	×		×	×	
State 13	×		×	×	
State 14	×		×	×	
State 15	×		×	×	
State 16	×			×	
State 17	×			×	
State 18	×			×	
State 19			×	×	

Table 40: The 20 Interface Visual States classification.

By analyzing Table 40, it is verifiable that visual state indicated as *State 19* (the final state) is the only one that is not considered to be a *self visual state* (when classified considering the *CCs* abstracted at *Level 1*). The visual states (*State 16*, *State 17* and *State 18*) are only classified as *self visual states* (when considering any *CC* abstracted in any of the two levels). These 3 visual states are the three possible visual states, prior to the final state. The other 17 visual states can be classified as *delegate visual states* or *simultaneously self and delegate visual states*, in both levels.

5.3.4 Reducing Complexity with Complex Components

The results obtained from the game interface implementation, based in using *CCs*, will be now presented. The objective is to analyze the interface game representation process using 2 or 6 *complex components*. At *Level 1* above described, and considering the representation of 6 *CCs*, just the internal visual changes between states of each *CC* are

available. These visual changes result from transitions between the *normal*, the *selected* and the *correct* states. It is responsibility of the interface designer to complete the final game representation, establishing the missing links between the 6 *complex components*, through their *DEs*/*DAs*. At this level, the 9 *SEs* concerning the 6 user events that can be triggered over the 6 *CCs* are already represented. It remains to establish the 15 *DEs* and 15 (related) *DAs*, in order the interface functionality become complete. To achieve this, the interface designer may implement the correspondent events indicated through the *BT* and the *FT* functions, which will allow completing the missing functionality.

number of	**level 0**	**level 1**	**level 2**
components	15 (*SC*)	6 (*CC*)	2 (*CC*)
visual states	20	20	20
global visual transitions	48	48	48
user events	9	6	6
self events	×	9	9
simple events	60	×	×
delegate events/actions	×	30	18
total events	**60**	**36**	**24**

Table 41: To compare the use of *SCs* versus the use of *CCs* in the game interface design.

In the second abstraction level, using 2 *CCs*, one of the interface functionality conditions it is already established, which requires that only one ball could be selected at a time. Therefore, the interface designer just needs to ensure proper connection between balls and sport fields. To achieve that, he must establish the representation of 9 *DEs* related to other 9 *DAs*, which is represented through the *FT* function, allowing to complete the interface functionality.

In summary, comparing the game interface design process by using 6 or by using 2 *CCs*, versus using *SCs*, it is verifiable that in both situations the complete functionality of the game is obtained, with 20 visual states and 48 possible transitions being represented. However, at the *Level 0* above described, 60 single events are necessary to be established, while considering *Level 1* or *Level 2* of *CC*s abstraction, only 30 and 18 *DEs*/*DAs*, respectively, need to be established (Table 41). As each level of components abstraction (encapsulation) is made, less events the interface designer has to deal with. Also, a reduction in the number of user events necessary to be controlled is obtained (3 user events less), when comparing between using *SCs*, by one hand and by using 6 or 2 *CCs*, by the other hand. Thus, it is verified a reduction in the number of components to use to design and to represent a user interface functionality being these components more complex (with more functionality) as the abstraction level grows.

5.4 Conclusions

A test bed user interface was specified as a set of *SCs* (cf. Chapter 3) and as a set of *CCs* (as described in this chapter). In both situations, the user understands the interface as a unique entity. Concerning the user interface specification, a simplification is observed when using the proposed *CCs* compared with the use of *SCs*. This simplification can be observed at:

- *Visual presentation*: the process of changing a *SC* is individually done. In the case of a *CC*, the visual state change process is encapsulated and internally done through the *SEs* available in the *CC*;
- *Component composition*: the possibility to group *SCs* into a *CC* and also to group *CCs* into other *CCs* allows to establish their positions with a single operation;
- *Component dialog*: the encapsulation provided by *CCs* allows to group components into other more *CCs*, but with more functionality. When a *CC* is used it will successively activate events, not only on the present *CC*, but also on the *CCs* contained in it. Therefore, there is a proliferation of events inherited from the *CCs* inside of it.

The apparent complexity perceived from the vast number of events involved is actually simplified by using *CCs*. Since the whole interaction process between *CCs* becomes totally transparent to the interface designer, the number of events that the interface designer has to control is reduced. This reduction is obtained as a consequence of the unneeded control of the interface designer over the *DAs* behaviour inside *CCs*. The designer just needs to verify the *SEs* (which arrive to the *CC*) and to identify the *DEs* in other *CCs*.

From the interface abstraction analysis described in this chapter, it was possible to verify that as the user interface abstraction increases through *CCs* encapsulation, more simplified becomes the user interface design, due to: a) reduced number of needed components; and b) simpler components to be used, due to the reduced number of events that the interface designer has to control.

The next chapter addresses the identification and reuse of *CCs* in order to further contribute to the simplification of the design of visual user interfaces.

Chapter 6

Components Identification and Reuse

6.1 Introduction

Following the previous demonstration of a process to abstract *complex components* (*CCs*), this chapter addresses the identification and reuse of *CCs*, in order to further simplify the design of visual user interfaces. The first part of this chapter describes a method to identify *CCs* from a state diagram, representing a complete visual user interface (Rodeiro & Teixeira-Faria 2013a). It is known that an interactive user interface can be described as a set of states and transitions between them (Carr 1994). A user interface state is a concrete situation where a user event is expected. The transitions between user interface states can be represented through an informal description of user actions (as occurs with user interface state diagrams). Thus, from a simple description of a visual interface behaviour (e.g. it can be represented by a set of drawings/visual elements obtained from a hand-made prototype) it is possible to automatically obtain which are the *CCs* that the interface may support and thereby to simplify the visual interface specification process. An original algorithm is described in order to identify *CCs* from a state diagram, representing global visual states and transitions as a result of user interaction. The algorithm is used to simplify the representation of that state diagram by reducing the number of represented states and transitions. The test bed user interface (cf. Chapter 3) is used here, together with some variations of it, which imply to change game rules and consequently the interface functionality, in order to verify three considered hypotheses.

The second part of this chapter is focused on verifying the reuse properties under the object-oriented paradigm, supported by *CCs* usage (Rodeiro & Teixeira-Faria 2013b). As previously referred (cf. Chapter 2) the specification of user interfaces is considered to be one solution for the standardization and interoperability between applications. Aligned with this premise, during the last decade, a large number of user interface description languages have emerged, in particular languages using XML as a support language (XML-UIDL). However, most of those languages are entangled with high-level components provided by toolkits and usually referred as *widgets*. The use of pre-defined *widgets* limits the customization options available, resulting in limitations on their reuse. Despite that XML supports reuse, and also that some of the XML-UIDLs allow visual presentation reuse, those languages were not designed to support functional reuse. Therefore, as *CCs* supports the definition of interface visual elements, its composition and user interaction, a focus on its reuse features is here presented. It is exemplified how the *CCs* can be adapted, and thus reused, when the visual user interface goal changes.

6.2 Identification of Complex Components

In previous chapter an interactive visual user interface prototype was described. Knowing in advance all functioning rules of the previous implemented game interface, it is possible to obtain a visual representation of it, with the transitions between all visual states. The chosen visual representation is the state diagram available in Figure 22 (Chapter 3). Looking at the state diagram that represents the visual game interface, a question emerged: knowing in advance all interface functioning rules, and having a visual representation of it (a state diagram in this case) with the transitions between all visual states, could eventually be possible to reduce the representation complexity concerning the number of states and the number of visual transitions, through the identification and grouping of visual elements which have a common logic? These groups can be characterized as *CCs*. Therefore, it is intended to address the possible issue of *CCs* identification, at a higher abstraction level, which would allow simplifying user interface design. In the literature, it was not found any approach to *CCs* identification from a state diagram representing the visual interface functionality based in free design components. All current approaches to the use of interface *CCs* refer using pre-defined *widgets*. Thus, from the interface state diagram representation, a possibility to establish an automatic, incremental and iterative process of

grouping interface visual elements will be verified, in order to achieve different levels of *CCs* abstraction.

Basically, the system behaviour is reflected in the state diagram and is obtained from an interface definition based in a prototype or from user description. It wasn't used any tool or notation to obtain the state diagram, which can be obtained for example, from paper prototyping. Two characteristics that stand out from the state diagram (Figure 22) are the existence of cyclical changes between states, and the existence of states that, after being achieved, do not have return to a previous state. Hence, the possibility of grouping visual elements with consistent behaviour will be verified. This will be done by introducing the *equivalence class* concept (Devlin 2004) as the basis: given the R equivalence relation in A and $a \in A$, is called *equivalence class* of a through R, to the set of all A elements that satisfy the R relation, which means that $[a] = \{x \in A : xRa\}$. Based in the notion of *equivalence class*, a concept called *equivalent state* was established, which results from grouping of visual elements that hold a consistent behaviour. This behaviour is concerned with the possibility of visual elements can cyclically change its state between each other. In order to create the *equivalent states*, an equivalent relation was necessary to establish, assuming the following:

- Exists a cyclical relationship between visual states;
- And, at least one of these states is a *stable state* (described below).

An example of a cyclical relationship occurs between *State 0* and *State 1* and is provided by grouping *UIT_1* and *UIT_2* transitions. In this example, *UIT_1* transition occurs triggered by only one event over *SC_Ball_1_N* component. In the case of *UIT_2* transition, it may occur triggered by three possible events over three components: *SC_Ball_1_S*, *SC_Field_1_N* and *SC_Field_3_N*. Other cyclical relationships exist and will be identified later in this study. A *stable state* is achieved through user interaction with the visual interface and is *stable* if the only state reachable from it, which has already been visited, is the state itself (including the initial state). In Figure 22, it is possible to identify the following *stable states*: *State 0*, *State 2*, *State 4*, *State 6*, *State 7*, *State 8*, *State 9* and *State 19*. Once some of these states are achieved, in result of some transition occurrence, it doesn't exist any event that triggers a visual transition to a previous visual state. For instance, *State 8* is obtainable through 2 transitions (*UIT_36* and *UIT_37*). Each one is triggered by one possible event, but otherwise, from this state is not possible to return to some of the previous states.

An automatic mechanism was introduced to identify *CCs*, using the *equivalence class* and the *stable state* concepts. This mechanism can be represented through an algorithm, consisting on a sequence of 5 steps (Figure 46).

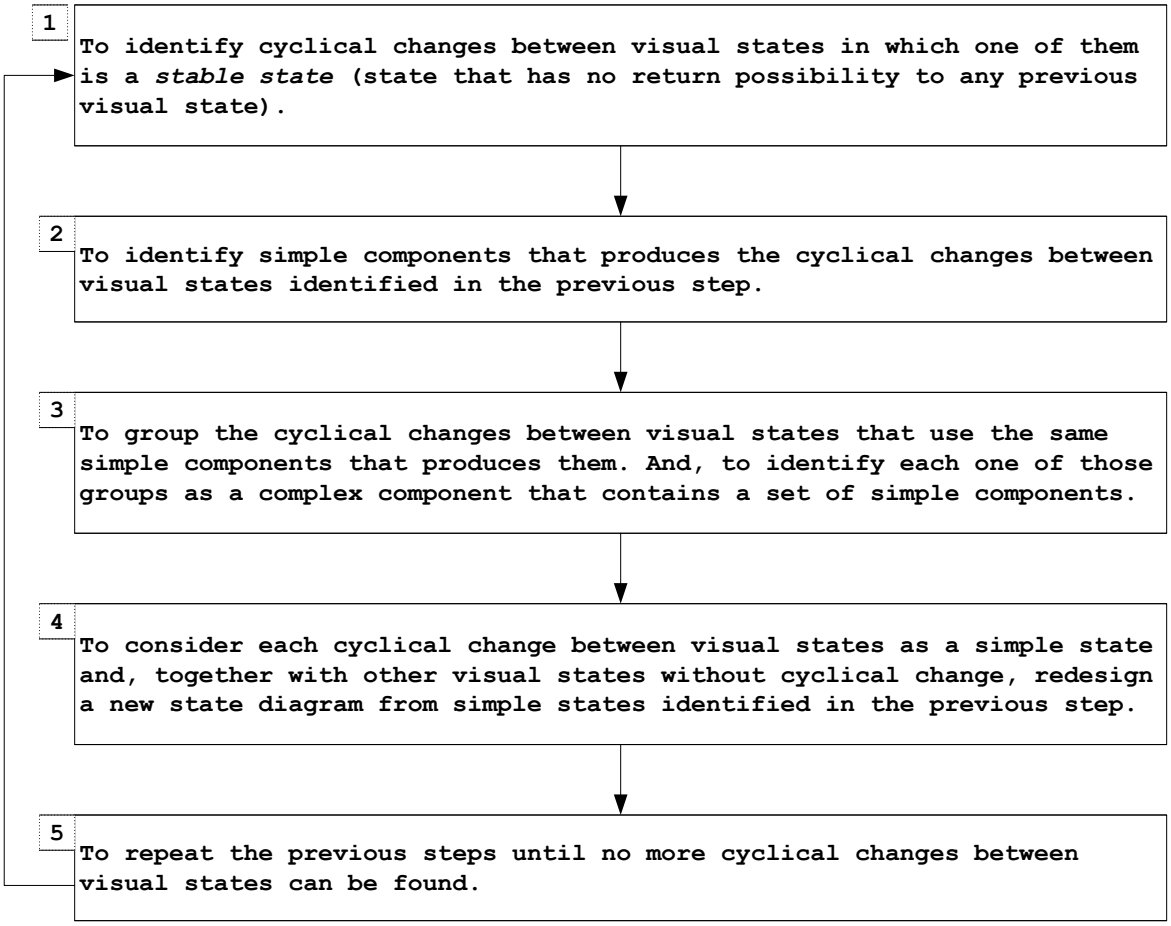

Figure 46: Algorithm used to identify *complex components*.

The algorithm was applied to the state diagram representing the previous visual game interface and its use is detailed in the following section.

6.2.1 Original Visual Game Interface

Figure 47 results from the first algorithm iteration (*step 1* to *step 4*). In the first step, the cyclical changes between visual states are sought. From Figure 22, 18 cyclical changes are identifiable. However, as the objective is to seek those that involve a *stable state*, only 12 of them are considered (identified from *ses0* to *ses11* in Figure 47).

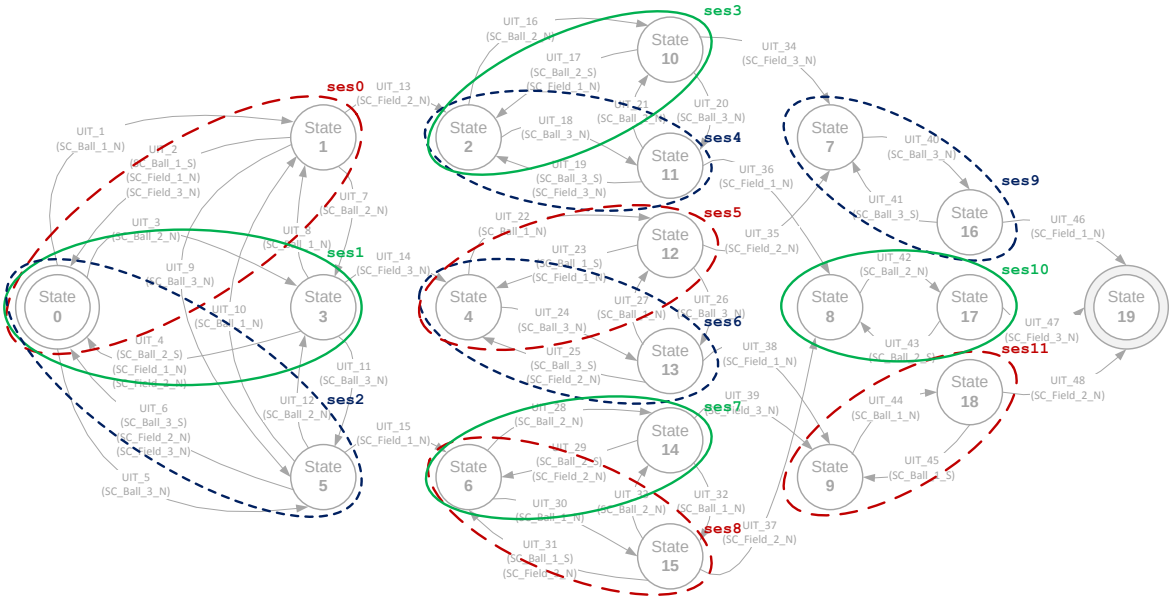

Figure 47: Three sets of cyclical changes identified after first algorithm iteration.

The second step involves identifying components that produce the cyclical changes previously considered (information indicated in the transitions represented by the arrows in the state diagram). On the third step, *CCs* are identified. After the *SCs* involved in the cyclical changes have been identified, those cyclical changes are grouped together as being just one, and identified as one *CC*. For example, the cyclical changes identified as (*ses0*, *ses5*, *ses8* and *ses11*) have in common the identification of two *SCs* (*CC_Ball_1_N* and *CC_Ball_1_S*). Thus, is possible to group these two *SCs* into a single *CC* and to identify it as *CC_Ball_1*. The fourth step of the algorithm considers cyclical changes identified in *step 1* as simple visual states. These will be characterized as *equivalent states* since they hold a consistent behaviour. And, along with the states identified as not participating in those cyclical changes, a new state diagram is redrawn (Figure 48). It must be completed with the visual transitions involving these new states and which do not belong to the group of cyclic changes identified in *step 1*.

The next state diagram is composed by 13 states (*ses0* to *ses11* plus the final state (*State 19*) because is not involved in a cyclical change) (Figure 48). The number of visual transitions to be considered is 27. Still considering the information provided by Figure 47, it should be noted that (*ses*) abbreviation, used to identify the cyclical changes, has the meaning of (*shared and equivalent state*). The *shared* expression is used because, in this first abstraction level, all cyclical changes have one visual state in common (e.g. *State 2* is shared in cyclic changes identified by *ses6* and *ses7*). It is possible to verify, in Figure 47, the

ellipses (identified with three different colours) represent cyclical changes occurring with the balls. Thus, three different components are clearly identifiable (the 3 balls) and classified as *CCs*.

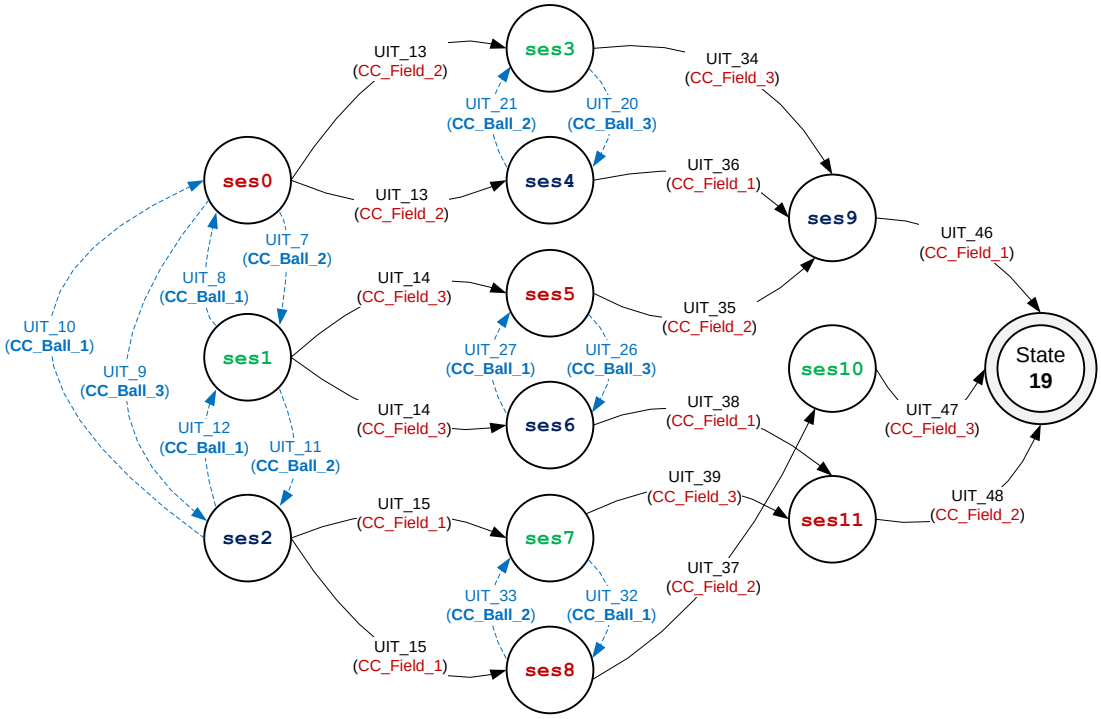

Figure 48: State diagram resulting from first iteration, representing 13 states and 27 visual transitions.

The *step* 5 of the algorithm must be executed if cyclical changes between visual states can be found in the new state diagram created in previous step. It is possible to identify cyclical changes between states in the new state diagram (having 13 states and 27 visual transitions). Thus, a new algorithm iteration must be executed (*step 1* to *step 4*). The execution of this (*step 5*) will increase the abstraction level of the game interface representation.

Second Iteration

A second iteration of the algorithm is detailed. By executing it first step (over the state diagram represented in Figure 48) is possible to verify the existence of 4 cyclical changes (gray dashed lines surrounded by ellipses depicted in Figure 49). Then, the cyclical changes that involve a *stable state* are identified. Because all 13 states are *stable states*, all cyclical changes are considered (identified from *es0* to *es3*).

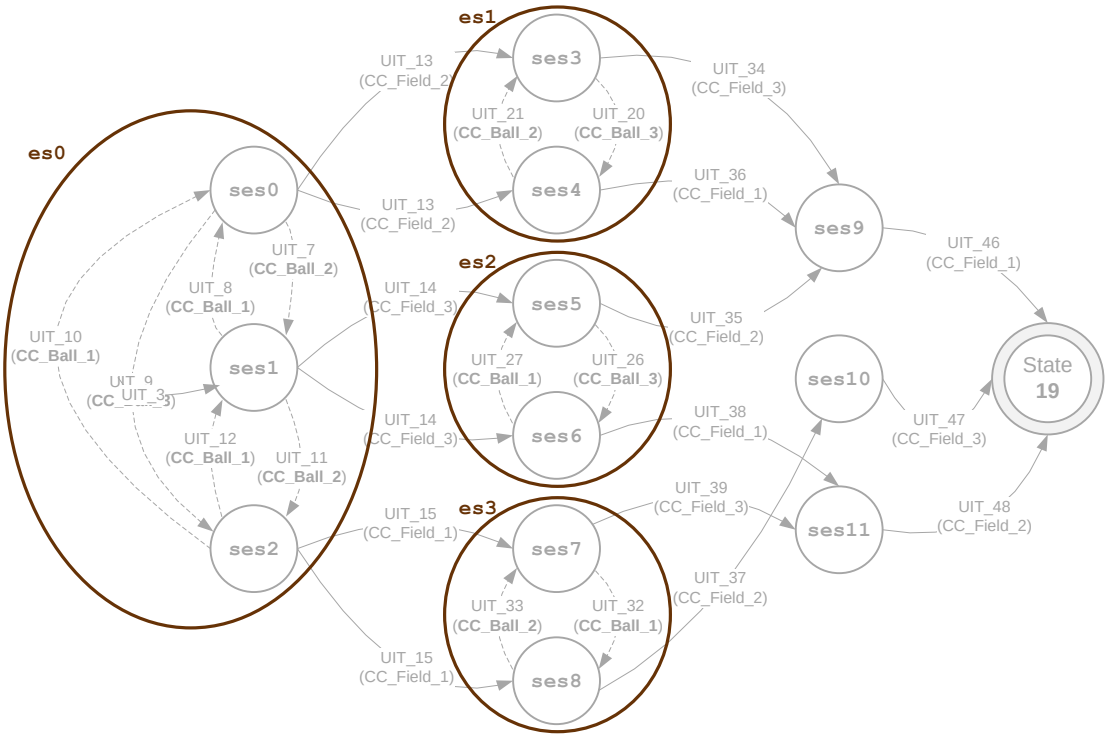

Figure 49: Obtaining 1 *complex component* from the state diagram of the first algorithm iteration.

The second step involves identifying the components (considered as *SCs* in this second iteration) that produce the 4 cyclical changes previously considered (that information is indicated in all transitions). In the third step, other *CC* is identified (in this case, beyond the 3 *CCs* identified in the first algorithm iteration). After the *SCs* involved in cyclical changes have been identified, those cyclical changes involving the same *SCs* are grouped together as being just one, in order to be identified as a *CC*. For example, in Figure 49, the cyclical changes identified (*eso* and the group composed by three other cyclical changes and identified as *es1*, *es2* and *es3*) have in common the identification of three "*simple components*" (*CC_Ball_1*, *CC_Ball_2* and *CC_Ball_3*) (these three components are temporarily considered to be simple, in the second algorithm interaction). As in the first iteration, also in this one is possible to group these three "*simple components*" into a single *CC* that can be identified as *CC_Balls*.

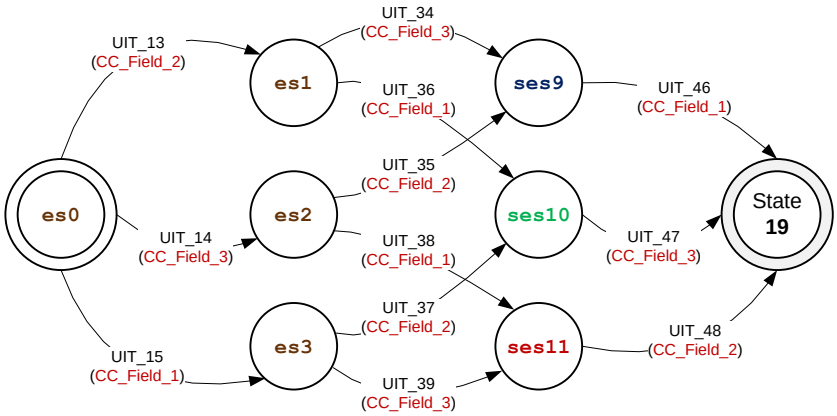

Figure 50: Final state diagram without any more cyclical changes.

The fourth step of the algorithm involves considering the cyclical changes identified in *step 1* as simple visual states. These will also be characterized as *equivalent states* and, along with the states identified as not participating in those cyclical changes, a new state diagram is redrawn. It must be completed with the visual transitions involving these new states and which do not include the group of cyclical changes identified in *step 1*. Exemplifying, by considering Figure 49, it is verifiable that next state diagram is composed by 8 states (*es0* to *es3* plus *ses9*, *ses10*, *ses11* and *State 19*) which are not involved in a cyclical change. The number of visual transitions to be considered is 12 (Figure 50). Still considering the information provided by Figure 49, it should be noted that (*es*) abbreviation, used to identify the cyclical changes, has the meaning of (*equivalent state*). The *shared* expression is not used here, because in this abstraction level none of the cyclical changes have any visual state in common. In Figure 49, it is possible to verify that ellipses represent cyclical changes that occur between the balls. Thus, one component is clearly identifiable (the balls) and can be classified as a *CC*. Figure 50 represents 8 states and 12 interface visual transitions. It is possible to verify no more cyclical changes that can be identified, and thus the algorithm execution ends.

Hence, after the algorithm has been applied to the state diagram representing the original game, 3 *CCs* were identified in the first iteration and, in the following iteration one *CC* has been identified as a container of the 3 previous *CCs*. However, it was not possible to identify the *CCs* concerning the game sport fields. Thus, a question emerges: why this version of the algorithm does not allow identifying all possible *CCs*? Three hypotheses were considered:

– *Hypothesis 1*: the characteristics of the components related with the sport fields do not be identical to the characteristics of the balls;

– *Hypothesis 2*: the no existence of cyclical changes between visual states;
– *Hypothesis 3*: the no existence of a minimum of 3 visual states.

In order to verify these hypotheses, it was decided to introduce several variations to the original test bed user interface.

6.2.2 Visual Game Interface Variations

The algorithm above presented was tested with 3 variations of the original game interface and the results obtained are presented in this section.

Visual Game Interface – Variation A

Considering the original game interface and once the characteristics of the sport fields' components were not identical to the balls' components, the Hypothesis 1 was tested in order to verify if is possible to identify balls and sport fields' *CCs*, by equating the sport fields' behaviour to the balls' behaviour (Figure 51). A game variation was created (Variation A) through the introduction of one more visual state and one more transition.

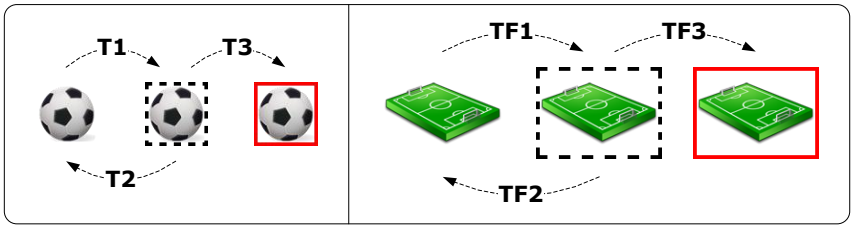

Figure 51: Ball and field states and transitions, in Variation A of the game interface.

Hence, it was necessary to introduce some changes in the original game functionality, establishing the following game rules (in *italic* the new introduced rules):

– The child must associate the balls with the correct sport fields (using a mouse as the physical interaction device);
– *The child starts by selecting a ball or a sport field and then chooses the corresponding sport field/ball;*
– *If one ball or sport field is selected, the child can change the selected one, simply choosing other desired ball/sport field;*
– *If the child does not perform the correct match between one selected ball/sport field and the correspondent sport field/ball then the visual element is deselected;*
– The child cannot select/deselect a ball/sport field that is already correctly matched;
– The game ends when all associations are established.

For each ball and each sport field, three possible visual states have been considered (*normal*, *selected* and *correct*) (Figure 52). Comparing this new version of the game with the original game interface, is noted, concerning sport fields, the inclusion of three *SCs* corresponding to the *selected* visual state. Regarding the topological composition of the visual elements, all of them have been placed at the same level.

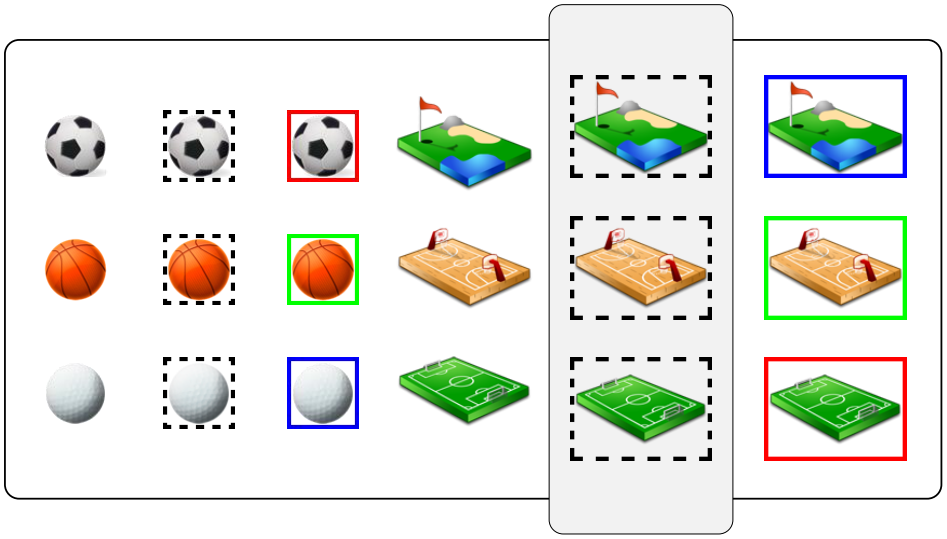

Figure 52: 18 *SCs* used to build the Variation A of the game interface.

It was decided to keep the "*mouseClick*" event detection for user interaction, as it was used in the original game interface. The complete functionality of the reformulated game is represented by 32 visual states (nodes) and 96 transitions (arcs) between those states (Figure 53). Again, in order to simplify the state diagram understanding, it was decided to not include the information concerning the event identification, once the user event is always the same. The initial state is represented by the *State 0* and the final state is represented by the *State 31*. The visual transitions range between *UIT_1* and *UIT_96*, and can be triggered by more than one user event. In Figure 53, each transition has indicated, beyond its identification, the component(s) identification, from which the transition is triggered. All transitions with more than one possible event to trigger them, have indicated with blue underlined text, the second and third (in some cases) components identification, responsible for triggering the transition. In total, this example supports 120 user events, with which the 96 possible visual transitions are triggered.

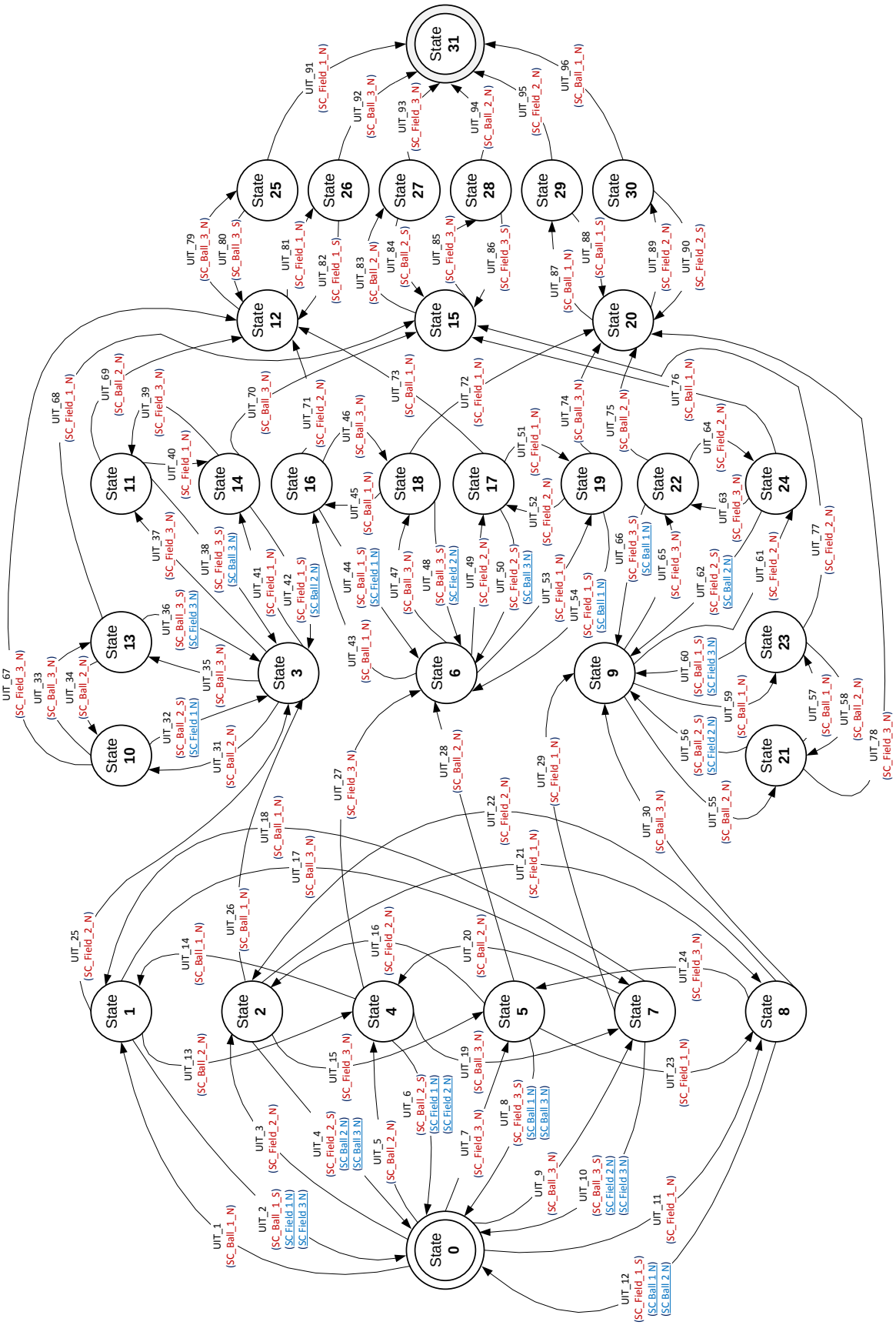

Figure 53: State diagram representing 32 states and 96 visual interface transitions.

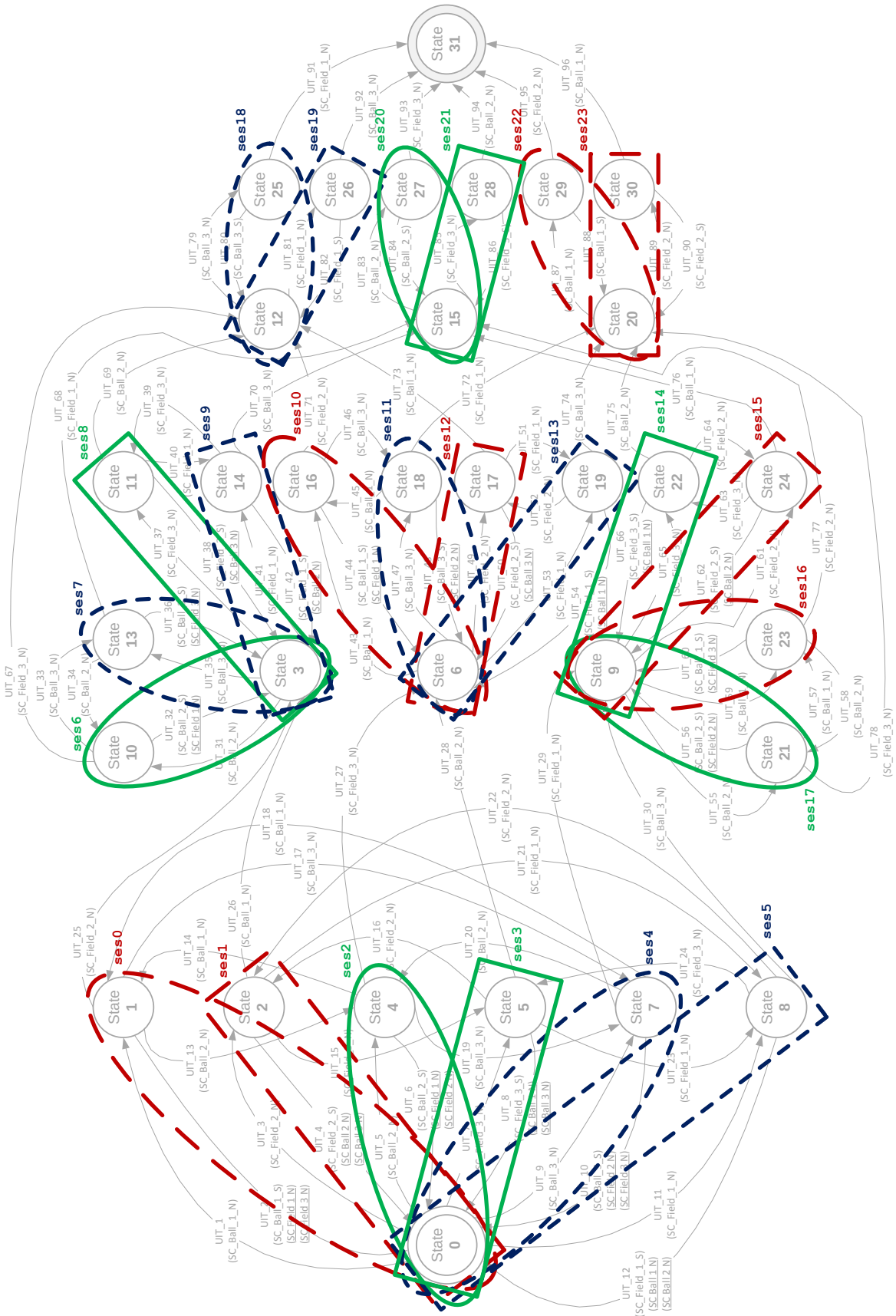

Figure 54: Obtaining 6 *CCs* from the state diagram concerned with the 18 *SCs*.

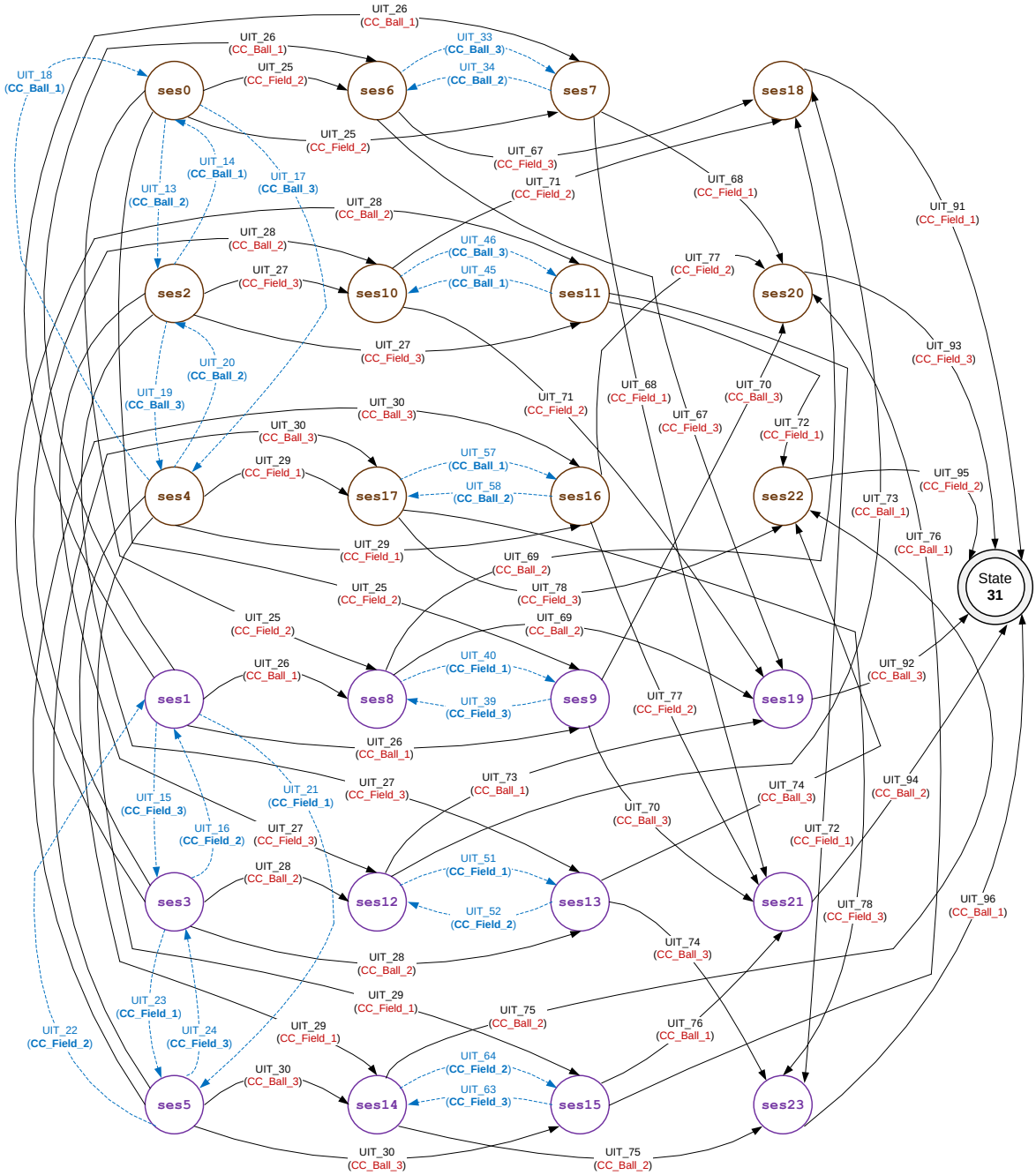

Figure 55: State diagram in result of the first iteration (25 states and 78 visual transitions).

Following, the algorithm was initiated and Figure 54 represents the results obtained from the first 3 steps (*step 1* to *step 3*) of the first iteration. From the set of cyclical changes found, 6 of them, which involve the same *SCs*, can be identified (indicated with three different and coloured ellipses and three different and coloured rectangles). The *SCs* identified and indicated, using the ellipses, are related with the balls visual components, and those indicated with the rectangles refers to the sport fields visual components.

The *step 4* of the algorithm execution allows obtaining the state diagram shown in Figure 55. It represents 25 states and 78 visual interface transitions. The blue dashed lines indicate the remaining cyclical changes between those states, which were not considered on this first iteration. The *step 5* of the algorithm must be executed if cyclical changes between visual states can be found, in the new state diagram created in previous step (which happens in this case). Thus, considering this new state diagram, new algorithm iteration is executed (*step 1* to *step 4*). That will increase the abstraction level of the game interface representation and the obtained results are shown in Figure 56. It results on identifying 8 cyclical changes (gray dashed lines surrounded by solid ellipses and solid rectangles in Figure 56). Since the objective is to find those cyclical changes involving a *stable state* and because all 25 states are *stable states*, all cyclical changes (identified from *es0* to *es7*) are considered.

Other *CCs* were identified (in this case, beyond the 6 identified in the first algorithm iteration). After the *SCs* involved in cyclical changes have been identified, those components are grouped together as being just one, in order to be identified as *CCs*. For example, in Figure 56, cyclical changes identified as (*es0*) and the group composed by three other cyclical changes and identified as (*es2*, *es3* and *es4*) have in common the identification of three "*simple components*" (*CC_Ball_1*, *CC_Ball_2* and *CC_Ball_3*) (as previously indicated, these three components are temporarily considered to be *simple*, in the second algorithm interaction). As occurred with the original game interface, also it is possible to group these three "*simple components*" into a single *CC* that can be identified as *CC_Balls*. Thus, two different components are clearly identifiable (the balls and the sport fields) and can be classified as *CCs*.

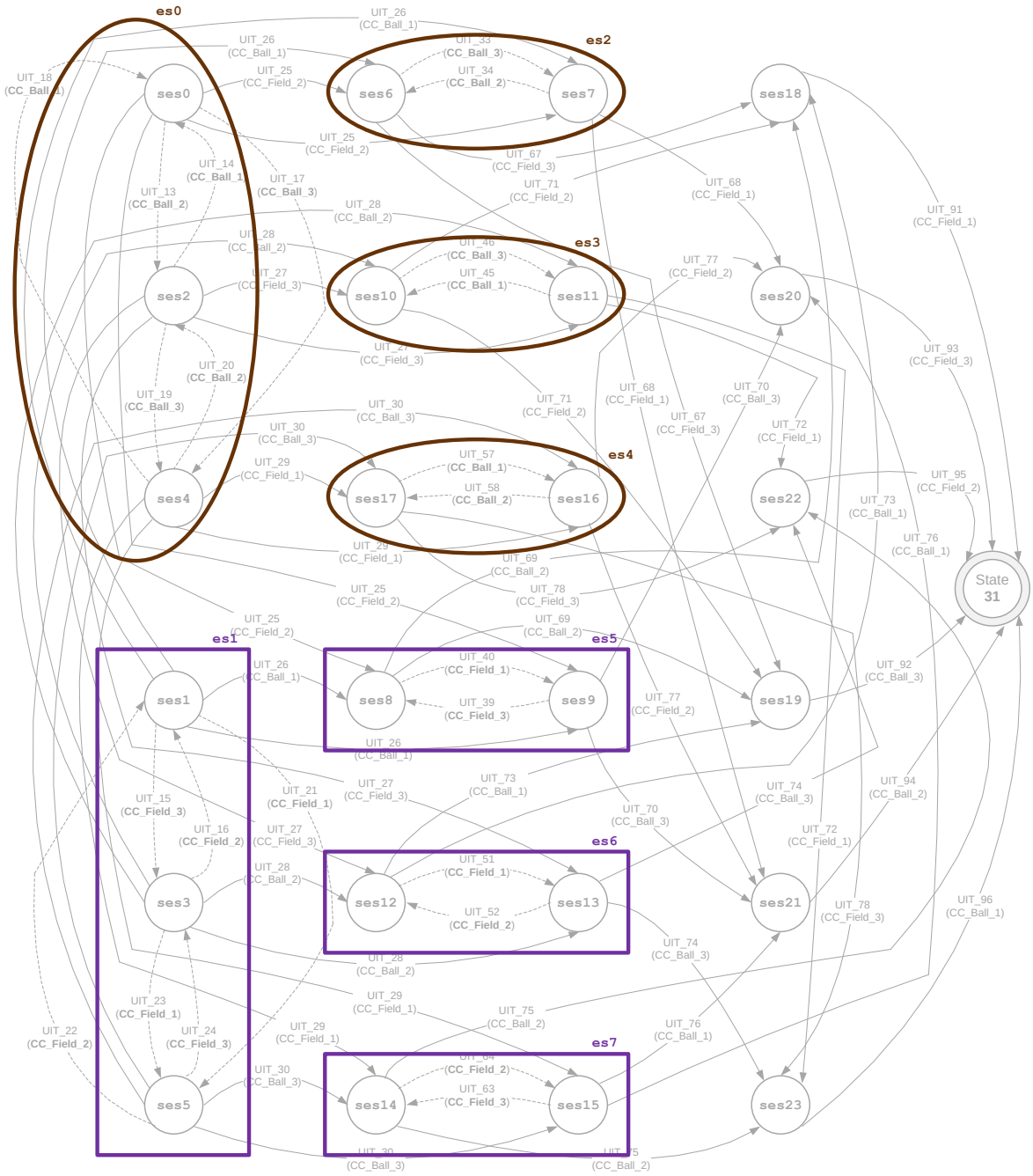

Figure 56: Obtaining 2 *CCs* from the state diagram, in result of the first algorithm iteration.

Visual Game Interface Variation A – Results

After running the algorithm described above it is possible to analyze the obtained results. Initially, the interface is represented by 32 states and 96 visual transitions. Eight of these 32 states are considered to be *stable states*. After the first algorithm iteration, the concept of *equivalence class* is applicable. Through this concept, 24 *equivalent states* are considered, as the result of cyclical changes occurring between the states of the initial representation. Thus, a new state diagram is created, composed by 25 states and 78 visual

transitions. It occurs also an increase in the number of states to be considered *stable states*. After the second iteration, a state diagram consisting of 15 states and 42 visual transitions is obtained. The number of states evolution is presented in Table 42.

	Visual States	*Visual Transitions*	*Equivalent States*	*Stable States*
Initial Representation	32	96	×	8
After 1st iteration	25	78	24	25
After 2nd iteration	15	42	8	15

Table 42: Evolution of the number of states.

An analysis to Table 42 demonstrates a reduction in the number of states and transitions, when comparing the initial interface representation with the 2 algorithm iterations. That allows simplifying the interface functionality representation.

Cyclic changes between visual states	*Simple components which produces cyclic changes*	*Complex Components*
ses0, ses10, ses16, ses22	SC_Ball_1_N, SC_Ball_1_S	CC_Ball_1
ses1, ses12, ses15, ses23	SC_Field_2_N, SC_Field_2_S	CC_Field_2
ses2, ses6, ses17, ses20	SC_Ball_2_N, SC_Ball_2_S	CC_Ball_2
ses3, ses8, ses14, ses21	SC_Field_3_N, SC_Field_3_S	CC_Field_3
ses4, ses7, ses11, ses18	SC_Ball_3_N, SC_Ball_3_S	CC_Ball_3
ses5, ses9, ses13, ses19	SC_Field_1_N, SC_Field_1_S	CC_Field_1

Table 43: Identification of 6 *CCs*.

After running the *step 3* of the first algorithm iteration, it is possible to identify 6 *CCs*, by grouping *SCs* involved in the same cyclical changes. By existing 6 distinct cyclical changes, it is possible to identify 6 *equivalence classes* that correspond to 6 *CCs*. The Table 43 indicates the 6 *CCs* obtained from cyclical changes identification between states. In the table, is also indicated the *SCs* identification responsible for triggering those cyclical changes. Analogously, in *step 3* of the second algorithm iteration, 2 *CCs* are identified (Table 44).

Cyclic changes between visual states	*Simple components which produces cyclic changes*	*Complex Components*
es0, {es2, es3, es4}	CC_Ball_1, CC_Ball_2, CC_Ball_3	CC_Balls
es1, {es5, es6, es7}	CC_Field_1, SC_Field_2, CC_Field_3	CC_Fields

Table 44: Identification of 2 *CCs*.

As previously referred, the *step 5* of the algorithm must be executed if cyclical changes between visual states can be found in the new state diagram created in *step 4*. The results obtained from the second algorithm iteration shows that there are no more iterative cycles (Figure 57).

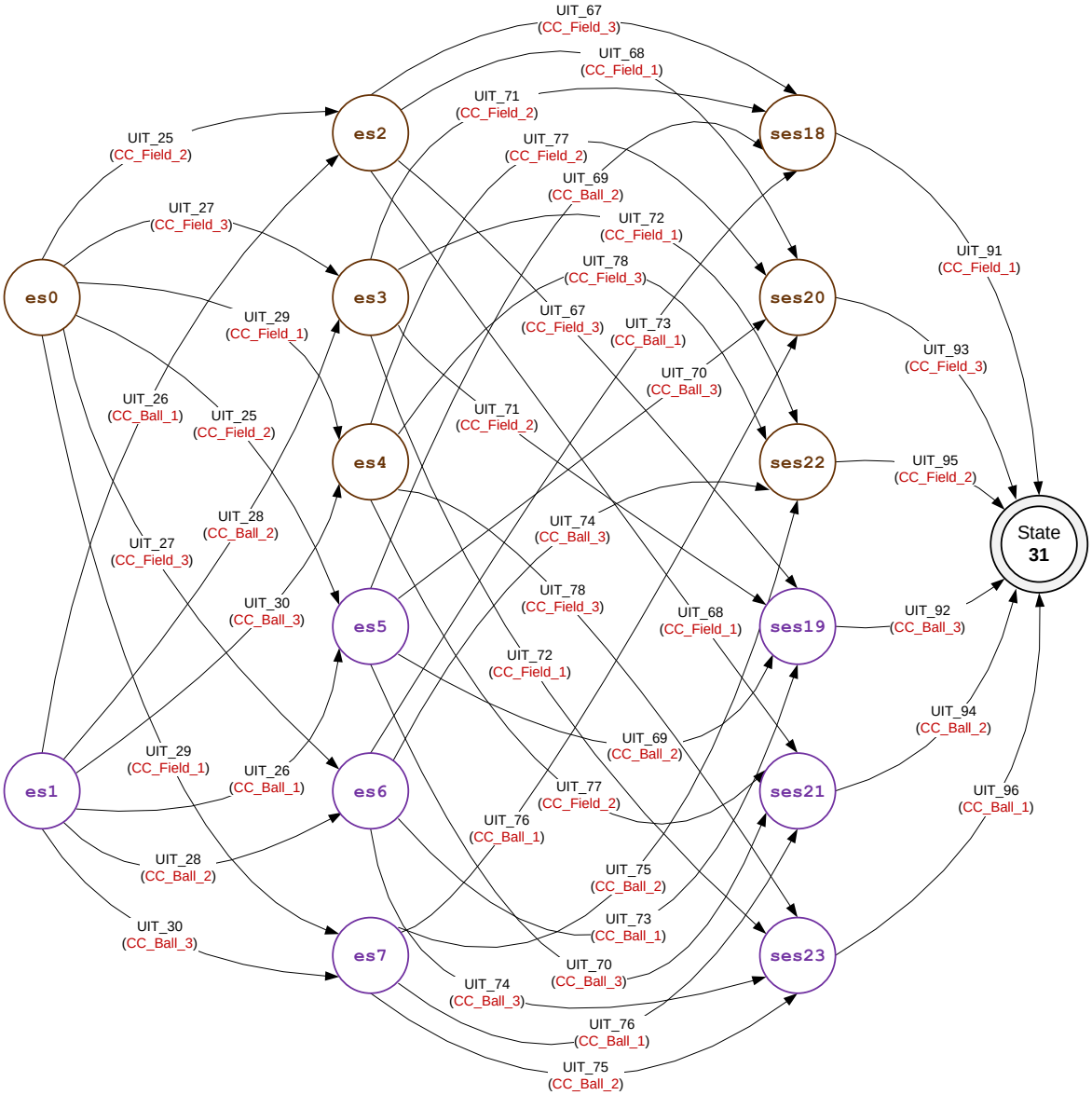

Figure 57: Visual Game Interface – Variation A: final state diagram.

The simple visual components and transitions between them, considered in this study (Visual Game Interface – Variation A), where the fields' behaviour was equated to the balls' behaviour, allowed identifying *CCs* concerned with the balls and the sport fields. Thus, the interface designer may create the same game interface, using 6 *CCs* at a higher abstraction level, when compared with the information provided by the initial *SCs* state diagram representation (Figure 53). Moreover, he can increase the abstraction level and thus only needs to use two *CCs* to design the interface, simplifying even more the interface representation. After analysing and presenting the Visual Game Interface – Variation A, it was verified that by considering the characteristics of the components related with the

sport fields be identical to the characteristics of the balls, the proposed algorithm allows identifying all *CCs* available.

The following section is focused in verifying Hypothesis 2.

Visual Game Interface – Variation B

Still considering the original game interface, the Hypothesis 2 was tested in order to verify: if doesn't exist cyclical changes between visual states is not possible to identify *CCs*. A game variation was created (Variation B) through removing the *T2* transition between the *selected* visual state and the *normal* visual state of each ball (Figure 58).

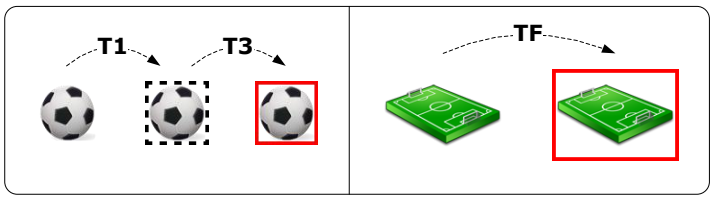

Figure 58: The visual states and transitions in Variation B of the game interface.

Hence, it was necessary to introduce some changes in the original game functionality, establishing the following game rules (in *italic* the new introduced rules):

- The child must associate the balls with the correct sport fields (using a mouse as the physical interaction device);
- The child starts by selecting a ball and then try to match the correct sport field;
- *If one ball is selected the child cannot then change that selection;*
- *If the child does not perform the correct match between one selected ball and the correspondent sport field, the correct visual state is not achieved (in both ball and sport field);*
- The child cannot select/deselect a ball/sport field that is already correctly matched;
- The game ends when all associations are established.

The game functionality is represented by 20 visual states and 24 transitions between those states (Figure 59). It was decided to keep the same visual transitions identification, used in the state diagram representing the original game interface (Figure 31 in Chapter 4). The new example (Figure 59) supports 24 user events, each one triggering the related transition. The algorithm was initiated and at the very first step it is possible to detect the absence of cycles between global visual states. Thus, it becomes impossible to identify any *CCs* in this new game interface variation.

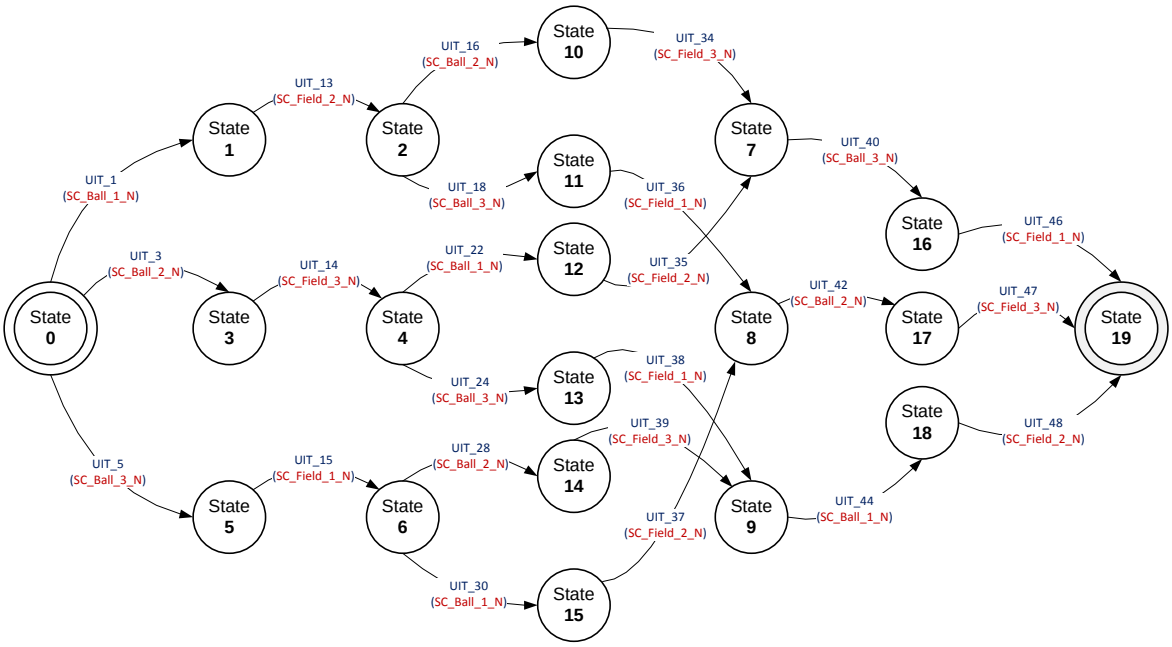

Figure 59: State diagram representing Variation B of the game interface.

After the Hypothesis 2 of the game interface has been tested, another hypothesis was studied, considering the existence of a minimal number of 3 visual states, in order to be possible to identify *CCs*: Hypothesis 3. The next section is focused in verifying this new hypothesis.

Visual Game Interface – Variation C

The Hypothesis 3 to be verified still considers the original game interface. A game variation was created (Variation C). For each ball, it was decided to remove an intermediate visual state (and the related transitions) and to add new transitions. For each sport field, it was decided to add one visual transition (Figure 60). Therefore, each ball will have two visual states (*normal* and *correct*) and two transitions between them (*TB1* and *TB2*) and each sport field will also have (*normal* and *correct*) visual states and two transitions (*TF* and *TF2*).

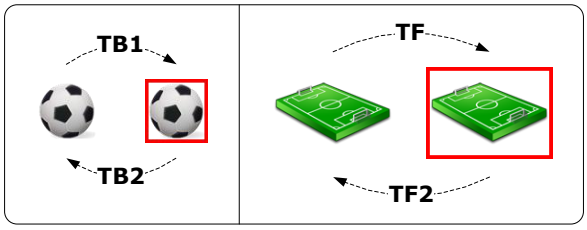

Figure 60: Ball and sport field states and transitions, in Variation C of the game interface.

Hence, it was necessary to introduce some changes in the original game functionality, establishing the following game rules (in *italic* the new introduced rules):

- The child must associate the balls with the correct sport fields (using a mouse as the physical interaction device);
- *The child starts by selecting a ball or a sport field and then chooses the corresponding sport field/ball (in this case the selection is visually represented by the second visual state (correct));*
- *If one ball or sport field is selected, the child can change the selected one, simply choosing other desired ball/sport field;*
- *If the child does not perform the correct match between one selected ball/sport field and the correspondent sport field/ball then the visual element is deselected;*
- The child cannot select/deselect a ball/sport field that is already correctly matched;
- The game ends when all associations are established.

The game functionality is represented by 32 visual states and 96 transitions between those states (Figure 59). It was decided to keep the same visual transitions identification, indicated in the state diagram correspondent to the original game interface (Figure 22).

Comparing this state diagram (Figure 61) with the other in Figure 53, it is possible to verify equality in the number of states (32) and visual transitions (96). The only difference between these two state diagrams is present in some components identification, responsible for global visual transitions (new visual components receiving user events are those related with *correct* visual state, instead of components related with the *selected* visual state). Twelve *SCs* are used now, instead of 18. Therefore, the algorithm implementation allows obtaining 6 *CCs* after performing the first iteration, in the same way it happens with game interface studied in Hypothesis One (Figure 54). Executing the *step 4* of the first iteration of the algorithm, results in a state diagram similar to the one in Figure 55, represented by 25 states and 78 visual interface transitions.

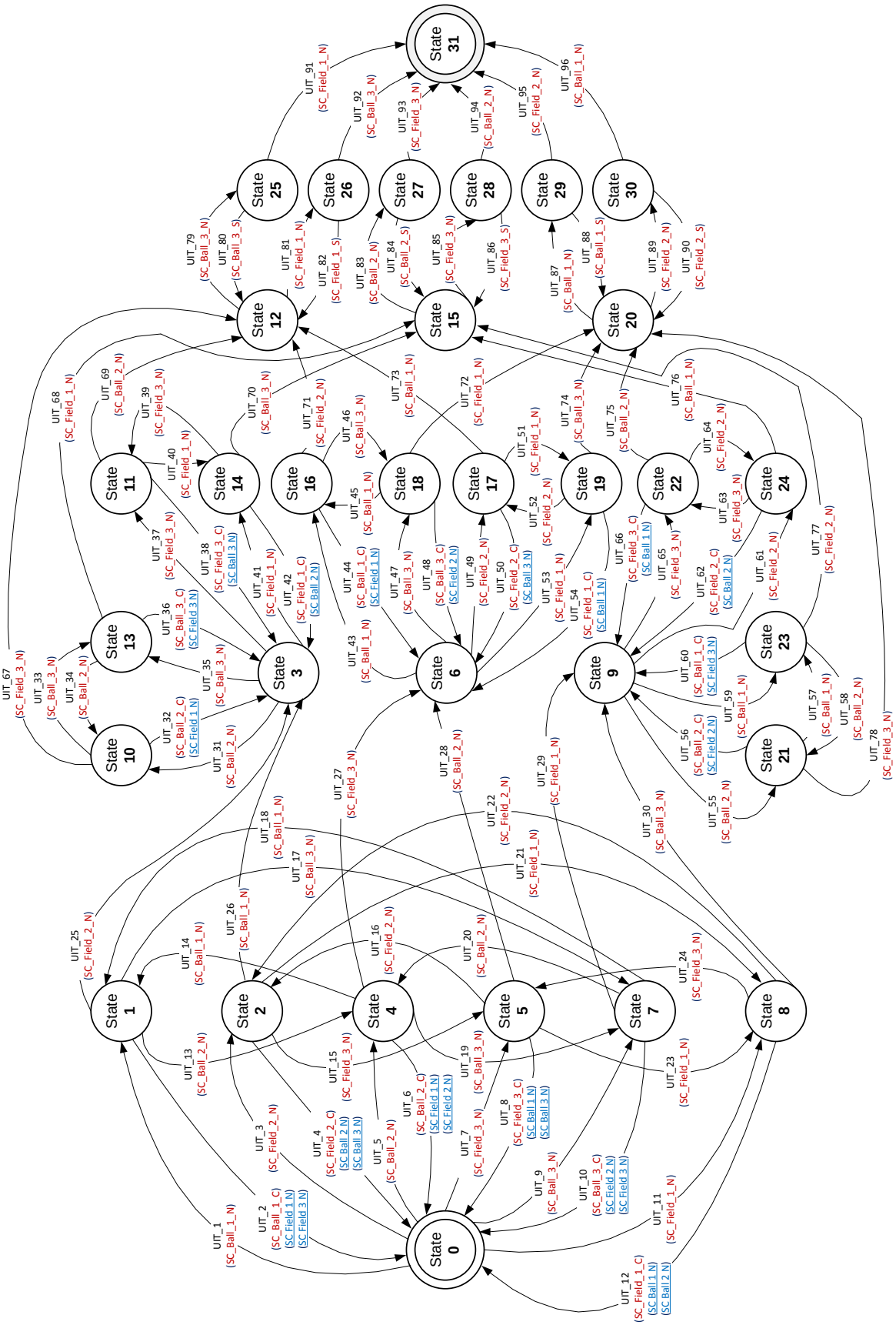

Figure 61: The state diagram representing the Hypothesis 3 of the new game interface.

The *step 5* of the algorithm indicates the possibility to repeat the previous steps, what happens in this case, by increasing the abstraction level of the game interface representation. The *step 1* of this new algorithm iteration results in a state diagram similar to the depicted in Figure 56. Also, two new *CCs* are clearly identifiable (in this case, beyond the 6 identified in the first algorithm iteration). Finally, the results obtained from the second algorithm iteration leads to a state diagram (identical to the one in Figure 57) which demonstrates no more interactive cycles available and as such, the algorithm finishes its execution. Thus, it was possible to identify *CCs* considering just 2 visual states. The identifications of *CCs* is not dependent on existing a minimum of 3 visual states.

6.2.3 Interface Variations Results

After the study of several hypotheses and the original visual game interface be redefined, it is possible to observe and to compare the obtained results (Table 45).

		ball	*field*	*1st abstraction level*	*2nd abstraction level*
Original Interface	*number of states*	3	2	3	1
	number of visual transitions	3	1		
	exists any cyclical transition between any 2 visual states?	yes	no		
Interface Variation A	*number of states*	3	3	6	2
	number of visual transitions	3	3		
	exists any cyclical transition between any 2 visual states?	yes	yes		
Interface Variation B	*number of states*	3	2	0	0
	number of visual transitions	2	1		
	exists any cyclical transition between any 2 visual states?	no	no		
Interface Variation C	*number of states*	2	2	6	2
	number of visual transitions	2	2		
	exists any cyclical transition between any 2 visual states?	yes	yes		

Table 45: Comparison between the 3 hypotheses considered.

As previously verified, from the state diagram of the original game interface, the algorithm just allows identifying the *CCs* concerned with the balls. It identifies 3 *CCs* at the 1st abstraction level and 1 *CC* at a higher abstraction level. Each ball is composed of 3 visual states and each sport field has 2 visual states. Both have transitions between those states, but only the ball has a cyclical transition. Also looking at components characteristics,

considered in the interface variations A and C, it is possible to confirm the existence of cyclical transitions between 2 visual states, independently of each ball and each sport field has two or three visual states. All *CCs* are identifiable (at both abstraction levels) when considering these two interface variations. From the characteristics of components used with the interface variation C, which do not consider the existence of cyclical transitions between 2 visual states, and considering also the number of visual states (the ball has 3 and the sport field has 2) none *CC* can be identified, whatever be the abstraction level.

The obtained results demonstrates that even without having a minimum of 3 visual states in a *CC*, since there exists one cyclical change between 2 states, it is possible to identify *CCs* in a state diagram, at various levels of components abstraction.

The following study is focused in presenting other possibility to simplify the visual interface design process, through the reuse features provided by *CCs* usage.

6.3 Reuse of Complex Components

Usually, the term *reusability* is easily related with the OOP paradigm and most of the times specifically related to *reusing code* (Ambler 1998). Other related term is *inheritance reuse* which refers to using the inheritance concept in an application, in order to take advantage of the behaviour implemented in existing classes. Other term is *component reuse* which refers to the use of prebuilt, fully encapsulated components, usually called *widgets*. They are typically self-sufficient and encapsulate only one concept. Usually, the *component reuse* concept differs from *code reuse*, because the programmer don't have access to the source code and it differs from *inheritance reuse* in that it doesn't use *sub classing* (new classes based in existing ones). Common examples of reusable software components are *Java Beans* and *ActiveX* components. There are many advantages in component reuse. First, it offers a greater scope of reusability than either code or inheritance reuse because components are self-sufficient (typically, *plug and play*). The main disadvantage of component reuse is: usually, components are small and encapsulate only one concept, which may imply the need of a large library of components in order to create an application (although when a component encapsulates one concept, it is a cohesive component). One possibility to simplify a visual interface specification process could be achieved by providing the interface designer with the ability to reuse pre-built visual components representations. Therefore, in the interface designing process, the selection of appropriate *AIOs* becomes

necessary. In the study here described, an *AIO* was selected: the *complex component*. Beyond this type of component supports visual appearance, topological composition and interaction (Teixeira-Faria & Rodeiro 2011), also supports OOP features (cf. Chapter 4). For example, a *CC* has features that can be related with the *aggregation* concept existent in OOP, which differs from ordinary *object composition* in that it does not imply ownership. Thus, by eliminating one of the containers will not imply to eliminate the objects it contains (the same happens with *CCs*). Each container can be identified as a class, which keep a list of their child components, and allow adding, removing, or retrieving components amongst their children. Knowing that OOP supports objects reuse (e.g. by association, aggregation or inheritance) (De Champeaux 1991), that feature is verified for *CCs* at semantic and functional levels. The semantic level was established as the possibility to change *CCs* visual appearance, maintaining its functionality (e.g. polymorphism in OOP). And thus, allowing to use components on different platforms (e.g. to be possible to change game graphics, while maintaining its functionality). The components reuse at functional level (e.g. inheritance in OOP) implies more profound changes in *CCs*, regarding its functionality (e.g. more or less visual states and transitions between them). Therefore, from an interface prototype designed using *CCs*, and assuming the existence of a particular *CC* (with a specific visual appearance and behaviour) which the designer wants to reuse in another interface, it will be verified if it can be done under considering those two perspectives.

6.3.1 Semantic perspective

When a user looks to the test bed user interface (cf. Chapter 3) he has at his disposal two perfectly distinct groups of visual elements (which correspond to three balls and three sport fields). As previously seen, the interface functionality can be implemented using *CCs* at two abstraction levels: in one of them, 6 *CCs* are used (each one corresponding to one ball or one sport field) and in the other abstraction level, 2 *CCs* are used (one corresponding to a group of balls and the other corresponding to a group of sport fields). A characteristic resulting from the use of *CCs* to represent a user interface is related to the ease of components reuse. In a first perspective to achieve that, the interface designer can create a new user interface component, maintaining its functionality. A new visual interface can be immediately obtainable, due to the fact of *CC* concept considers components reusability in its characteristics.

Figure 62: Original *CC_Balls complex component* (on the left) with new visual appearance.

If the interface designer wants to reuse a *CC* in another interface, keeping the functionality but with a different visual appearance, he can do it. This perspective is focused in drawing a new game interface by simply changing the component visual states, while still maintaining its functionality. Instead of the user (in this case, a child) has to relate balls with sport fields, he could for e.g. to relate objects with colours or sport shoes with balls (Figure 62). In this semantic *CCs* reuse perspective, the designer only has to be care to change visual presentation attributes.

6.3.2 Functional perspective

Another perspective of components reuse can be analysed considering changes in the functionality of the used *CCs*. On a first approach, the components functionality changes are related with the number of components contained inside a *CC* (number of visual elements to be used) (e.g. instead of using three balls and three sport fields, a reduction or an increase in the number of available components could be analysed, maintaining the components functionality). In OOP, a *class* is established as a base structure used to create instances of it (objects). A *CC* can be identified as an interface component (with visual appearance, composition properties and supporting user interaction) which can be compared to an OOP class. A general and introductory comparison was previously made (cf. Chapter 4). However, this part of the study will be focused on identifying *CCs* reuse features, comparing them with the OOP reuse provided by the concepts of *association*, *aggregation* and *inheritance* (Eck 2011).

Create Components by Association and Aggregation

An association represents a relationship between classes, and gives the common semantics and structure for many types of "connections" between objects. Associations are the mechanism that allows objects to communicate to each other through messages. Analogously, the communication between *CCs* associated with each other is performed by using *delegate events (DEs)/delegate actions (DAs)*.

Class Association

Each *CC* representing a ball and used to design the interface of the game previously referred can be compared to a class with 3 possible visual states (*normal*, *selected* and *correct*) and 3 methods responsible for changing those states (visual transitions). Also each *CC* representing a sport field can be identified as having features like a class with 2 visual states (*normal* and *correct*) and one method (visual transition). The structure of a ball and a field is represented in Figure 63.

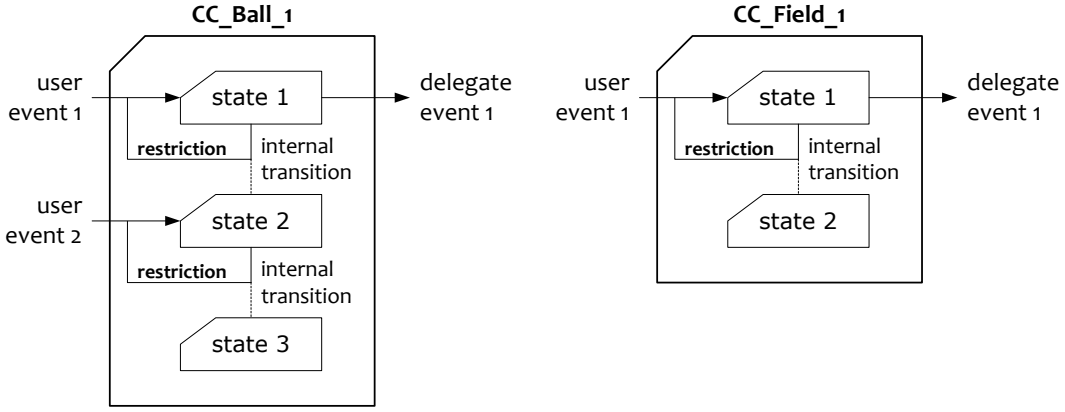

Figure 63: The structures of a ball (left) and a sport field (right) used in the game interface.

Each ball may receive 2 user events and triggers 1 *DE* (in other component) (Figure 63). It is also verifiable internal transitions between *CC* states, limited by preconditions. In the case of the sport field, it receives 1 user event and triggers 1 *DE*. It has an internal transition between the 2 states, whose triggering is dependent on a restriction. By establishing the relation between the *CC* concept and the class concept, 3 *CC_Ball* class instances and 3 *CC_Field* class instances need to be created, in order to implement the referred game interface. The relation between these balls and sport fields' classes can be established by OOP *association*, which defines a relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf. In this case, a similitude

with OOP method invocation exists, through the action performed by a *DA* triggered from a *CC*.

Class Aggregation

Aggregations are a special type of associations in which the participating classes don't have an equal status, but make a "whole-part" relationship. A *CC* also has features that can be related with the *aggregation* concept existent in OOP. The *CC_Balls* and the *CC_Fields* act as containers of 3 balls and 3 sport fields, respectively (Figure 64).

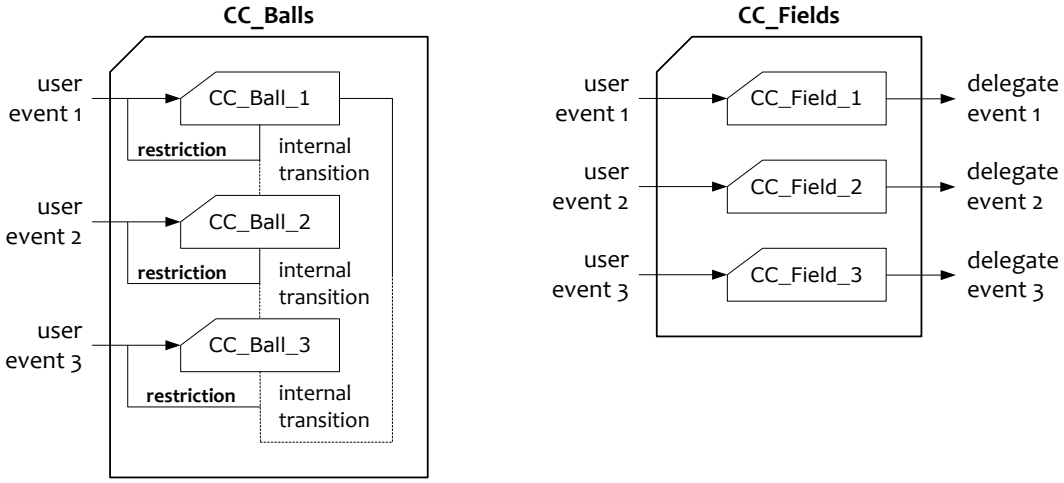

Figure 64: The structures of *CC_Balls* and *CC_Fields* containers used in the game interface.

However, *aggregation* differs from ordinary *composition*, on the perspective that it does not imply ownership. Thus, by eliminating one of the containers will not imply to eliminate the objects it contains. Each container could be identified as a class, which keep a list of their child components, and allow adding, removing, or retrieving components amongst their children.

Create Components by Inheritance

After analyzing *CCs* reuse approach, by using the components and maintaining its original states and visual transitions, it seems to be adequate to analyze components reuse through another perspective: *inheritance*, in which a *CC* can be modified for e.g. in order to contain more/less states and visual transitions or simply to add/remove *DAs* to/from the component. *CCs* are *interaction objects* which can be abstractly defined as objects with properties and methods. As a basic object, it shows properties of *visual appearance* and *topological composition* and its methods are the responses from user, other components or the system *interaction*. The *CCs inheritance* characteristic allows reusing features and

procedures and thus, it is possible for e.g. to create a new type of basic button from a ball (left side of Figure 51), through the elimination of one visual state and one of the transitions (T_3) (visually resulting for e.g. in left side of Figure 60). Thus, the class specification example of this new button had to be redefined. In a similar way, another type of button having four visual states could be designed from the previous basic button. This may involve for e.g. to add a new visual state and one more transition (T_4) (Figure 65).

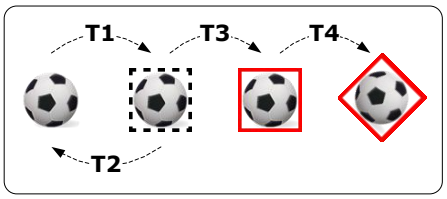

Figure 65: A new button with four visual states and four transitions.

The result is two new *CCs* related with two new types of buttons that can be created, both inheriting from the same basic button.

Complex Component Application Domain Change

In order to demonstrate the *CC* level of usage, it was decided to change the application domain. Thus, *CC_Balls* has been chosen to be reused as a toolbar visual component. As previously mentioned, it is possible to reuse a *CC*, by simply changing its visual appearance and hence its semantic. However, beyond the domain change, it is intended to change the number of states and visual transitions of the *CCs* that compose the chosen container *CC_Balls*. In this way, the following changes were decided to be performed:

- To increase the number of visual states: it is intended that each *CC* inside *CC_Balls* has one more visual state (e.g. each ball has three visual states and it is intended that each tool in the toolbar has four visual states);
- To increase the number of transitions between visual states: each *CC_Ball* inside *CC_Balls* contains three possible visual transitions between the three visual states. It is intended that each tool in the toolbar has five possible visual transitions between the four visual states.

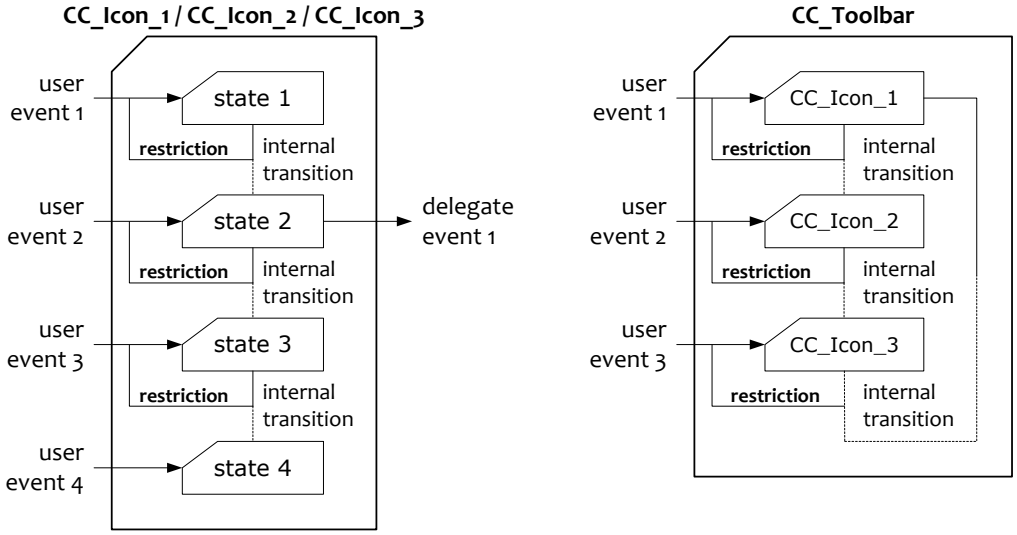

Figure 66: Three icons (3 equal instances on the left) and one toolbar (on the right) represented as CCs.

The structure of each tool (*CC_Icon_1*, *CC_Icon_2*, *CC_Icon_3*) inside (*CC_Toolbar*) is indicated on left side of Figure 66. Keeping in mind the possibility of *CCs* reusing, it is possible to verify a similitude with inheritance concept, when comparing it with OOP reuse. Basically, considering a *CC_Balls* component (which contains 3 balls) represented as a class, it can be reused as a toolbar (with 3 tools), through the creation of a class by inheritance and extending it to support a new visual state and to redefine the existent transitions, by keeping some of them, eliminating and creating other.

6.4 Conclusions

Two approaches concerned with the simplification of the design of user interfaces were introduced. The first approach intends to identify *CCs* from a state diagram. The second is concerned with the reuse properties of *CCs*. In the first approach a semi-automated process has been developed, in order to identify sets of *CCs*, at different abstraction levels, from a state diagram representing the interface's functionality. The process provided by the proposed algorithm allows to semi-automatically (meaning, visually perceived by the user interface designer when analysing a state diagram) to identify which *CCs* can be obtained from the description of a visual interface behaviour, since cyclical changes between *SCs* or *CCs* exists. From the visual interface aesthetic perspective, which can be represented by a set of drawings/visual elements obtained from a hand-made prototype, it is possible to

extract an identification of elements comprising a component structure identified as a *CC*. The equivalence class concept, with application to visual user interfaces, was used to propose a new concept called *equivalent state*. From the literature it was not possible to found any homologous approach to this concept. It was demonstrated that the proposed algorithm is able to identify *CCs* in a state diagram. This is a major contribution as it was not found any approach to this process in the literature. The proposed algorithm proved to be very promising in the verification of existing interfaces.

The second approach presented in this chapter allowed verifying that *CCs* supports components reusability, which contributes to simplify the user interface design. Given that a *CC* is independent from the user interface, and that it supports reuse flexibility, both at the component semantic perspective and at the functional perspective, a designer may reuse a *CC*, according with his needs, to create other visual user interfaces with different purposes. It was verified a correspondence between reuse properties from *CCs* with those available in object-oriented paradigm. This correspondence was verified under semantic and functional perspectives:

- Polymorphism (component semantic reuse): it is possible to change the visual appearance of a reused component by simply changing the related parameters;
- Inheritance (user interface functional reuse): it is possible to change the functionality of a reused component (and simultaneously the user interface functionality) by simply adding/removing DAs to/from that component.

These features enhance the customization abilities of visual and functional components, improving the versatility of a CC when compared with a typical *widget*. As a result, the *CCs* reuse properties contribute to increase their flexibility regarding the user interface design process.

The following step to be accomplished is the establishment of a XML specification language to support the *CC* concept, and thereafter to be possible to create complete and functional user interfaces.

Chapter 7

Proposed Specification and Implementation

7.1 Introduction

The study presented throughout this dissertation has a designer centred focus, since the objective is that the interface functionality can be freely established by the interface designer, including the visual elements to be used, independently of any platform or programming environment. The current chapter introduces a new XML-based specification, UIFD – *User Interface Free Design* (Teixeira-Faria & Rodeiro 2013), in order to support the abstract previously introduced *complex component* (*CC*) concept (cf. Chapter 4). Additionally, the second part of the chapter describes the implementation of a user interface prototyper (ProtoUIFD) able to generate a prototype of a complete and functional user interface from the UIFD specification.

As previously stated, and considering the user interface behaviour as a whole, there are three features to be considered regarding the development of a specification: visual presentation, components composition and dialog. Thus, the proposed UIFD specification considers that a *CC*:

- Supports visual appearance, components spatial positions and its own dialog;
- It has internal states triggered by self events (*SEs*) in result of user interaction;

- It has actions affecting other interface components (*delegate actions* (*DAs*)) and has events that come from other interface components (*delegate events* (*DEs*)).

Additionally, it is presented an implementation of the prototyping system, composed by 3 modules:

- *Base Module*: responsible for reading the UIFD document containing the user interface specification;
- *Nodes Calculation Module*: responsible for obtaining the global visual states;
- *Interaction Module*: responsible for visual interface graphical representation and user interaction management.

In order to perform practical tests and validations of the UIFD specification usefulness, the test bed user interface was used (cf. Chapter 3). Also, several adaptations to the user interface purpose were made, which implied to do the necessary changes to the UIFD specifications. The prototyper was used to take the specifications as input and to generate the desired user interfaces.

7.2 UIFD Specification

As previously stated, XML-UIDL languages have the advantage of being transparent to different interface technologies and to provide a uniform resource for heterogeneous communication modes. Hence, it was decided to develop a specification which allows supporting *CCs*. The specification name is UIFD and its structure it is divided in three parts:

- *Repository*: contains nominal definition of components (*simple* and *complex*) to be used in the library;
- *Library*: contains the *CCs* complete description, ready to be used in the interface design. It is possible to use multiple instances of the same repository component to create a library component;
- *Interface*: this structure supports the complete interface (obtained from *library* components) based in *CCs* and absolutely independent of any other interface components. It is called *complete complex component* (*CCC*).

Each one of these three specification parts is supported in a XML file having a correspondent *document type definition* (DTD) validation, with the purpose to support the documents structure, through a list of legal XML elements and attributes.

7.2.1 Repository

As stated in Chapter 5, when considering the *CC* concept, it is always necessary to establish an entrance level, in which *simple components* (*SCs*) to be used, and all possible visual transitions occurring between them are characterized. The *SCs* will be responsible for the interface visual appearance and all components nominal definitions (including CCs) are specified in *repository*.

Simple Components in Repository

A *SC* represents just one visual state. It can be described in three ways: graphic (based on graphical primitives e.g. lines, circles, rectangles), text or bitmap (based on pixels, by enumeration). The distinct component attributes values constitute one interface state. An excerpt of the XML specification used to represent one *SC*, having several established attributes, is listed below. It represents a visual frame having many other attributes.

```
<Simple-Component Name="name" Visible="true" Active="true">
    <Visual-Appearance>
      <Graphic>                                              (a)
        <Rectangle>
          <Coordinate>
            <Px>0</Px>
            <Py>0</Py>
          </Coordinate>
          <Coordinate>
            <Px>400</Px>
            <Py>400</Py>
          </Coordinate>
        </Rectangle>
        <LineStyle Style="continuous"/>
        <LineTickness>1</LineTickness>
        <LineColor>000000</LineColor>
        <FillColor>FFFFFF</FillColor>
        <Size Type="fixed">
          <ValueX>400</ValueX>
          <ValueY>400</ValueY>
        </Size>
      </Graphic>
    </Visual-Appearance>
</Simple-Component>
```

This previous specification example allows specifying a visual graphic component (**a**) that can be used, for example, in visual grouping of other components. It represents a rectangle primitive with properties indicated by XML tags, such as: *LineStyle*, *LineTickness*, *LineColor* and *FillColor*. Another example of a *SC* specification is presented below. It indicates that *SC* component is *visible* and *active*, and its visual appearance is based on an image file (obtained by enumeration (**b**)).

```
<Simple-Component Name="name" Visible="true" Active="true">
     <Visual-Appearance>
       <Enumeration>                                          (b)
         <File>file.png</File>
         <Size Type="fixed">
           <ValueX>70</ValueX>
           <ValueY>70</ValueY>
         </Size>
       </Enumeration>
     </Visual-Appearance>
</Simple-Component>
```

The specification of other *SCs*, used to design the test bed user interface, is similar to this one.

Complex Components in Repository

As previously indicated, similarly to *SCs*, *CCs* are also specified in the *repository* in its nominal definition. Each *CC* has a specification structure divided in three parts:

- ***Composition***: indicates all components (*simple* or *complex*) used to compose it;
- ***CC_States***: each *CC* can have several states and, for each state, it is possible to specify its:
  - 'status': establishment of the *visible* and *active* attributes values of each component, previously indicated in composition;
  - 'dialog_state': the state dialog is here specified, by indicating *SEs* supported by individual visual state;
- ***External_Events***: this part it used to specify the *DEs*/*DAs*, received/triggered from/to other components outside the present *CC*.

It is indicated below the basic structure of a *CC* specified in the *repository*.

```
<Complex-Component Name="name">
     <Composition></Composition>                            (c)
     <CC-States>                                            (d)
       <CC-State>
         <Status></Status>
         <Dialog-State></Dialog-State>
       </CC-State>
       …
     </CC-States>
     <External-Events>                                      (e)
       <Delegate-Events></Delegate-Events>
       <Delegate-Actions></Delegate-Actions>
     </External-Events>
</Complex-Component>
```

Considering the above *CC* structure, it is detailed below the three parts of the corresponding specification.

Composition

The first part of the *CC* specification is the hierarchical composition of each *CC*, indicated in *composition* (**c**) (e.g. below it is composed by two *SCs* indicated through its names).

```
<Composition>
   <SC>SC-Field-1-N</SC>
   <SC>SC-Field-1-C</SC>
</Composition>
```

CC_States

The second part of the *CC* specification concerns its visual states. Each component (*simple* and *complex*) can be composed by one or many visual states. The representation of an individual state of each component, available at a given moment of the user interaction, represents one global interface visual state. And, as previously referred, the global interface is represented by all its global visual states. The following XML specification represents one example of a *CC* visual state.

```
<CC-State ID="0" Visible="true" Active="true">
   <Status>
      <SC Name="SC-Field-1-N" Visible="true" Active="true" />
      <SC Name="SC-Field-1-C" Visible="false" Active="false" />
   </Status>
   <Dialog-State>
      <Self-Evt ID="1" Event="Event-X" Component="SC-Field-1-N" Ini-State="0"
                                                    End-State="1" />
         <Preconditions>
            <Pre_Cond Component="Component-X" State="Y" />
         </Preconditions>
   </Dialog-State>
</CC-State>
```

For each state available inside <CC-States> (**d**) it is possible to indicate:

- the <Status> element: to indicate the values of 'visible' and 'active' attributes of each *SC*;
- the <Dialog-State> element: in case of existing dialog, to indicate the *SEs* triggered over that particular visual state. For each *SE*, is necessary to prepare it to receive an event (**Event-X**, e.g. a mouse click) in result of an interaction with the *CC*, which will result in a transition between visual states;
  - the <Preconditions> element: to indicate any constraint to the *SE* occurrence. It is necessary to include the name of the other *CC* (**Component-X**) and its visual state identification (**Y**).

External_Events

A *CC* may "communicate" with its internal visual states or with other components external to it, through its *DEs*/*DAs* (to receive/send) in order to perform visual transitions between component states. The UIFD XML specification supports that kind of "communication" (**e**) detailed in the <External-Events> element. Once an event/action is received/triggered, the component is responsible for changing its visual state or the visual state of other *CC*.

```
<Delegate-Actions>
    <Trigger-DA ID="1" SELF_STATE="1" TO="Component-X" Trigger_DE_ID="Y" />
</Delegate-Events>
```

The above structure exemplifies one *DA* of a *CC* available in the *repository*. The attributes of <Trigger-DA> element are:

- the *DA* <ID> attribute: indicates the *DA* identification;
- the <SELF_STATE> attribute: when a visual state of a *CC* is achieved in result of an event, internal actions may be triggered (the *DAs*). This attribute indicates the visual state identification which, when achieved, will be responsible for triggering this *DA*;
- the <TO> attribute: the name of the other *CC*, which this *CC* is "connected" to, through this *DA* (e.g. **Component-X**);
- the <Trigger_DE_ID> attribute: to indicate the *DE* identification which will be triggered in other *CC* (e.g. **Y**).

Complex Components at a Higher Abstraction Level

The basic specification structures needed to create *CCs* at the first abstraction level were presented. It should be noted, in first abstraction level, each *CC* is not encapsulated in any other component. However, as previously stated, it is possible to increase the abstraction levels using *CCs*, by encapsulating components into other *CCs*. In practical terms of the UIFD specification, in order to simplify it, it was decided to specify just the <Composition> element at the new and higher abstraction level. Thus, it was decided to leave in first abstraction level the contents of <CC-States> and the <External-Events> elements.

Following is an example of its usage.

```
<Complex-Component Name="name">
     <Composition>
       <CC>CC-Field-1</CC>
       <CC>CC-Field-2</CC>
     </Composition>
</Complex-Component>
```

The above structure represents a *CC* composed by two other *CCs*, as an example of the encapsulation feature supported by this type of components.

7.2.2 Library

The second part of the UIFD specification structure is the *library*. It supports *SCs* and *CCs* specifications, available to be used to design the interfaces. Each *library* component is specified from instances of *repository* components (*simple* and *complex*), and their features are extended in order to complete the component expected functionality (including the behaviour concerning communication between components and the spatial position to represent them in the display). Taking the example of the XML representation of one *SC* available in *repository*, it can be used and extended, in order to be available in *library*.

```
<Simple-Component Name="name" Visible="true" Active="true">
     <Visual-Appearance>
       <Enumeration>                                          (b)
         <File>file.png</File>
         <Size Type="fixed">
           <ValueX>70</ValueX>
           <ValueY>70</ValueY>
         </Size>
         <Position>
           <Relative>
             <Coordinate>
               <Px>50</Px>
               <Py>35</Py>
             </Coordinate>
           </Relative>
         </Position>
       </Enumeration>
     </Visual-Appearance>
</Simple-Component>
```

The above specification was extended in order to include the graphical 2D position to represent the *SC* in the display. In the case of the one possible *CC* visual state, previously introduced, the specification in the *library* is also extended.

```
<CC-State ID="0" Visible="true" Active="true">
  <Status>
    <SC Name="SC-Field-1-N" Visible="true" Active="true" />
    <SC Name="SC-Field-1-C" Visible="false" Active="false" />
  </Status>
  <Dialog-State>
    <Self-Evt ID="1" Event="LeftClick" Component="SC-Field-1-N" Ini-
                                                State="0" End-State="1" />
      <Preconditions>
        <Pre_Cond Component="CC-Ball-3" State="1" />
      </Preconditions>
  </Dialog-State>
</CC-State>
```

It is possible to verify in the above structure the inclusion of the event responsible for triggering the *SE* (the **`LeftClick`**) and also the information related with the precondition considered (`Component="CC-Ball-3"` and `State="1"`). Other feature previously indicated is one *DA* of a *CC* available in the *repository*, which may be supported in the *library*.

```
<Delegate-Actions>
    <Trigger-DA ID="1" SELF_STATE="1" TO="CC-Ball-3" Trigger_DE_ID="1" />
</Delegate-Events>
```

The specification includes now the two attributes related to the other *CC*, which is "connected" with this one through the *DA* (`TO="CC-Ball-3"` and `Trigger_DE_ID="1"`).

7.2.3 Interface

The third and last part of the UIFD specification structure is the *interface*. As previously referred, it is possible to create an interface with any components (*simple* and *complex*) available in a *library*.

The typical structure is presented below.

```
<Interface Name="Interface-1-CC-2-SC">
    <Composition Source="Library.xml">
       <SC>
       <CC>
       …
    </Composition>

    <Interface-States>
       <Interface-State ID="0" Visible="true" Active="true">
         <Status>
         …
         </Status>
         <Dialog-State>
           <SC-Event ID="1" … >
           <SC-Event ID="2" … >
         </Dialog-State>
       </Interface-State>
       …
    </Interface-States>
```

```
    <Interface-External-Events>
      <Interface-Actions>
        <Trigger-DA ID="1" … />
        …
      </Interface-Actions>
    </Interface-External-Events>
</Interface>
```

The *interface* structure starts by establishing in `<Composition>` the components to be used from the *library*. Following, in the `<Interface-States>`, the `<Status>` and the `<Dialog-State>` of each interface state considered are indicated. And, finally, the *DAs* are specified in the `<Interface-External-Events>` sub-structure, in result of events triggered by user interaction with the interface. A concept introduced to indicate a particular interface, based in criterion 3 of the *complex component abstraction process* (cf. Chapter 4), is the *CCC*, representing a complete and functional user interface (following example).

```
<Complete-Complex-Component Name="UI">
  <Composition Source="Library.xml">
    <CC>CC-Ball-1</CC>
    <CC>CC-Ball-2</CC>
    <CC>CC-Ball-3</CC>
    <CC>CC-Field-1</CC>
    <CC>CC-Field-2</CC>
    <CC>CC-Field-3</CC>
  </Composition>
</Complete-Complex-Component>
```

It is clear the high abstraction level provided by this interface specification, since behaviour of all *CCs* is specified inside each component, including the *dialog* communication mechanism between these components.

7.2.4 UIFD Specification Example

In order to validate the applicability of the UIFD specification, it was decided to use it to specify the test bed user interface previously presented in Chapter 3. Following, the previously introduced concepts, the components available in *repository* and its typical structure are described.

Repository

As previously indicated, all components inside a *library* are created from components available in the *repository*. From the game interface example, 8 *CCs* were specified in the *repository* (*CC_Ball_1*, *CC_Ball_2*, *CC_Ball_3*, *CC_Field_1*, *CC_Field_2*, *CC_Field_3*, *CC_Balls*,

CC_Fields). Taking the *CC_Ball_1* as an example (representing the football ball) its three specification parts are detailed below, starting with the `<Composition>` element.

Composition

CC_Ball_1 is composed by three *SCs*.

```
<Composition>
   <SC>SC-Ball-1-N</SC>
   <SC>SC-Ball-1-S</SC>
   <SC>SC-Ball-1-C</SC>
</Composition>
```

Following, the visual states available in the *CC* are represented, enclosed by the `<CC_States>` element:

CC_States

The following specification structure corresponds to the second part of the *CC* structure.

It is verifiable that *CC_Ball_1* has 3 visual states. Two of them have one *SE* responsible for changing the component visual state (e.g. to change from `CC-State ID="0"` to `CC-State ID="1"`). The pending *SEs* (indicated by **`Event-X`**) are those which will be complete when used in the *library*.

```
<CC-States>
    <!-- Normal State of CC-Ball-1 -->
    <CC-State ID="0" Visible="true" Active="true">
       <Status>
         <SC Name="SC-Ball-1-N" Visible="true" Active="true" />
         <SC Name="SC-Ball-1-S" Visible="false" Active="false" />
         <SC Name="SC-Ball-1-C" Visible="false" Active="false" />
       </Status>
       <Dialog-State>
         <!-- Self-Event over SC-Ball-1-N -->
         <Self-Evt ID="1" Event="Event-X" Component="SC-Ball-1-N" Ini-
             State="0" End-State="1" />
       </Dialog-State>
    </CC-State>

    <!-- Selected state of CC-Ball-1 -->
    <CC-State ID="1" Visible="true" Active="true">
       <Status>
         <SC Name="SC-Ball-1-N" Visible="false" Active="false" />
         <SC Name="SC-Ball-1-S" Visible="true" Active="true" />
         <SC Name="SC-Ball-1-C" Visible="false" Active="false" />
       </Status>
       <Dialog-State>
         <!-- Self Event over SC-Ball-1-S -->
         <Self-Evt ID="2" Event="Event-X" Component="SC-Ball-1-S" Ini-
             State="1" End-State="0" />
       </Dialog-State>
    </CC-State>
```

```
    <!-- Correct state of CC-Ball-1 -->
    <CC-State ID="2" Visible="true" Active="true">
       <Status>
         <SC Name="SC-Ball-1-N" Visible="false" Active="false" />
         <SC Name="SC-Ball-1-S" Visible="false" Active="false" />
         <SC Name="SC-Ball-1-C" Visible="true" Active="false" />
       </Status>
    </CC-State>
</CC-States>
```

The last part of *CC_Ball_1* structure available in *repository* is the `<External-Events>` element.

External_Events

CC_Ball_1 continues to be following described.

```
<External-Events>
     <Delegate-Events>
         <Trigger-DE ID="1" Component="CC-X" Ini-State="1" End-State="2" />
     </Delegate-Events>
</External-Events>
```

CC_Ball_1 has one external event. It is a *DE* responsible for changing the component visual state (from `CC-State ID="1"` to `CC-State ID="2"`). The component is identified as `Component="CC-X"`. This event is triggered from outside the *CC*, meaning that other *CC* will be "connected" with this, in order to trigger the event.

The other *CCs* available in *repository* have a specification with a structure identical to the above.

Library

Still considering the *CC_Ball_1* in analysis, will be detailed bellow the evolution performed, from the *repository* to the *library* specification, in order to be possible to have a library of components to use in the interface design. Basically, the *CC* `<Composition>` structure remains the same, and the user interaction events over the components are specified in the `<Dialog-State>` inside of the `<CC-States>` structure. In the test bed user interface, it was decided that younger children use the mouse to interact and to play with the game, and the specific event decided to be used was the mouse *LeftClick* event.

```
<Self-Evt ID="1" Event="LeftClick" Component="SC-Ball-1-N" Ini-State="0" End-
 State="1" />
```

The above specification represents an example of a *SE* (having `ID="1"`) to be triggered from a user event (represented by a **`LeftClick`** event). The `<External-Events>`

components structure, available in *library*, is also updated from *repository*, in order to integrate other needed *DEs*/*DAs*, and to complete the component functionality. Thus, in the case of *CC_Ball_1* component, its specification will be updated to include two features:

- The possibility to deselect a ball from an external event received from other *CC* (one *DE* received to change from one visual state identified as `CC-State ID="1"` to other visual state identified as `CC-State ID="0"`);
- The possibility to execute two *DAs* which will trigger *DEs* in other two *CCs* (e.g. in the context of the game, once a sport ball passes to the selected visual state, any other sport ball must be deselected, according with the present game rules, just one ball can be selected at a time).

Therefore, the `<External-Events>` structure of the *CC_Ball_1* component will be updated and available in *library*.

```
<External-Events>
   <Delegate-Events>
     <!-- DE received (from Field) and if CC-Ball-1 is selected -->
     <Trigger-DE ID="1" Component="CC-Ball-1" Ini-State="1" End-State="2" />
     <! DE received (from other Ball) and if CC-Ball-1 is selected (to
      deselect the ball) OR DE received from (from Field) and if CC-Ball-1 is
      selected (when this is not the correct ball) -->
     <Trigger-DE ID="2" Component="CC-Ball-1" Ini-State="1" End-State="0" />
   </Delegate-Events>

   <Delegate-Actions>
     <!-- DAs sended to external components to deselect other balls -->
     <Trigger-DA ID="1" SELF-STATE="1" TO="CC-Ball-2" Trigger-DE-ID="2"/>
     <Trigger-DA ID="2" SELF-STATE="1" TO="CC-Ball-3" Trigger-DE-ID="2"/>
   </Delegate-Actions>
</External-Events>
```

From the above specification lines, it is possible to verify an evolution of the instance obtained from the *repository* and used in the *library*. In relation to external events, the component specification was extended and one more *DE* and two more *DAs* were included in specification, in order to complete the component functionality.

The other *CCs* to be included in the *library* (the other sport balls and sport fields) will have the necessary update in its specifications, from the components available in *repository*.

7.2.5 UI simplification

Two different usages of the components available in *library* can be considered, in order to study different *CCs* specification structures, at different abstraction levels. One of the approaches to create the game interface considers the usage of 6 *CCs* (3 balls and 3 sport

fields) and the other approach uses 2 *CCs* (the group of the balls and the group of the sport fields). By decreasing the number of *CCs* to be used to design the interface, through components encapsulation, that will increase the components abstraction and will simplify the interface specification. That is observable, in a simplified way, by verifying the two examples of `<Composition>` specification structures presented below, possible to be used to design the same user interface.

```
<Composition Source="Library-6-CC.xml">
  <CC>CC-Ball-1</CC>
  <CC>CC-Ball-2</CC>
  <CC>CC-Ball-3</CC>
  <CC>CC-Field-1</CC>
  <CC>CC-Field-2</CC>
  <CC>CC-Field-3</CC>
</Composition>

<Composition Source="Library-2-CC.xml">
  <CC>CC-Balls</CC>
  <CC>CC-Fields</CC>
</Composition>

<Complex-Component Name="CC-Balls">
  <Composition>
    <CC>CC-Ball-1</CC>
    <CC>CC-Ball-2</CC>
    <CC>CC-Ball-3</CC>
  </Composition>
  <Visual-Appearance></Visual-Appearance>
  <CC-States></CC-States>
  <External-Events></External-Events>
</Complex-Component>

<Complex-Component Name="CC-Fields">
  <Composition>
    <CC>CC-Field-1</CC>
    <CC>CC-Field-2</CC>
    <CC>CC-Field-3</CC>
  </Composition>
  <Visual-Appearance></Visual-Appearance>
  <CC-States></CC-States>
  <External-Events></External-Events>
</Complex-Component>
```

The results obtained from the proposed UIFD specification can be analysed under three perspectives. To analyze the results under a first perspective is important to do a comparison with previous study results (Rodeiro & Teixeira-Faria 2011), in which only *SCs* were used to build the game interface (cf. Chapter 3). The XML specification used in that study was divided in two files: a `<def>` file containing the *SCs* definition and a `<int>` file containing the global interface states and transitions between them. By doing a direct comparison of that approach with the approach used in the study presented here (using *CCs*) it is verifiable a huge difference in the number of lines needed to build the same visual

game interface. In the first approach, the interface specification contained in the `<int>` file has (~6000 lines) while in the case of using *CCs*, the number of specification lines needed is substantially lower (~600 lines). To analyze the results under a second perspective is important to reinforce that the game interface used in this study is always the same (having the same visual appearance, composition and behaviour). In this way, is possible to present a level of detail that has demonstrated the optimization achieved by applying this new *AIO* concept: the *complex component*. The introduction of it enables to establish components with an incremental level of abstraction, allowing a free and complete components customization at visual graphic level, composition and communication between components. Regardless the number of components used to design a UI, the capability introduced enables to increase abstraction (incrementally, by encapsulating components) and at the same time, this approach contributes to simplify the specification, which greatly reduces the designer tasks, especially if he wants to reuse components to create other user interfaces. The third perspective to adopt, when analyzing the obtained results, is based on (cf. Chapter 2) where is referred "*…the constant new emergence of UIDL-XML*". However, although some of these UI specification languages support UI elements definition, component composition and user interaction, none of the XML-UIDL analysed supports the possibility to create free design components independent of any platform.

The UIFD specification introduced in this chapter is focused in supporting UI design from low-fi prototyping, when considering the design process and the abstraction methods used here. UIFD offers the possibility to specify *SCs* and *CCs*, allowing the designer to establish the desired interaction, since it is supported by the final prototyping systems. One of those systems was built using Adobe Flash (*Actionscript 3*) technology (Adobe 2013a) and was used to verify and to validate the developed UIFD specification.

7.3 Prototyper Implementation

In many cases, a complete user interface representation becomes difficult to implement, because of the dependency on predesigned user interface components libraries (toolkits) that not allow user interface customization, as a user interface designer would like. It is imperative that a user interface defining process always takes in consideration the indications of user needs, in order to successfully accomplish the tasks that he needs/wants. Since a long time, a technique to achieve that is used and is called low-fidelity prototyping

("lo-fi" for short): the basic idea is building prototypes on paper (e.g. sketches) or using specific software and testing it with real users. The prototyping value is widely recognized (Rettig 1994) because it effectively educates developers to have a concern for usability and formative evaluation, and because it maximizes the number of times they get to refine the design before commit to code. After several tests performed with several different users, taking observations and redesigning the interface, there will be a simple description of visual interface behaviour (e.g. represented by a set of drawings/visual elements obtained from a hand-made prototype). Following, the next step will be the translation of those sketches to user interface components (usually called *widgets*) because most visual interfaces are created in order the user may interact with them, using those types of interactive visual components. However, usually, those visual components are not sufficient enough to represent a complete user interface (regarding the visual appearance, the components composition and the dialog). In many cases, because of the dependency on libraries of predesigned user interface components (toolkits) that not allow user interface customization, as a user interface designer would like. Hence, a process was developed. It allows, from the UIFD specification, to test and to validate a UI according with the user needs. Hereafter, it will be possible to refine the free components (and consequently the user interface) in order to use them.

7.3.1 Free Design of Components

As stated, each one of the previously referred three specification parts is supported on a XML (W3C Recommendation 2008) file with the correspondent DTD validation. The XML files constructing process is triggered by the designer when he's creating customized *CCs*, during the user interface design process. By knowing in advance the user interface functionality, it is possible to obtain a visual representation of it (as a state diagram) having states and transitions between components, in result of user interaction.

Taking the example of the 2D game interface for younger children, a simple schema representing the process to increase the simplification of a user interface representation (while the abstraction level to represent visual components increases) is shown in Figure 67. At the top of the figure, it is possible to verify the existence of 15 *SCs* (on the left side) that will successively be grouped into *CCs* (6 in the first abstraction level, 2 in the second and 1 *CCC* to represent the final user interface). At the bottom of the figure, 2 *CCs* are exemplified (*CC_Ball_1* and *CC_Field_1*). For each *CC*, it is possible to verify the visual states it contains

the restrictions (or preconditions), the internal visual transitions and the events it supports (user events and *DEs*). Thus, it is perceptible the amount of information that is encapsulated in each *CC*, and is also verifiable that, as complexity increases (and also the abstraction) at the same time the components are simplified and consequently the user interface representation.

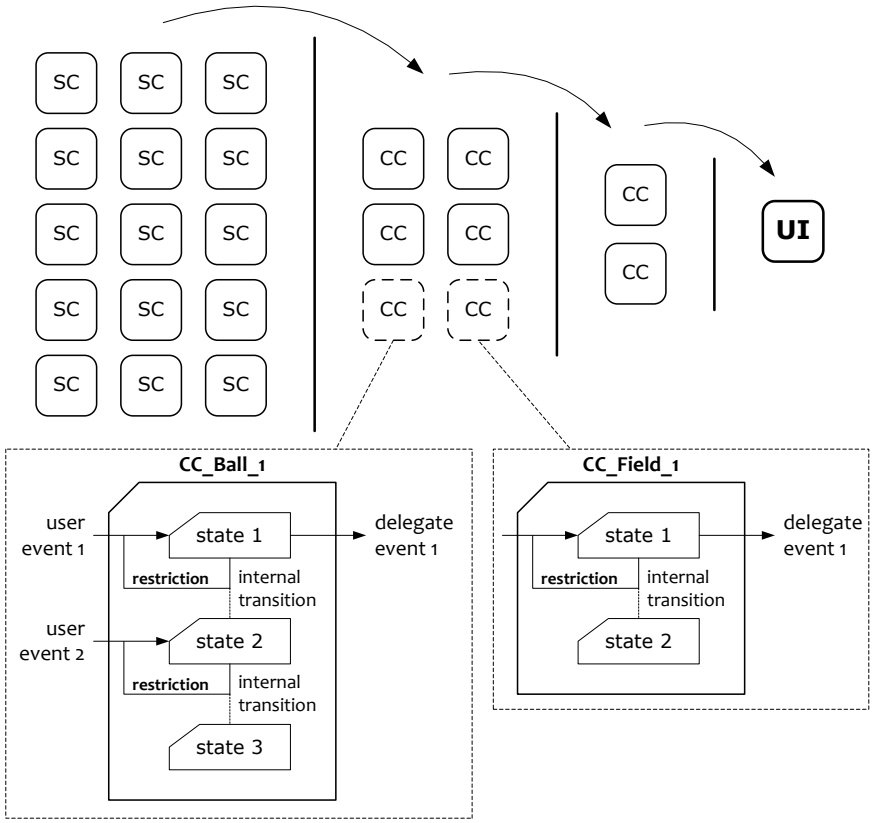

Figure 67: Decreasing the number of components as the abstraction level increases.

All components to be used are available in the common *repository* and the components customization is done at the *library* level, promoting components reuse and also contributing to simplify the interface design and behaviour, which can be freely established by the interface designer. This possibility of a designer to customize and to adapt a component according to his needs is a significant feature, enhancing the UIFD specification language versatility.

7.3.2 Data Structures Architecture

In order to validate the XML specification described in this chapter, an interface prototyper was implemented (ProtoUIFD). The developed tool allows to, automatically, read the XML UIFD specification and following, to represent a concrete interface prototype.

The obtained prototype is a functional representation of a complete user interface, provided with visual appearance and behaviour, ready to accept user interaction.

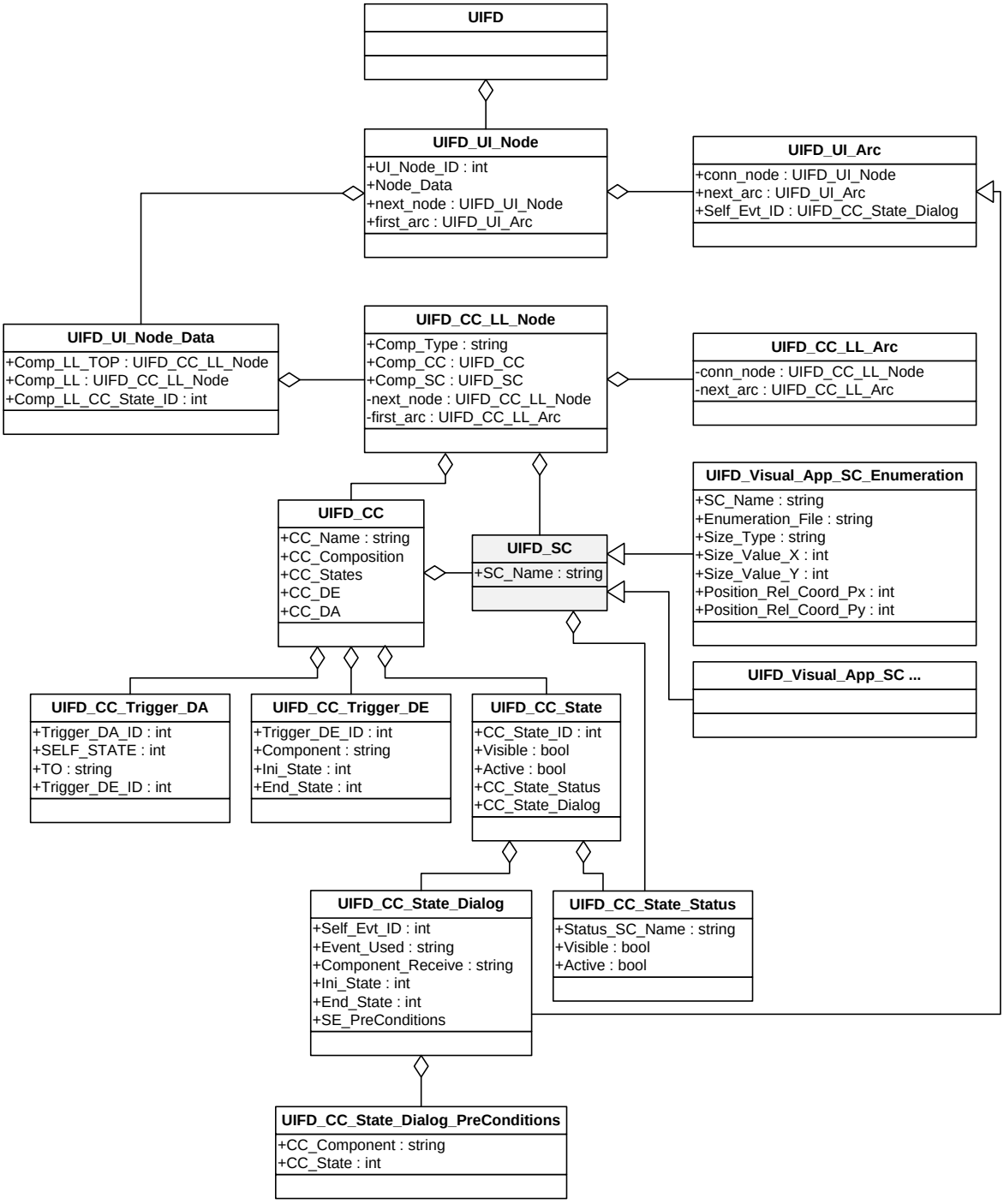

Figure 68: ProtoUIFD core class architecture (simplified).

A simplified diagram of the classes' main architecture is depicted in Figure 68, in respect to the interface prototyper software design and implementation. The diagram represents an overview of the application main classes used to achieve the experimental results here presented. The resulting framework architecture is rather flexible, enabling easy integration of new modules in the *CC* context.

The `UIFD_UI_Node` class supports all user interface global visual states and, the connection between those states, in result of user interaction, is stored in the `UIFD_UI_Arc` class. The `UIFD_UI_Node_Data` class stores a linked list (using the `UIFD_CC_LL_Node` and the `UIFD_CC_LL_Arc` classes) of components (*simple* or *complex*) that are being part of a specific interface global visual state. The `UIFD_CC` class is responsible for managing *SCs* and *CCs* available in this interface. In the case of *SCs* (`UIFD_SC`), several classes are available to support any graphical primitives (the `UIFD_Visual_App_SC_Enumeration` class is one of these examples, used to represent the images employed in the game interface previously created). In the case of *CCs*, the `UIFD_CC` class manages its visual states, through the `UIFD_CC_State` class. It supports the *SEs* triggered in result of user interaction with the components `UIFD_CC_State_Dialog` (including all preconditions available in `UIFD_CC_State_Dialog_Dialog_PreConditions`) and the *visible* and *active* properties of each *SC*, available in a *CC* (`UIFD_CC_State_Status`). The *DEs*/*DAs* of a *CC* are respectively managed by `UIFD_CC_Trigger_DE` and `UIFD_CC_Trigger_DA` classes.

The structure of classes was used in the prototype generation process, described in the following section.

7.3.3 Prototyper Modules

One of the major advantages of using *CCs* supported in the XML technology is its independence of any platform to represent a user interface. From several technologies available, it was decided to implement a first version of a prototyper (ProtoUIFD) using *Actionscript 3.0* supported by Adobe Flash (Adobe 2013b). Some advantages of this technology are:

- Currently, to be a technology widely distributed and able to be used by a large number of users;
- Supporting the visual design of graphical primitives and thereby to enable *CCs* visual representation, obtained from data structures (from the preloaded XML files). Despite being vector-based, it allows to use bitmaps when needed;

- Its flexibility allows to export exactly the same content through web browsers and platforms, without any extra code;
- Supports video, audio, animation, and advanced interactivity, which provides flexibility for future evolution of the UIFD specification and the prototyper which supports it.

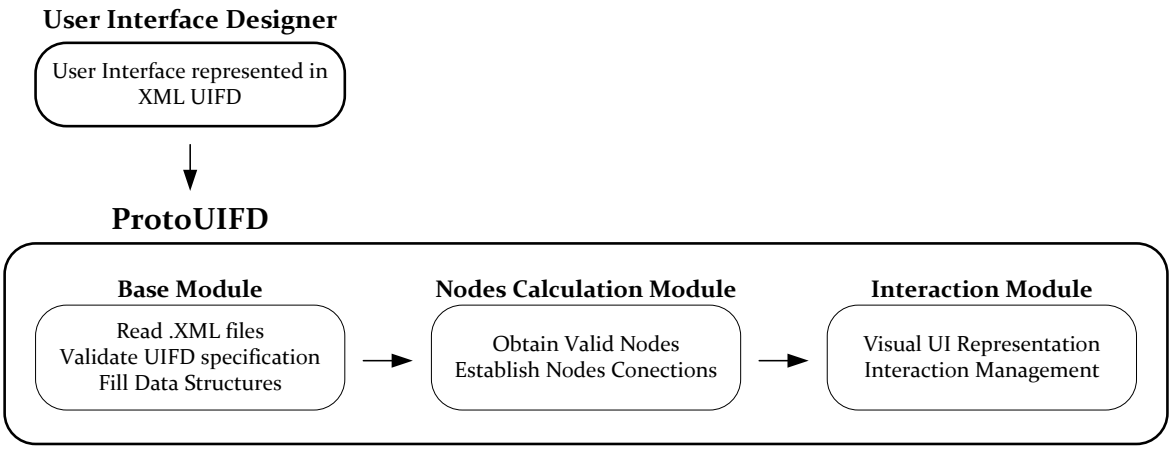

Figure 69: Protoype creation process.

A user interface prototype generation process/algorithm was implemented. This process is distributed over 3 modules (Figure 69) further explained.

ProtoUIFD – Base Module

This module is responsible for reading a .XML file containing the user interface specification, using E4X to obtain the data to be stored in data structures previously created (Figure 68). The components (*simple* and *complex*) that compose the complete user interface can be represented as a hierarchy of one components tree.

It is visible in Figure 70 the game interface logic data structure composed of 2 *CCs* (*CC_Balls* and *CC_Fields*) each one having three other *CCs*. Each ball (*CC_Ball_*) has 3 *SCs* and each sport field (*CC_Field_*) has 2.

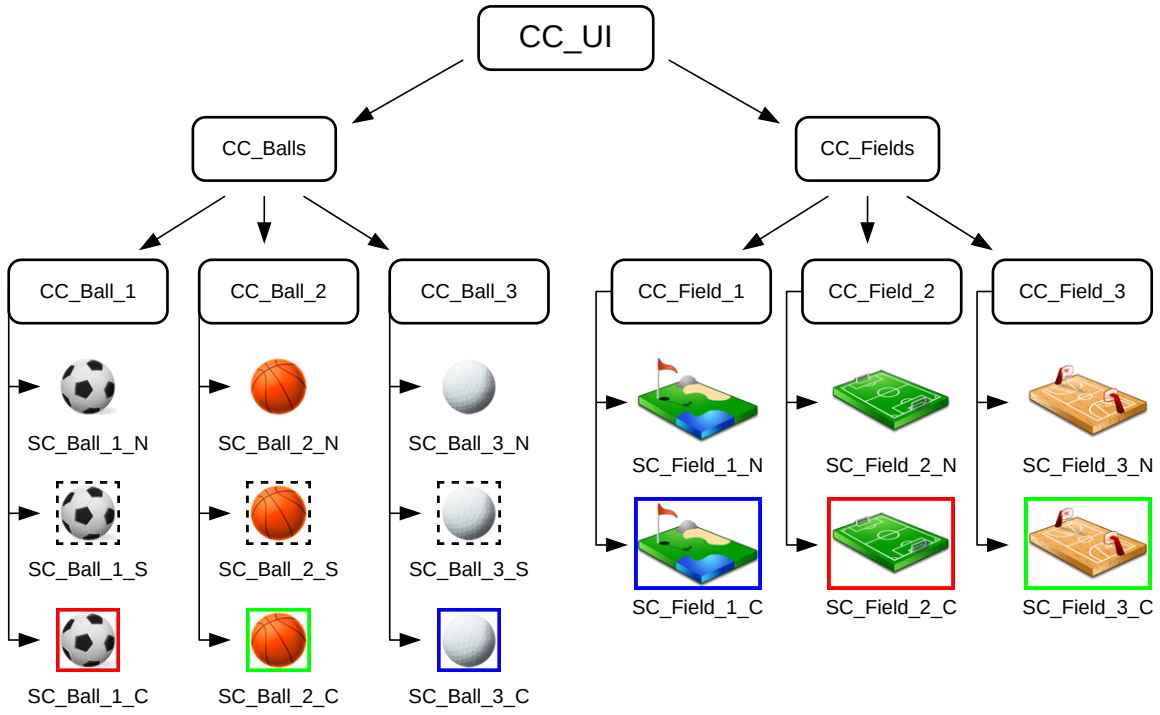

Figure 70: User interface *simple components* arranged in a tree.

In order to simplify the components traverse, the following algorithm step converts the components into another data structure in memory: a linked list (Figure 71), created through a recursive method. Each list node (left side of the figure) represents all *CCs* connected in sequence and has a list of connected arcs, representing the components (*simple* or *complex*) inside each individual node.

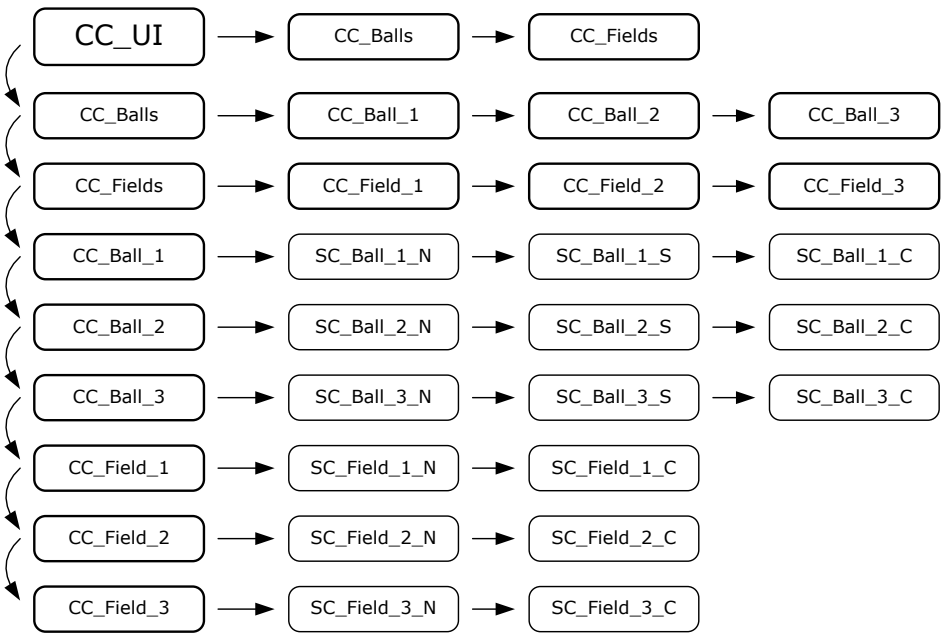

Figure 71: Linked list obtained from components tree.

After the components be arranged in a new data structure, the following step is to execute the Nodes Calculation Module.

ProtoUIFD – Nodes Calculation Module

In this module the global visual states are calculated. After concluding the components uploading process into the memory structures, is necessary to calculate the user interface global visual states and relate each other according with the dialog events established, to change from one visual state to the other. The objective is to obtain a state diagram representing the user interface functionality, with the global visual states (`UIFD_UI_Node`) and the existent transitions between them (`UIFD_UI_Arc`). Since the purpose at this point is to try to obtain user interface states as groups of components states, an algorithm to traverse the memory structure and to obtain global visual states was created. The previous linked list is traversed and all possible combinations of global states are generated from *SCs* (without considering any restrictions). Exemplifying for the original game, 216 combinations are obtained, which gives 216 potential global states. The `UIFD_UI_Node` class supports each one of those interface global visual states.

The following step concerns the selection of valid global visual states. In order to obtain them, an algorithm divided in three parts is executed.

Part I: Verification if a state change exists because of a SE

For each node being validated, the algorithm traverses the components structure and verify, for each *SE* of each state of each component if it has *Dialog*.

- if it doesn't has *Dialog*: the node is conditionally valid in this *Part I* and the algorithm will verify if it is valid in *Part II* of the algorithm;
- if it has *Dialog*: in case of not having preconditions, also the node is conditionally valid in this *Part I*. The process of preconditions verification is: for each *DA*, if its `SELF_EVENT` corresponds to the *dialog* `End_State`, and if the component state, indicated by the *DE* of the *DA*, is the same. If not, the node is *not valid*.

Each node that is not valid in this *Part I* returns *invalid node*, and just one being valid in this part, will then be verified if it is also valid in *Part II*.

Part II: Verification if a state change exists because of a DA

For each *DA*, the components are traversed and the algorithm verify if its name is equal to the `TO` indicated by the *DA*. If it is, then check whether the correspondent *DE* meets its

conditions, i.e., if any *DE* in which its `Ini_State` is equal to the present state, it returns *invalid node*.

Part III: Creation of transitions between components

After finding and selecting the valid global states, the algorithm will traverse the nodes linked list and, for each node, will compare with the others, in order to create transitions between them (arcs). Thus, for each component, a similar process to the above one is executed:

- For each component of that node in analysis, the algorithm verifies for each *SE* of each state of each component having *Dialog* (including preconditions) which node follows the "conditions", and can be achieved (creating the corresponding arc);
- Verifying the "conditions" is also similar to the previous process.

In the last step of the algorithm the prototyper will graphically represent the visual user interface. That is responsibility of the Interaction Module following introduced.

ProtoUIFD – Interaction Module

After running the algorithm to obtain the global visual states and once the state diagram correspondent to the UI prototype is created (represented by a linked list), this module is activated in order to be responsible for two tasks:

- Visual interface graphical representation (Figure 72). The prototyper will graphically represent the visual interface, in order the user may interact with it and test the interface prototype, which in this case is the 2D game interface;
- Interaction Management. From the information available in the nodes and in the arcs, the visual components are drawn in the display, supporting the correspondent listeners to the interaction device decided to be used (e.g. the mouse) and concerned with the *SEs* with responsibility to change the interface global state.

Figure 72: User interface represented using the UIFD prototyper (ProtoUIFD).

The small bar at the top of the window provides access to the menu of the prototyper (partially hidden). Additionally, in Figure 72, is observable some information regarding the algorithm execution:

- Time needed to create the UI prototype (milliseconds);
- Number of potential Global States;
- Number of valid Global States;
- Number of events which trigger valid Visual Transitions.

This information is available in order to be possible to verify and to compare it between different user interfaces, and thus to be aware of the UI complexity, according with the number of components, visual transitions and preconditions, used to build the interface.

7.3.4 Components Customization

Several customizations/adaptations were done to the components used in the test bed user interface, in order to test reuse features supported by *CCs*. It was decided to increase one ball and one sport field to the original game (having 4 balls and 4 sport fields). To achieve that, it was only necessary to add 4 more *DAs* (one to each ball (CC_Ball_n)) and it was not necessary to add any event to the sport fields (except to indicate which new ball is related to the new sport field, through the already existent *DA*). Following, a certain update to the UIFD specification regarding one of the balls is exemplified. By adding one XML line, representing one *DA*, becomes possible to deselect a fourth ball included in the game.

```
<External_Events>
  <Delegate_Events>
    <Trigger_DE ID="1" Component="CC_Ball_1" Ini_State="1" End_State="2" />
    <Trigger_DE ID="2" Component="CC_Ball_1" Ini_State="1" End_State="0" />
  </Delegate_Events>
  <Delegate_Actions>
    <Trigger_DA ID="1" SELF_STATE="1" TO="CC_Ball_2" Trigger_DE_ID="2" />
    <Trigger_DA ID="2" SELF_STATE="1" TO="CC_Ball_3" Trigger_DE_ID="2" />
    <Trigger_DA ID="3" SELF_STATE="1" TO="CC_Ball_4" Trigger_DE_ID="2" />
  </Delegate_Actions>
</External_Events>
```

Other components customizations of the original 2D game were tested. One of the tests was the conversion of one *CC* (CC_Balls) to behave as a toolbar with 3 buttons. The only thing it was necessary to change was the component visual presentation, by replacing the images that were used to visually represent a group of balls. With this simple change, a toolbar becomes ready to be used. Other modifications to this toolbar were done, by increasing the number of buttons and changing their images (Figure 73). As previously indicated, it is also possible to compare the UIs according with the time needed to create the UI prototype, the number of potential and valid global states and the number of events available to trigger valid visual transitions.

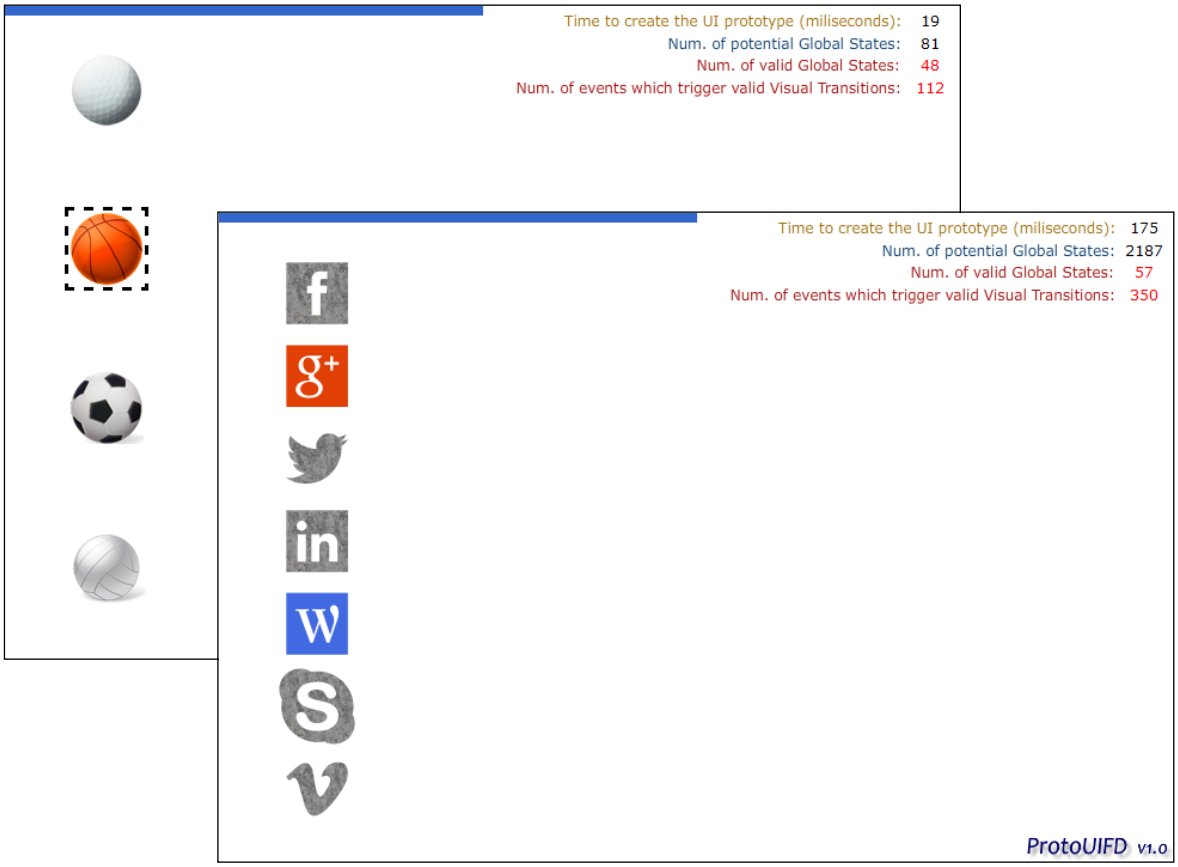

Figure 73: Two examples of the *CC* used in the case study adapted to be a toolbar.

Following, another customization was experimented: from the previous toolbar, it was decided to increment the number of options available and to connect this toolbar with other *CC* (the CC_Fields of the original game, with the necessary changes) to display images correspondent to each pressed button of the toolbar, and be able to deactivate the correspondent toolbar button when the displayed image was clicked (Figure 74). Two changes were introduced (in addition to the name of the components). In the case of the original toolbar, was necessary to add two more *DAs* to each button (related with showing/hiding the correspondent image displayed by the other *CC* (CC_Fields):

```
<External_Events>
  <Delegate_Events>
    <Trigger_DE ID="1" Component="CC_Icon_4" Ini_State="1" End_State="2" />
    <Trigger_DE ID="2" Component="CC_Icon_4" Ini_State="1" End_State="0" />
  </Delegate_Events>
  <Delegate_Actions>
    <Trigger_DA ID="1" SELF_STATE="0" TO="CC_Img_Big_4" Trigger_DE_ID="2" />
    <Trigger_DA ID="2" SELF_STATE="1" TO="CC_Img_Big_4" Trigger_DE_ID="1" />
    <Trigger_DA ID="3" SELF_STATE="1" TO="CC_Icon_1" Trigger_DE_ID="2" />
    <Trigger_DA ID="4" SELF_STATE="1" TO="CC_Icon_2" Trigger_DE_ID="2" />
    <Trigger_DA ID="5" SELF_STATE="1" TO="CC_Icon_3" Trigger_DE_ID="2" />
  </Delegate_Actions>
</External_Events>
```

The original CC_Fields was adapted. It was necessary to add one more internal *CC* (obtained from a CC_Field_n, and in order to have 4 images available). Each component representing an image was updated in order to have two *DEs* responsible for, internally, change the correspondent visual state, resulting from the *DAs* received from the toolbar.

```
<External_Events>
  <Delegate_Events>
   <Trigger_DE ID="1" Component="CC_Img_Big_1" Ini_State="0" End_State="1" />
   <Trigger_DE ID="2" Component="CC_Img_Big_1" Ini_State="1" End_State="0" />
  </Delegate_Events>
  <Delegate_Actions>
    <Trigger_DA ID="1" SELF_STATE="0" TO="CC_Icon_1" Trigger_DE_ID="1" />
  </Delegate_Actions>
</External_Events>
```

In the UI example represented in Figure 74, stands out also the numbers of valid global states and events available to trigger valid visual transitions, which are substantially higher than the previous examples.

Figure 74: Example of one of the toolbar evolutions/adaptations.

Finally, is import to reinforce that the user of the system presented here does not need to have any programming knowledge, but the ability to handle with UIs, because is only necessary to establish the components and its behaviour (always at the user interface level). With the first version of the implemented prototyper, it was possible to simulate a visual representation of all possible global visual states, and to support interaction with the components, which are responsible for triggering transitions between those visual states.

7.4 Conclusions

The study presented in this chapter is a major contribution to support the freely design of user interfaces. A XML-based specification language to design user interfaces was proposed: UIFD. The versatility of the UIFD language allows to specify *SCs* and *CCs* and to simplify visual interface representation. The simplification obtained with the UIFD specification is given by the decreasing number of components available, as the abstraction level increases. This happens in virtue of encapsulation property available in *CCs* (fewer components are being used from lower to higher abstraction level). As stated, the use of this

type of component, being incrementally more complex and at the same time being at a more abstract level, simplifies the process of designing and prototyping the user interface.

There is not a restriction regarding the application domain of the UIFD specification, since it allows simulating input, output and UI conditions. Thus, a prototyper can be created and used to simulate and to test several kinds of user interfaces, and contribute to a variety of user interface platforms and interaction modalities. The original test bed user interface was specified using UIFD and then loaded by the developed prototyper (ProtoUIFD), in order to visually represent the user interface and to allow simulate the user interaction with the components.

The approach described in this chapter allows representing the interface functionality, supporting the CFFI criteria, according with the goals established for this research. The user interface prototyper validates the XML specification, which supports the new *complex component* concept and enables to specify the user interface as a whole.

Chapter 8

Conclusions and Future Work

Although most of contemporaneous software systems use more or less elaborated components to build their interfaces, it was not found in literature any approach similar to the proposed *complex component* concept. This concept enables to represent complete visual interface functionality, based in interactive free design components. In many cases, the dependency on predesigned user interface components libraries (*toolkits*) limits the interface designer regarding interface customization, both visual and functional.

From the literature, it was found that none of the reviewed methods completely complies with the established criteria – *CFFI*, concerning the objective of being possible to freely specify complete functional user interface prototypes. Thus, an approach to a new *complex component* abstraction concept was proposed. A test bed user interface was defined and used in order to illustrate and compares the application of specification methods. This test bed was later used to validate the abstraction process and the resulting simplification of the specification process.

Then, a deep insight on the new component usage was sought. A new algorithm to identify components from state diagrams was proposed. The reusability provided by *complex components* was also demonstrated. The ultimate goal pursued by both approaches is the simplification of visual user interfaces design process.

In order to provide the proposed methods with effective tools, a new XML-based specification language was introduced. Moreover, a user interface prototyper supporting the

new XML specification was described and implemented. The prototyper has been used to generate prototypes of complete functional user interfaces.

8.1 Main Contributions

This dissertation has presented work that addressed several issues that contribute to the complete achievement of the main research goals. A key objective was concerned with the introduction of a new abstraction concept – the *complex component* – that enables to specify customizable interface components. Another key objective was to develop of a new method that makes use of the new *complex component* concept in order to simplify the specification process of complete user interfaces. This new component concept approach links together several features as can be observed in Figure 75.

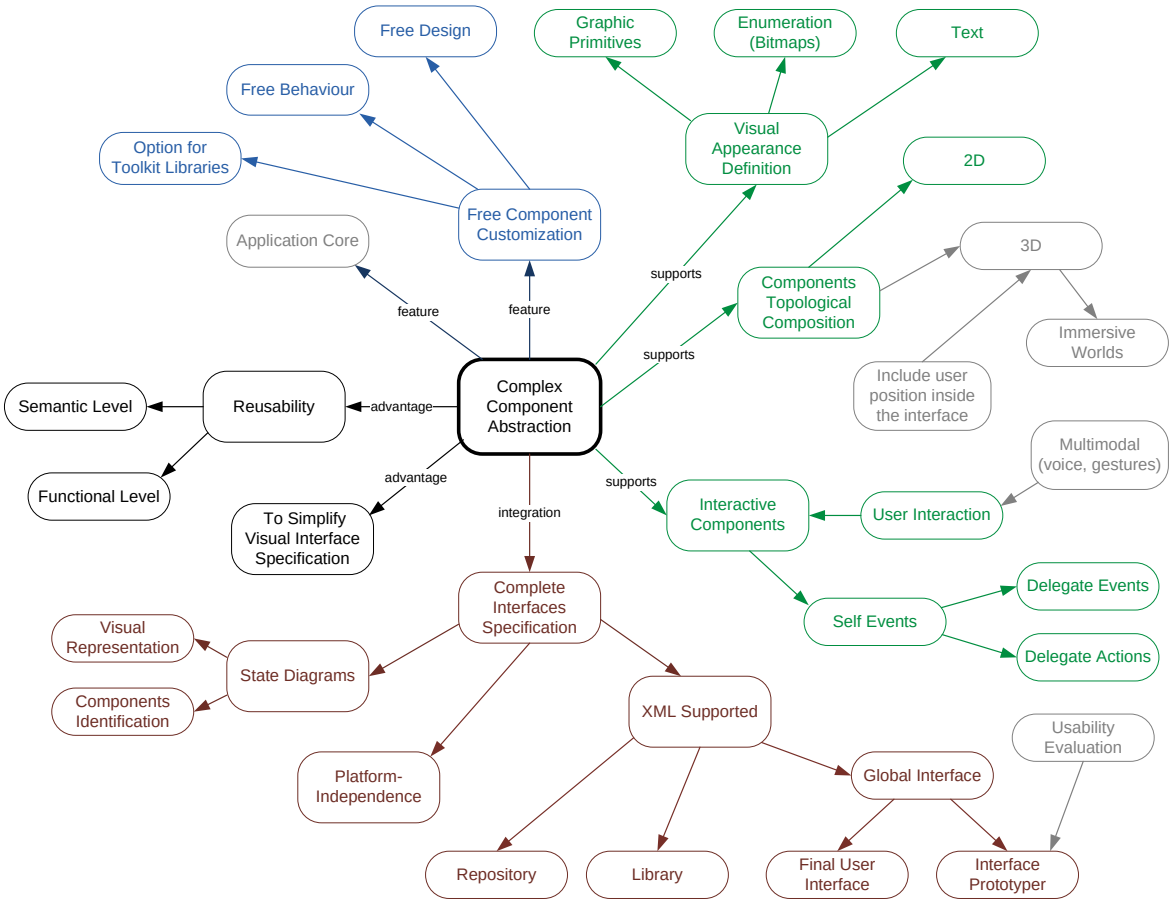

Figure 75: *Complex component* application overview.

This dissertation has made the following main contributions:

- The definition of a set of criteria – *CFFI* – that specification methods should meet in order to enable the specification of complete and functional user interface prototypes;
- A comprehensive survey of specification methods. The methods were compared and analyzed according with the *CFFI* criteria;
- A component abstraction – *complex component* – was introduced in order to enable the development of a specification method that fully complies with the *CFFI* criteria. The new concept was established as a result of the assembly of relevant features available in existing methods, and by the introduction of new features. The *complex component* was designed in order to support semantic and functional reusability.
- A new method to abstract *complex components* was designed and proposed. This method was validated and compared with a previous method using a test bed user interface. It was demonstrated that as the user interface abstraction increases trough *complex components*, more simplified becomes the user interface design;
- A new original algorithm was introduced in order to identify *complex components* from state diagrams. This algorithm enables to simplify the state diagram representation by reducing the number of represented states and transitions.
- A XML-based specification language – *User Interface Free Design* (UIFD) – was introduced in order to support the specification using the newly proposed *complex component*.
- A prototyping system was developed – ProtoUIFD. The prototyper is able to generate a prototype of a complete and functional user interface from UIFD documents. The prototyper was further used to demonstrate the validity of the proposed concepts and methods.

Parts of the work described in this thesis were accepted in international peer-reviewed conferences and resulted several published papers:

- Teixeira-Faria, P., Rodeiro, J.: Complex Components Abstraction in Graphical User Interfaces. Proceedings of the 14th international conference on Human-computer interaction: design and development approaches, Springer-Verlag LNCS 6761, pp. 309–318 (2011);
- Rodeiro, J., Teixeira-Faria, P.: User Interface Representation Using Simple Components. Proceedings of the 14th international conference on Human-

computer interaction: design and development approaches, Springer-Verlag LNCS 6761, pp. 278–287 (2011);

– Teixeira-Faria, P., Rodeiro, J.: An interface prototyper supporting free design components specification. Proceedings of the 15th international conference on Human-Computer Interaction: human-centred design approaches, methods, tools and environments, Springer-Verlag LNCS 8004, pp. 490–499 (2013).
– Rodeiro, J., Teixeira-Faria, P.: Visual interfaces design simplification through components reuse. Proceedings of the 15th international conference on Human-Computer Interaction: human-centred design approaches, methods, tools and environments, Springer-Verlag LNCS 8004, pp. 441–450 (2013);
– Rodeiro, J., Teixeira-Faria, P.: Interactive Complex Components Identification from State Diagrams. Proceedings of the IADIS International Conferences: Interfaces and Human Computer Interaction and Game and Entertainment Technologies 2013, in K. Blashki (Ed.) pp. 189-196 (2013).

8.2 Future Work

Some future directions are identified:

– Multimodal Interaction: to increase the interaction module features in order to support voice, gestures and other interaction modalities;
– 3D Integration: to evolve the UIFD specification in order to support 3D interface components, able to respresent the observer position and orientation in a immersive 3D world, and thus leverage the creation of 3D user interfaces;
– Usability Evaluation Tool: to develop and to integrate a usability evaluation module to be used to contribute to improve the user interfaces usability;
– Application Core Integration: to support calls to the application core in order to enable data exchanges with the interface components;
– UIFD IDE: to build an IDE supporting UIFD specification and able to integrate individual modules being created.
– Components Identification Tool: to develop an automatic tool to identify *complex component* from low-fidelity prototypes;

- Algorithm Optimization: to optimize the proposed algorithm to identify *complex components* from a state diagram (presented in Chapter 6);
- Code Generator Tool: to create a final user interface code generator, with bindings to several languages/platforms;
- Global Repository: to create a global and shared repository that supports *complex components*; thus, the designer's contributions become available in an online platform.

REFERENCES

Abrams, M. & Phanouriou, C., 1999. UIML: An XML Language for Building Device-Independent User Interfaces. In *Proceedings of XML '99*. XML '99. Philadelphia.

Adobe, 2013a. Adobe ActionScript 3. *ActionScript Technology Center*. Available at: http://www.adobe.com/devnet/actionscript.html [Accessed December 9, 2013].

Adobe, 2013b. Adobe Flash Platform. *Flash Professional*. Available at: http://www.adobe.com/pt/products/flash.html [Accessed November 1, 2013].

Adobe, 2009. Adobe MXML. *Adobe MXML - Functional and Design Specification*. Available at: http://opensource.adobe.com/wiki/display/flexsdk/MXML+2009 [Accessed May 19, 2010].

Alencar, P. et al., 1995. *A Formal Approach to Design Pattern Definition & Application*, Ontario, Canada: University of Waterloo. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.5451.

Alencar, P. et al., 2002. A Logical Theory of Interfaces and Objects. *IEEE Transactions on Software Engineering*, 28(6), pp.548–575.

Ambler, S., 1998. A realistic look at object-oriented reuse. *Journal Software Development*, 6(1), pp.30–38.

Ball, T. et al., 2000. Sisl: Several Interfaces, Single Logic. *International Journal of Speech Technology*, 3(2), pp.93–108.

Bauer, G. et al., 2005. Luxor - XML UI Language (XUL) Toolkit. *XML UI Language (XUL) Toolkit*. Available at: http://luxor-xul.sourceforge.net/ [Accessed November 1, 2011].

Beard, S. & Reid, D., 2002. MetaFace and VHML: A First Implementation of the Virtual Human Markup Language. In AAMAS02 Embodied Conversational Character Workshop. Bologna, Italy, pp. 1–7.

Bodart, F. et al., 1994. Towards a Dynamic Strategy for Computer-Aided Visual Placement. In *Proceedings of the workshop on Advanced visual interfaces, AVI '94*. AVI '94. Bari, Italy: ACM, pp. 78–87.

Bodart, F., Noirhomme-Fraiture, M. & Vanderdonckt, J., 1993. Guidelines for choosing interaction objects. In *Proceedings of the Vienna Conference on Human Computer Interaction*. VCHCI '93. Vienna, Austria: Springer Verlag, pp. 431–432.

Bodart, F. & Vanderdonckt, J., 1996. Widget Standardization through Abstract Interaction Objects. In *Proceedings of 1st International Conference on Applied Ergonomics*. ICAE '96. Istanbul, Turkey: Springer Verlag, pp. 300–305.

Bojanic, P., 2007. The Joy of XUL. *Mozilla Developer Network*. Available at: http://developer.mozilla.org/en/docs/The Joy of XUL [Accessed November 1, 2011].

Calvary, G. et al., 2003. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers*, 15(3), pp.289–308.

Calvary, G. et al., 2005. The CAMELEON reference framework. CAMELEON Project. *Cameleon reference framework*. Available at: http://www.w3.org/2005/Incubator/model-based-ui/wiki/Cameleon_reference_framework [Accessed October 27, 2011].

Carneiro, L., Cowan, D. & Lucena, C., 1993. *ADVcharts: a Visual Formalism for Highly Interactive Systems*, Department of Computer Science, University of Waterloo. Available at: http://www.cs.uwaterloo.ca/research/tr/1993/20/93-20.pdf [Accessed April 24, 2011].

Carnero, S., 2007. *Sistematización de la validación de Interacción del usuario sobre la visualización en Interfaces de Usuario usando Especificación Abstracta*. PhD Thesis. Universidad de Vigo.

Carr, D., 1994. Specification of Interface Interaction Objects. In *CHI '94 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. Boston, USA: ACM, pp. 372–378.

De Champeaux, D., 1991. Object-Oriented Analysis and Top-Down Software Development. In *ECOOP'91 European Conference on Object-Oriented Programming*. ECOOP'91. Geneva, Switzerland: Springer-Verlag, pp. 370–376.

Coad, P. & Yourdon, E., 1990. *Object-Oriented Analysis* 2nd Edition., Prentice-Hall.

Cowan, D. & Lucena, C., 1995. Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. *IEEE Transactions on Software Engineering*, 21(3), pp.229–243.

Crowle, S. & Hole, L., 2003. ISML: An Interface Specification Meta-Language. In *Proceedings of the 10th International Workshop on Design, Specification and Verification of Interactive Systems*. DSV-IS '2003. Madeira Island, Portugal: Springer, pp. 362–376.

Van Dam, A., 1997. Post-WIMP User Interfaces. *Communications of the ACM*, 40(2), pp.63–67.

Dermler, G. et al., 2003. Flexible pagination and layouting for device independent authoring. In *Proceedings of the Emerging Applications for Wireless and Mobile Access Workshop*. WWW2003.

Devlin, K., 2004. *Sets, Functions, and Logic: An Introduction to Abstract Mathematics* 3rd ed., Chapman & Hall/ CRC Press.

Dicker, J. & Cowan, B., 2008. Platforms for Interface Evolution. In *CHI 2008 Workshop Proceedings*. CHI 2008. Florence, Italy.

Dix, A. et al., 2004. *Human Computer Interaction* 3rd ed., Harlow, UK: Addison-Wesley.

Duke, D. et al., 1994. Unifying Views of Interactors. In *Proceedings of Advanced Visual Interfaces '94*. AVI '94. Bari, Italy, pp. 143–152.

Duke, D. & Harrison, M., 1993. Abstract Interaction Objects. *Computer Graphics Forum*, 12(3), pp.25–36.

Eck, D., 2011. *Introduction to Programming Using Java* 6th ed., New York, USA. Available at: http://math.hws.edu/javanotes/.

Faconti, G. & Paternò, F., 1990. An approach to the formal specification of the components of an interaction. In *Proceedings of the European Computer Graphics Conference*. Eurographics '90. Montreux, Switzerland: Elsevier North-Holland, Inc., pp. 481–494.

Figueroa, P., 2009. InTml: A Case Study on Virtual Reality Development. In *Proccedings of the Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '2009. Florida, USA: ACM, pp. 745–746.

Figueroa, P., Green, M. & Hoover, H., 2001a. 3DML: A Language for 3D Interaction Techniques Specification. In Eurographics. Manchester, United Kingdom: The Eurographics Association 2001.

Figueroa, P., Green, M. & Hoover, H., 2001b. 3DML. VR Applications for Non-Programmers. Available at: http://webdocs.cs.ualberta.ca/~pfiguero/3dml/ [Accessed May 17, 2010].

Figueroa, P., Green, M. & Hoover, H., 2002. InTml: A Description Language for VR Applications. In *Proceedings of the 7th International Conference on 3D Web Technology*. Web3D '02. Arizona, USA: ACM, pp. 53–58.

ForeUI, 2012. ForeUI. *Easy-to-use UI prototyping tool*. Available at: http://www.foreui.com/ [Accessed May 25, 2012].

GladeXML, 2003. GladeXML. *Glade - A User Interface Designer*. Available at: http://search.cpan.org/~mlehmann/Gtk2-GladeXML-0.94/GladeXML.pm [Accessed May 17, 2010].

Gobël, S. et al., 2006. A Device-Independent Multimodal Mark-up Language. In INFORMATIK 2006. Dresden, Germany, pp. 170–177.

Gottfried, Z., Vanderheiden, G. & Gilman, A., 2002. Universal Remote Console - Prototyping for the Alternate Interface Access Standard. In *Proceedings of the User interfaces for all 7th international conference on Universal access: theoretical perspectives, practice, and experience*. Lecture Notes in Computer Science (LNCS). ERCIM '02. Paris, France: Springer, pp. 524–531.

Guerrero-Garcia, J. et al., 2009. A Theoretical Survey of User Interface Description Languages: Preliminary Results. In *LA-WEB '09 Proceedings of the 2009 Latin American Web Congress (la-web 2009)*. 2009 Latin American Web Congress. Merida, Mexico: IEEE Computer Society, pp. 36–43.

Harel, D., 1987. Statecharts: a visual formalism for complex systems. *Journal Science of Computer Programming*, 8(3), pp.231–274.

Harrison, M. & Duke, D., 1994. A Review of Formalisms for Describing Interactive Behaviour. In *Proceedings of the Workshop on Software Engineering and Human-Computer Interaction*. ICSE '94. Sorrento, Italy: Springer-Verlag, pp. 49–75.

Hutchings, H. & Pierce, J., 2005. *DIAMOND: A Framework for Dividing Interfaces Across Multiple Opportunistically aNnexed Devices*, IBM Research.

Hutchins, E., Hollan, J. & Norman, D., 1985. Direct Manipulation Interfaces. *Human-Computer Interaction*, 1(4), pp.311–338.

Hypercard, 1992. Hypercard. *Hypercard*. Available at: http://hypercard.org/ [Accessed April 20, 2012].

Jacob, R., 1986. A Specification Language for Direct Manipulation User Interfaces. *ACM Transactions on Graphics (TOG)*, 5(4), pp.283–317.

Johnson, P. & Parekh, J., 2003. *Multiple Device Markup Language A Rule Approach*, Chicago, USA: DePaul University.

Katsurada, K. et al., 2003. XISL: A Language for Describing Multimodal Interaction Scenarios. In *International Conference on Multimodal Interfaces*. Vancouver, Canada: ACM, pp. 281–284.

Kost, S., 2006. *Dynamically generated multi-modal application interfaces*. PhD Thesis. Germany: Technical University of Dresden and Leipzig University of Applied Sciences.

Krasner, G. & Pope, S., 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3), pp.26–49.

Landay, J., 1996. SILK: sketching interfaces like krazy. In *CHI '96 Conference companion on Human factors in computing systems*. New York, USA: ACM, pp. 398–399.

Lauridsen, O., 1995. Abstract Specification of User Interfaces. In *CHI '95 Conference companion on Human factors in computing systems*. Colorado, USA: ACM, pp. 240–241.

Leite, J., 2006. A Model-Based Approach to Develop Interactive System Using IMML. In *Proceedings of the 5th international conference on Task models and diagrams for users interface design*. Lecture Notes in Computer Science. TAMODIA '06. pp. 68–81.

Lin, J. et al., 2002. Denim: An informal sketch-based tool for early stage webdesign. In *Proceedings of AAAI 2002 Spring Symposium*. Sketch Understanding Workshop. Stanford, CA.

LiquidApps, 2009. LiquidApps. *LiquidApps*. Available at: http://www.liquidappsworld.com/ [Accessed November 5, 2011].

Markopoulos, P., 1997. *A compositional model for the formal specification of user interface software*. PhD Thesis. University of London.

Meixner, G., Paterno', F. & Vanderdonckt, J., 2011. Past, Present, and Future of Model-Based User Interface Development. *i-com*, 10(3), pp.2–11.

Meixner, G. & Thiels, N., 2008. Tool Support for Task Analysis. In *Workshop on User Interface Description Languages for Next Generation User Interfaces*. CHI'08 26th Annual CHI Conference on Human Factors in Computing Systems. Florence, Italy: ACM, pp. 76–80.

Microsoft Corp, 2012. Microsoft Visual Studio. *Visual Studio*. Available at: http://msdn.microsoft.com/en-us/vstudio [Accessed May 17, 2012].

Microsoft Corp, 2011. XAML. *XAML*. Available at: http://msdn.microsoft.com/en-us/library/ms752059.aspx [Accessed April 6, 2010].

Mori, G., Paternò, F. & Santoro, C., 2003. Tool Support for Designing Nomadic Applications. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*. IUI '03. Florida, USA: ACM, pp. 141–148.

Mozilla Developer Center, 2010. XUL. *XUL*. Available at: https://developer.mozilla.org/en/XUL [Accessed April 6, 2010].

Muller, A., Forbrig, P. & Cap, C., 2001. Model-Based User Interface Design Using Markup Concepts. In *Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification*. DSV-IS '01. Scotland, UK: Springer-Verlag, pp. 16–27.

Myers, B., 1990. A New Model for Handling Input. *ACM Transactions on Information Systems*, 8(3), pp.289–320.

Myers, B., 1995. User interfaces software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(1), pp.64–103.

Navarre, D. et al., 2009. ICOs: a Model-Based User Interface Description Technique dedicated to Interactive Systems Addressing Usability, Reliability and Scalability. In *Transactions on Computer-Human Interaction*. ACM SIGCHI. Boston, USA: ACM, pp. 1–56.

Negroponte, N., 1994. *Being Digital*, New York: Vintage Books.

NetBeans, 2012. NetBeans. *NetBeans*. Available at: https://netbeans.org/ [Accessed May 15, 2012].

Ngo, D. & Byrne, J., 2001. Another Look at a Model for Evaluating Interfaces Aesthetics. *International Journal of Applied Mathematics and Computer Science*, 11(2), pp.515–535.

Ngo, D., Teo, L. & Byrne, J., 2002. Evaluating Interface Esthetics. *Knowledge and Information Systems*, 4(1), pp.46–79.

Norman, D., 1988. *The Phychology of Everyday Things*, Basic Books.

Okazaki, N. et al., 2002. A Multimodal Presentation Markup Language MPML-VR for a 3D Virtual Space. In *Workshop on Virtual Conversational Characters: Applications, Methods, and Research Challenges*. Melbourne, Australia.

Paternò, F., 1994. A Theory of User-interaction Objects. *Journal of Visual Languages and Computing*, 5(3), pp.227–249.

Paternò, F., Mancini, C. & Meniconi, S., 1997. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*. INTERACT '97. Australia, pp. 362–369.

Paternò, F., Santoro, C. & Spano, L.D., 2009. MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments. *ACM Transactions on Computer-Human Interaction*, 16(4).

Phanouriou, C., 2000. *UIML: A Device-Independent User Interface Markup Language*. Virginia: Faculty of the Virginia Polytechnic Institute and State University.

Picard, E. et al., 2003. *Atelier de composition d'ihm et évaluation du modèle de composants*, Livrable l3, RNTL ASPECT, Laboratoire I3S, Université de Nice.

Preece, J., Sharp, H. & Rogers, Y., 1994. *Human-Computer Interaction: Concepts and Design*, Addison-Wesley.

Puerta, A. & Einstein, J., 2001. XIML: A Universal Language for User Interfaces. Available at: http://www.ximl.org/Docs.asp [Accessed January 17, 2010].

Quentin, L. & Vanderdonckt, J., 2004. Comparing Task Models for User Interface Design. In *The Handbook of Task Analysis for Human Computer Interaction*. Lawrence Erlbaum Associates, pp. 135–154.

Rettig, M., 1994. Prototyping for tiny fingers. *Communications of the ACM*, 37(4), pp.21–27.

Righetti, X., 2006. *Study of Prototyping Tools for User Interface Design*. Bachelor. Université de Genéve.

Rodeiro, J., 2001. *Representácion y Análisis de la Componente Visual de la Interfaz de Usuario*. PhD Thesis. Universidad de Vigo.

Rodeiro, J. & Teixeira-Faria, P., 2013a. Interactive Complex Components Identification from State Diagrams. In *Proceedings of the IADIS International Conferences: Interfaces and Human Computer Interaction and Game and Entertainment Technologies 2013.*. Prague, Czech Republic: IADIS Press, pp. 189–196.

Rodeiro, J. & Teixeira-Faria, P., 2011. User Interface Representation Using Simple Components. In *Proceedings of the 14th International Conference on Human-Computer Interaction. Design and Development Approaches*. HCII '11. Orlando, USA: Springer-Verlag, pp. 278–287.

Rodeiro, J. & Teixeira-Faria, P., 2013b. Visual interfaces design simplification through components reuse. In *Proceedings of the 15th International Conference on Human-Computer Interaction.*. HCII '13. Las Vegas, USA: Springer-Verlag, pp. 441–450.

Rodríguez, H. et al., 2010. Interacción Multimodal con Espacios Virtuales, un caso de estudio: Museo Virtual 3D MultiModal. In *XI Congreso Internacional de Interacción Persona-Ordenador, Interacción 2010*. INTERACCION '2010. Valencia, Spain.

Rodríguez, H. et al., 2007. XMMVR: Especificación y Arquitectura para el desarrollo de aplicaciones de Interacción Multimodal en Escenarios 3D. In *VI Congreso Internacional de Interacción Persona-Ordenador, Interacción 2007*. INTERACCION '2007. Zaragoza, Spain.

Rogers, Y., Sharp, H. & Preece, J., 2011. *Interaction Design: Beyond Human Computer Interaction* 3rd ed., John Wiley & Sons.

Rosenberg, J. et al., 1988. UIMSs: Threat or Menace? In *Proceedings of the ACM CHI 88*. Human Factors in Computing Systems. Washington, pp. 197–200.

Rouillard, J., 2003. Plastic ML and its toolkit. In *Proceedings of the Human-Computer Interaction 2003*. HCI '2003. Crete, Greece, pp. 612–616.

Savidis, A., 2004a. Supporting Virtual Interaction Objects with Polymorphic Platform Bindings in a User Interface Programming Language. In *International Workshop on Rapid Integration of Software Engineering Methods (RISE 2004)*. Luxembourg: Springer Verlag, pp. 11–23.

Savidis, A., 2004b. *The I-GET user interface Programming Language: User's Guide, Technical Report, ICS-FORTH*, Available at: ftp://ftp.ics.forth.gr/tech-

reports/2004/2004.TR332.IGET User Interface Programming Language.pdf [Accessed March 29, 2011].

Savidis, A. & Stephanidis, C., 2006. Automated user interface engineering with a pattern reflecting programming language. *Automated Software Engineering*, 13(2), pp.303–339.

Schaefer, R., Steffen, B. & Wolfgang, M., 2006. Dialog Modeling for Multiple Devices and Multiple Interaction Modalities. In *Proceedings of the 5th International Conference on Task Models and Diagrams for User Interface Design*. TAMODIA '2006. Hasselt, Belgium: Springer Verlag, pp. 39–53.

Schlungbaum, E. & Elwert, T., 1996. Dialogue Graphs - A Formal and Visual Specification Technique for Dialogue Modelling. In *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*. Sheffield Hallam University: Springer, p. 13.

Shneiderman, B., 1998. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* 3rd ed., Addison-Wesley.

Siebelink, R., 2000. Towards a profile driven service authoring and adaptation platform. Available at: http://www.w3.org/2000/10/DIAWorkshop/siebelink.htm [Accessed May 17, 2010].

Silva, P., 2000. User Interface Declarative Models and Development Environments: A Survey. In *Proceedings of the 7th international conference on Design, Specification and Verification of Interactive Systems*. DSV-IS '2000. Springer Verlag, pp. 207–226.

Smith, J., 1996. *ISO and ANSI ergonomic standards for computer products. A guide to implementation and compliance*, New York: Prentice-Hall.

Souchon, N. & Vanderdonckt, J., 2003. A Review of XML-compliant User Interface Description Languages. In *DSV-IS - Interactive Systems, Design, Specification, and Verification*. Springer Verlag, pp. 377–391.

Sousa, L. & Leite, J., 2004. XICL — An Extensible Mark-up Language for Developing User Interface and Components. In *Proceedings of Fourth International Conference on Computer-Aided Design of User Interfaces*. CADUI '2004. Madeira, Portugal, pp. 245–256.

Szekely, P., 1995. User interface prototyping: Tools and techniques. In *Software Engineering and Human-Computer Interaction*. Springer-Verlag, pp. 76–92.

Teixeira-Faria, P. & Rodeiro, J., 2013. An interface prototyper supporting free design components specification. In *Proceedings of the 15th International Conference on Human-Computer Interaction.*. HCII '13. Las Vegas, USA: Springer-Verlag, pp. 490–499.

Teixeira-Faria, P. & Rodeiro, J., 2011. Complex Components Abstraction in Graphical User Interfaces. In *Proceedings of the 14th International Conference on Human-Computer*

Interaction. Design and Development Approaches. HCII '11. Orlando, USA: Springer-Verlag, pp. 309–318.

Tractinsky, N., 1997. Aesthetics and Apparent Usability: Empirically Assessing Cultural and Methodological Issues. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '97. Atlanta USA, pp. 115–122.

UIML, 2009. UIML. *UIML 3.1, Draft Specification*. Available at: http://www.uiml.org/specs/uiml3/DraftSpec.htm [Accessed November 13, 2011].

Vanderdonckt, J. et al., 2004. USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In *Proceedings of the ACM AVI2004 Workshop Developing User Interfaces with XML Advances on User Interface Description Languages*. AVI '2004. Italy: Citeseer, pp. 55–62.

Vitzthum, A., 2006. SSIML/Components: A Visual Language for the Abstract Specification of 3D Components. In *Proceedings of the eleventh international conference on 3D web technology*. Web3D '06. Maryland, USA: ACM, pp. 143–151.

W3C Recommendation, 2011a. CCXML. *Voice Browser Call Control: CCXML Version 1.0*. Available at: http://www.w3.org/TR/ccxml/ [Accessed May 17, 2010].

W3C Recommendation, 2009. EMMA. *EMMA: Extensible MultiModal Annotation Markup Language*. Available at: http://www.w3.org/TR/emma/ [Accessed May 14, 2010].

W3C Recommendation, 2003. InkML. *Ink Markup Language*. Available at: http://www.w3.org/TR/2003/WD-InkML-20030806 [Accessed May 18, 2010].

W3C Recommendation, 2011b. SVG. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. Available at: http://www.w3.org/TR/SVG/ [Accessed October 26, 2011].

W3C Recommendation, 2000. VoiceXML. *VoiceXML 2.0*. Available at: http://www.w3.org/TR/voicexml/ [Accessed November 11, 2011].

W3C Recommendation, 2012. XForms 2.0. *XForms 2.0*. Available at: http://www.w3.org/TR/2012/WD-xforms20-20120807/ [Accessed April 6, 2010].

W3C Recommendation, 2002. XHTML. *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)*. Available at: http://www.w3.org/TR/xhtml1/ [Accessed October 26, 2011].

W3C Recommendation, 2008. XML. *XML 1.0 (Fifth Edition)*. Available at: http://www.w3.org/TR/REC-xml/ [Accessed June 12, 2010].

Web3D, 2011. Web 3D Consortium. *Web 3D Consortium*. Available at: http://www.web3d.org/x3d/ [Accessed February 13, 2011].